ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2012-ENST-00xx

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité ”Communication et Électronique“

présentée et soutenue publiquement par

Guillaume BARBU

le 3 Septembre 2012

De la sécurité des plateformes Java Card™ face aux attaques matérielles

Directeur de thèse: **Philippe HOOGVORST**
Co-encadrement de la thèse: **Guillaume DUC**
Encadrement industriel: **Vincent GUERIN**

Jury

Mme Assia TRIA, Responsable dép. SAS, ENSMSE/CEA-LETI	Rapporteur
M. Jean-Louis LANET, Professeur, XLIM, Université de Limoges	Rapporteur
M. Viktor FISCHER, Professeur, Lab. Hubert Curien, Université de Saint-Étienne	Examinateur
M. Renaud PACALET, Professeur, Lab. SoC, Télécom ParisTech	Examinateur
M. Pascal URIEN, Professeur, dép. INFRES, Télécom ParisTech	Examinateur
M. Philippe HOOGVORST, Chercheur, dép. COMELEC, Télécom ParisTech	Directeur de thèse
M. Vincent GUERIN, Java Card & GlobalPlatform Group Manager, Oberthur Technologies	Encadrant
M. Éric VÉTILLARD, Java Card Principal Product Manager, Oracle Inc.	Invité

TELECOM ParisTech
école de l’Institut Mines-Télécom - membre de ParisTech

THÈSE

Remerciements

Je saisis l'occasion qui m'est donnée ici de remercier chaleureusement toutes les personnes qui m'ont accompagné au cours de cette thèse et de la rédaction de ce manuscrit: Merci!

Ah ! Non ! C'est un peu court, jeune homme !
On pouvait dire... oh ! Dieu ! ... bien des choses en somme...
En variant le ton, -par exemple, tenez :

Académique : Merci à Philippe Hoogvorst et à Guillaume Duc pour leur encadrement et les conseils prodigués. Un grand merci également aux membres du laboratoire ComElec de Télécom ParisTech et en particulier à Jean-Luc Danger et Sylvain Guilley pour avoir initier cette thèse et avoir su me donner l'envie de me plonger dans ce projet. À ce titre, Philippe Gaborit doit également être remercié.

Candidatorial[1] :* Merci à Assia Tria, Jean-Louis Lanet, Viktor Fischer, Renaud Pacalet, Pascal Urien, Éric Vétillard et Pierre Paradinas (même si d'obscurs raisons administratives ;) et un calendrier chargé ont empêchés ce dernier d'être officiellement membre du jury de ma soutenance de thèse) pour avoir accepter de s'investir dans l'évaluation des travaux qui dans le cadre de cette thèse ont été réalisés.

Corporate : Merci à Oberthur Technologies de m'avoir donné l'opportunité de donner un fort caractère industriel à mes travaux de recherche.

Collèg(u)ial :* Merci à mes collègues pessacais pour m'avoir accueilli parmi eux en dépit de mes convictions footballistiques (heureusement je n'étais pas seul !). Un grand merci à ceux qui ont été impliqué (parfois malgré eux `:p`) dans mes travaux : Philippe Andouard, Stéphane Arzur, Matthieu Boisdé, Nicolas Bousquet, Olivier Chamley, Marc Dubuisson, Hugo Grenèche, Nicolas Morin et Florent Oulières. Merci à mes *coéquipiers* (actuels et ancien) n'étant pas dans la liste précédente, par ordre chronologique : Nicolas Vasseur, Soline Renner et Alberto Battistello. Merci également à ceux qui sont jusqu'ici passés entre les mailles du filet (mais je n'abdique pas !) : Stéphane Andrau, Angie Coppé, Patrick Davieaud, Yannick Manot, Stéphanie Souchard, Jérémy Billaud, Gabriel Gomez, Benoît Linxe, Soazic Landais et Samuel Duclou. Et enfin un merci tout particulier à ceux sans qui cette thèse n'aurait sans doute pas la même valeur : Vincent Guerin,

[1] Les adjectifs marqués d'un astérisque sont bel et bien des néologismes

Christophe Giraud et Hugues Thiebeauld.

Chirurgical : Merci aux services orthopédiques de la Clinique du Sport et de l'HIA Robert Picqué pour m'avoir donné le temps de prendre un peu de recul sur mes travaux, et accessoirement de m'avoir remis sur pied une fois chacun... Le sport, c'est vraiment dangereux pour la santé ! ;-)

Républicain : Merci à la République Française d'avoir financé cette thèse par le biais de l'Agence Nationale de la Recherche Technique, et de m'avoir permis d'effectuer jusqu'à ce stade des études (quasiment) gratuites, laïques et (pas toujours) obligatoires.

Doctorant : Un grand merci à mes collègues *thésards* de Darreau pour m'avoir toujours trouvé une place dans leur bureau, pour les discussions (scientifiques) et les pauses sur la terrasse.

Éditorial : Merci (encore) à mes co-auteurs : Hugues Thiebeauld, Vincent Guerin, Philippe Hoogvorst, Guillaume Duc, Christophe Giraud et Philippe Andouard. Il est vrai qu'il est parfois difficile de savoir si une virgule est nécessaire, ou pas... si un mot vaut véritablement mieux qu'un autre, ou pas... si l'on satisfera la limitation du nombre de pages sans toucher aux marges, ou pas...

Démagogique : Merci au lecteur qui par essence justifie la rédaction de ce document.

Amical : Cette thèse marque également la fin de mes années d'étudiants. Je tiens donc également à remercier ceux qui m'ont accompagnés en journée et en soirée durant ces années, les "Parisiens" (c'est comme a qu'on vous appelle dans le reste du monde... enfin de la France), les "Limougeauds" (même si les vrais limousins se comptaient sur les doigts d'une seule main), les "Bordelais" (enfin le SAM Basket quoi !).

Familial : Je ne saurais les remercier uniquement pour ça, mais mille mercis à mes parents pour m'avoir toujours soutenu et m'avoir permis de poursuivre mes études jusqu'à ce stade avancé. J'en profite aussi pour remercier ma p'tite sis' d'être... bah ma p'tite sis' quoi ... et dire que j'aurai pu te remercier pour avoir relu et corrigé ce manuscrit (qui n'en est pas un) ! :p Et plus généralement un grand merci à toute la famiglia !

Personnel : Enfin, le meilleur pour la fin, un énorme merci à ma petite Carole pour tout ce qu'elle m'apporte et pour avoir subi quotidiennement les aléas de la (de ma) recherche : les relectures et deadlines des articles et de ce manuscrit, les répétitions des présentations, les déplacements aux conférences.

-Voilà ce qu'à peu près, mon cher, vous m'auriez dit
Si vous aviez un peu de lettres et d'esprit [...]

Résumé

Le sujet de ce manuscrit est l'étude des attaques matérielles, logicielles et combinées contre les plateformes Java Card et la sécurisation de ces plateformes face à de telles attaques.

Contexte et État de l'Art

La Partie I de ce manuscrit établit le contexte général des travaux réalisés dans le cadre de cette thèse et décrit brièvement l'état de l'art de la sécurité des cartes à puce et, de manière plus complète, celui de la sécurité des plateformes Java Card.

Introduction aux Cartes à Puce

Le Chapitre 1 introduit la notion de carte à puce. Les cartes à puce jouent aujourd'hui un rôle crucial dans de nombreuses applications que tout un chacun utilise de façon quotidienne. Ceci tient au fait que les cartes à puce se sont imposées comme le moyen privilégié pour assurer la confidentialité et l'intégrité de données personnelles et/ou secrètes par des moyens cryptographiques.

L'évolution des différents champs d'applications tels que le paiement par carte bancaire, la communication par téléphonie mobile, la validation de titre de transport, *etc*, a généré un déploiement massif de ces cartes. Un autre facteur pour la forte croissance de cette technologie a été l'important effort de standardisation mondiale démarré dès 1987 et qui permet aujourd'hui d'utiliser une même carte dans tous les pays du monde.

D'autre part, si certaines caractéristiques physiques des cartes à puce suivent également des normes très strictes, d'autres en revanche varient d'une carte à une autre. En particulier, les différents composants électroniques constituant la puce évoluent avec les avancées technologiques et varient selon les fabricants et selon les cibles applicatives.

Attaques Matérielles sur les Systèmes Embarqués

De part le caractère sensible des informations qu'elles contiennent, les cartes à puces sont la cible de nombreuses attaques. Les attaques dîtes matérielles en particulier représentent une menace sérieuse et sont introduites dans le Chapitre 2. Cette famille d'attaques regroupe elle-même deux types d'attaques : les attaques par analyse des

canaux auxiliaires, ou cachés (SCA, de l'anglais *Side-Channel Analysis*), et les attaques par analyse de fautes (FA, de l'anglais *Fault Analysis*).

Les attaques SCA visent à gagner de l'information sur les données manipulées par le composant (typiquement des clefs de chiffrement) par le biais d'une fuite d'information, provenant par exemple du temps que met le composant à effectuer une opération, de sa consommation électrique ou de son rayonnement électromagnétique durant cette opération.

Les attaques FA reposent, elles, sur la capacité de l'attaquant à perturber physiquement le composant durant une opération. Cette perturbation peut être réalisée de différentes manières (pic de tension électrique, de champ électromagnétique, de rayonnement optique) et peut avoir différentes conséquences selon qu'elle perturbe le flot d'éxécution d'une application ou bien la/les valeur(s) manipulée(s) par le composant.

Ces deux types d'attaques ont été, et sont encore, majoritairement étudiés contre les algorithmes cryptographiques embarqués.

La Technologie Java Card

Le Chapitre 3 introduit la technologie Java Card qui est au centre des travaux décrits dans ce manuscrit. Cette dernière a été introduite en 1996 et s'est rapidement imposée dans l'industrie des cartes à puce.

Ce vif succès est principalement dû à la réduction du coût du développement et du déploiement d'applications pour cartes à puce rendue possible par cette technologie. En effet, elle offre une couche d'abstraction (la machine virtuelle Java Card) permettant à une seule et même application Java Card d'être exécutée sur n'importe quelle carte à puce implémentant les spécifications Java Card, indépendamment des spécificités matérielles de la puce et de son jeu d'instructions natif.

Une application Java Card est écrite dans un sous-ensemble du langage Java et est compilée, puis éventuellement convertie, en un fichier binaire dont le format est standardisé et contenant des instructions compréhensibles par toute machine virtuelle Java Card, appelées *bytecodes*.

On trouve également autour de cette technologie un écosystème permettant d'intégrer facilement les plateformes Java Cards dans les différents champs d'applications des cartes à puce. De plus, la plateforme Java Card est capable d'accueillir plusieurs applications (on parle de plateforme multi-applicative) et autorise le chargement d'application *post-issuance* (i.e. après que la carte ait été délivrée à l'utilisateur).

Enfin, les spécifications Java Card définissent plusieurs mécanismes permettant d'assurer la sécurité de la plateforme elle-même ainsi que des applications qu'elle embarque.

État de l'Art des Attaques contre des Java Cards

Étant la plateforme dominante dans le monde des cartes à puce, Java Card est, à l'instar des systèmes *Windows* dans le monde des ordinateurs personnels, une cible privilégiée pour les attaquants. Le Chapitre 4 montre comment la communauté scientifique a essayé de contourner certains mécanismes sécuritaires des plateformes Java Card en tirant parti de la possibilité de charger et d'exécuter des applications potentiellement malicieuses. Nous verrons que de telles applications peuvent permettre à un attaquant d'obtenir de l'information sur la plateforme ou sur les autres applications embarquées grâce à des attaques logicielles et SCA.

Cependant, nous exposons par la suite que la majorité des attaques publiées tirent encore d'avantage parti de la facilité de déploiement d'applications et utilisent des applications dites mal-formées (i.e. ne respectant pas les règles du langage Java Card) afin de contourner certaines règles de sécurité.

De telles applications sont effectivement assez facilement produites par modification du fichier binaire résultant des étapes de compilation et de conversion : le fichier CAP. Le Listing 1 illustre avec quelle simplicité une application mal-formé peut être généré. En effet, la modification d'un seul octet dans le code binaire suffit à rendre l'application illégale au regard des règles du langage Java.

Listing 1: Création d'une application mal-formée

```
// Séquence d'instruction correspondant à l'affectation : obj1 = obj2
// obj1 et obj2 étant des instances d'objets Java.
aload_1  (0x19)
astore_2 (0x2D)

// Séquence modifiée permettant de fixer la référence (l'adresse)
// de l'instance d'objet obj2. (interdite dans le langage Java)
iload_3  (0x23)
astore_2 (0x2D)
```

Avec la généralisation de l'utilisation d'outils d'analyse statique de code permettant d'interdire le chargement de ces applications mal-formées (notamment le vérificateur de bytecode, ou *bytecode verifier*), de telles attaques deviennent moins réalistes.

Néanmoins, et comme nous le voyons tout au long de ce mémoire, l'introduction des attaques dites combinées (CA, de l'anglais *Combined Attack*) alliant des applications malicieuses et une/plusieurs injection(s) de fautes remettent en cause la sécurité apportée par ces outils et représente donc une nouvelle menace pour les plateformes Java Card, modifiant significativement le modèle d'attaquant.

Analyse Sécuritaire, la Recherche de Chemins d'Attaque

La Partie II de ce manuscrit décrit les résultats de l'analyse sécuritaire menée afin d'évaluer les conséquences d'attaques combinées.

La Propriété de Bon-Typage

Le Chapitre 5 présente les résultats de notre analyse sécuritaire vis à vis de la propriété de *bon-typage*, assurant qu'un objet va toujours être utilisé en accord avec son type (sa classe Java), et en particulier qu'un objet ne pourra être utilisé de la même manière qu'une valeur scalaire.

Cette propriété est la pierre angulaire de la sécurité des systèmes Java, en grande partie car elle interdit l'utilisation d'opération arithmétique sur les références des objets. Il est donc très envisageable qu'un attaquant tente de mettre cette propriété en défaut en provoquant une confusion de type.

Ce chapitre décrit deux méthodes permettant de provoquer une confusion de type et expose différentes exploitations d'une telle confusion de type sur des plateformes Java Card. Ces attaques nécessitent que l'attaquant ait l'opportunité de charger des applications sur la Java Card ciblée. Si le droit de charger des applications sur une carte n'est pas garanti, la sécurité de la plateforme doit néanmoins envisager un tel scénario, ce droit existant nécessairement.

Perturbation de l'instruction `checkcast`. La première méthode présentée est la première attaque publiée combinant une injection de faute et une application malicieuse. Cette attaque a été mise au point avec Hugues Thiebeauld et Vincent Guerin et publiée dans les actes de la conférence CARDIS 2010 [BTG10].

Ce travail présente en fait deux nouveautés :

- premièrement, une telle attaque combinée n'avait jamais été exploitée auparavant,
- secondement, cette attaque cible la dernière version des spécifications Java Card (Java Card 3 Édition Connectée), première version recquiérant la présence d'un outil de vérification des applications embarqué, et interdisant donc toute application mal-formée.

L'attaque exposée consiste à perturber l'exécution d'une instruction particulière de la plateforme Java Card lui permettant ainsi de s'assurer de la validité d'une opération de conversion de type : `checkcast`, générée par la compilation d'une ligne de code contenant l'opérateur de conversion de type : `()`.

Cette attaque implique l'utilisation des trois classes A, B et C définies dans le Listing 2 et repose sur leur représentation interne, ainsi qu'illustrée dans la Figure 1 pour les classes B et C.

Listing 2: Les classes impliquées

```
public class A {
  byte b00,...,bFF;
}

public class B {
  short addr;
}

public class C {
  A a;
}
```

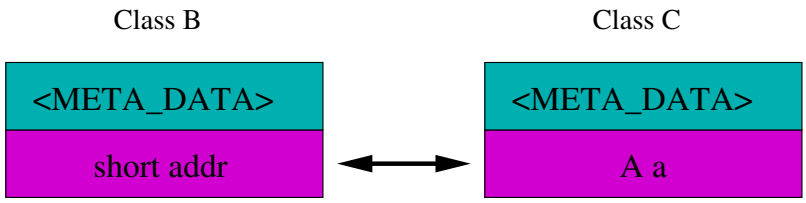


Figure 1: Représentation interne des instances de classes `B` et `C`

Le principe de l'attaque est donc de tirer parti de la structure interne des objets afin de forger la référence du champ d'instance `c.a`, comme illustré dans la Figure 2.

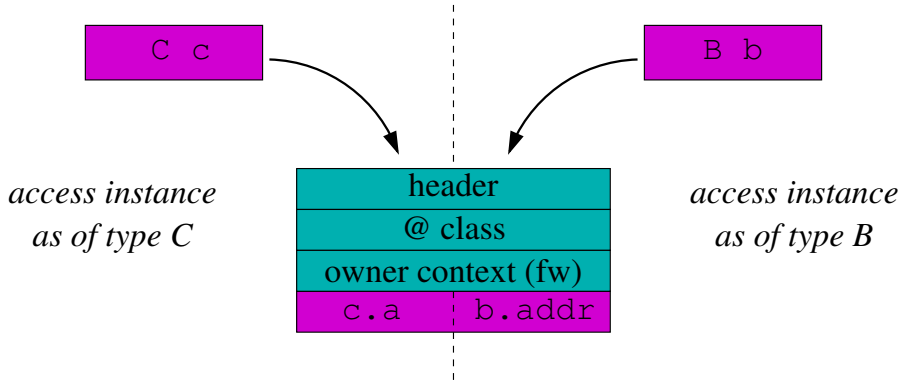


Figure 2: Accès au même objet en tant qu'instance de `B` ou de `C`

La partie active de l'attaque repose ensuite sur l'exécution de l'application décrite dans le Listing 3 et sur la perturbation de l'instruction `checkcast` correspondant à la conversion de type ligne 10.

Cette perturbation a été mise en pratique avec succès sur le banc d'attaque laser du laboratoire de sécurité d'Oberthur Technologies. Les Figures 3 et 4 présentent les mesures de consommation relatives Ã l'exécution de la méthode `process` du Listing 3

Listing 3: L'applet malicieuse

```
public class AttackExtApp extends Applet {
  B b; C c; boolean classFound;
  ... // Constructeur (initialisation), méthode install
  public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    ...
    switch (buffer[ISO7816.OFFSET_INS]) {
      case INS_ILLEGAL_CAST:
        try {
          c = (C) b;  // Opération ciblée par le laser
          return; // Succès
        } catch (ClassCastException e) {
          /*Echec*/
        }
      }
    }
  }
}
```

respectivement sans et avec perturbation du composant.

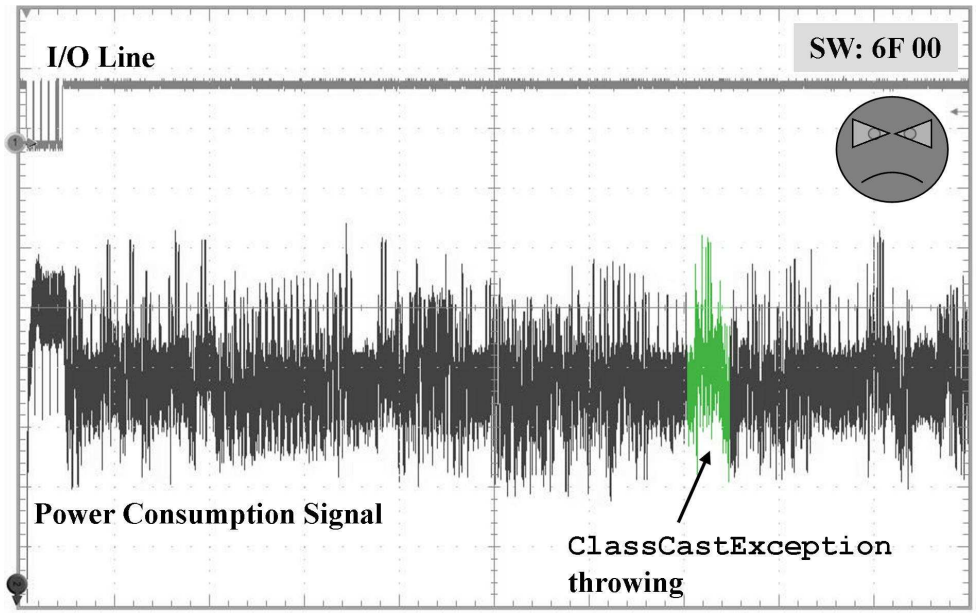


Figure 3: Exécution de l'instruction INS_ILLEGAL_CAST de l'applet

Perturbation de la pile d'opérandes. La seconde méthode exposée est une attaque visant un composant essentiel de la machine virtuelle Java Card, la pile d'opérandes. En effet, la majorité des instructions définies par la machine virtuelle Java Card consiste en un dépilement d'un certains nombre d'opérandes, l'exécution d'opérations sur ces opérandes et l'empilement du résultat de ces opérations. Cette attaque a été mise

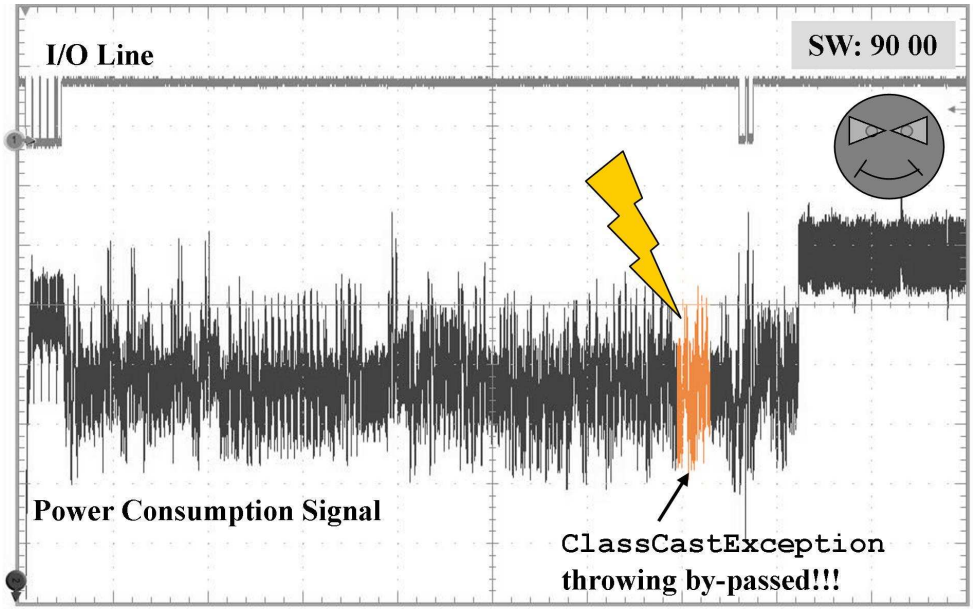


Figure 4: Exécution perturbée de l'instruction INS_ILLEGAL_CAST de l'applet

au point avec Guillaume Duc et Philippe Hoogvorst et publiée dans les actes de la conférence CARDIS 2011 [BDH11].

Il s'avère que la perturbation des opérandes empilées permet également de provoquer une confusion de type au sein d'une application malicieuse développée à cet effet. Dans le contexte de cette attaque, le modèle de faute considéré consiste en une perturbation de la mise à jour de la pile d'opérande lors de l'empilement du résultat d'une instruction de façcon à ce que la taille de la pile soit cohérente mais qu'une opérande ne soit pas mise à jour.

En considérant une application qui empile successivement deux objets de types différents `A` et `B`, une perturbation fidèle au modèle de faute défini provoque une confusion de types entre les types `A` et `B`. En outre, le concept de confusion d'instance est introduit.

Ce concept est similaire au concept de confusion de type, excepté le fait que l'on considère une confusion de deux instances d'un seul et même type, ou bien de deux instances de deux types implï¿½mentant une seule et même interface. Le Listing 4 décrit l'application utilisée pour réaliser une attaque menant à une telle confusion d'instance.

Cette attaque a été mise en pratique sur le banc d'attaque laser du laboratoire de sécurité d'Oberthur Technologies. Le modèle de faute défini a pu être validé, le taux de succès de l'attaque atteignant presque 10%. Les résultats des expériences menées sont donnés dans l'Appendice A.

Listing 4: L'applet malicieuse

```
public interface I {
    // Déclaration des méthodes de l'interface
}
public class A implements I {
    // Une implémentation des méthodes de l'interface
}
public class B implements I {
    // Une autre implémentation des méthodes de l'interface
}
public class Application {

    public I i;                         // Champs d'instance de type I
    public B b00, b01, ..., bFF;        // 256 objets de type B

    public void confusion() {

        b00 = new B();
        //...
        bFF = new (B);

        A a = getA();    // la méthode getA renvoyant une instance de A
                         // une perturbation de la pile a de grande chance
                         // de changer la référence de l'objet de type A
                         // retourné en une référence d'un des objets
                         // de type B.
    }
```

Exploitations de confusions de type. Enfin, différentes exploitations de confusions de type sont exposées.

La première de celles-ci consiste en la modification du code binaire d'une application Java Card embarquée en tirant parti des particularités de la classe `Class` définie par les spécifications Java Card. Ceci peut être utilisé de différentes manières par un attaquant. Premièrement, ce dernier peut à loisir modifier sa propre application et donc la transformer en une application mal-formée, redonnant alors vie aux différentes attaques présentées dans le Chapitre 4. Mais l'attaquant peut également modifier le code d'une autre application embarquée. L'exemple donné consiste à supprimer le code implémentant l'exécution d'une opération de vérification d'une signature électronique, et donc de gagner certains privilèges au sein de l'application cible.

La seconde exploitation tire parti de la représentation des chaînes de caractères sur la plateforme par la classe `String`. Cette exploitation montre qu'une confusion de type adéquate peut permettre à l'application de l'attaquant d'usurper l'identité d'une autre application sur la plateforme en modifiant le contenu de la chaîne de caractère représentant l'URL (Unified Resource Locator, *i.e.* l'adresse) d'une application sans en modifier la référence. Un utilisateur demandant l'accès à l'application légitime sera alors redirigé vers l'application de l'attaquant sans en être averti. Cette exploitation a été présentée lors de la conférence E-SMART'10 [Bar10].

Enfin la dernière exploitation applique le concept de confusion d'instance. L'exemple donné d'une telle confusion d'instance permet à un attaquant d'usurper l'identité d'un utilisateur autorisé en créant une confusion entre des objets de classes implémentant l'interface `Authenticator` permettant à un utilisateur de s'authentifier en fournissant un mot de passe par exemple. La confusion d'instance permet ici de forcer l'utilisation d'une instance de la classe implémentant `Authenticator` dont la méthode `check`, supposée vérifier la validité du mot de passe soumis, n'effectue aucun contrôle et valide donc l'authentification quelque soit le mot de passe soumis.

L'Intégrité du Flot d'Exécution

Il est assez facile d'imaginer que la perturbation du flot d'exécution d'une application peut avoir une grande influence sur son fonctionnement et sur sa sécurité. La modification du flot d'exécution peut donc être un objectif pour un attaquant. Le Chapitre 6 présente les moyens de perturber le flot d'exécution avec plus ou moins d'efficacité et les différentes conséquences que ces perturbations peuvent avoir.

Perturbation de la pile d'opérandes lors de branchements conditionnels. Ainsi que présenté dans le Chapitre 5, la première attaque concerne à nouveau une perturbation de la pile d'opérandes. En effet, une analyse de la spécification de différentes instructions de branchement conditionnel (nommément `if_eq`, `if_ne`) a permis d'exhiber une faiblesse au regard du modèle de faute précédemment défini, ainsi qu'au regard des modèles de fautes usuellement admis.

Cette faiblesse vient du fait que ces instructions effectuent des comparaisons de la valeur au sommet de la pile d'opérandes avec la valeur 0 pour décider si branchement sera pris ou non. Hors comme vu précédemment, un attaquant peut avoir la capacité de perturber les valeurs qui sont empilées, et donc de forcer un branchement quelque soit la condition qui lui est associée.

La validité de l'attaque a été vérifiée en la mettant en pratique sur le code défini dans le Listing 5.

Listing 5: L'application de test

```
// b = true
if (b) {
        Util.setShort(buffer, (short)0,(short)0x1111);
}
else {
        Util.setShort(buffer, (short)0,(short)0x2222);
}
Util.setShort(buffer, (short)2, proof);
```

L'opération ciblée par l'injection de faute est l'empilement de la valeur de la variable `b`

avant l'exécution de l'instruction `ifeq`. Le taux de succès obtenu est assez spectaculaire : 78,25%. Il apparaît donc tr'es clairement que ces instructions sont particulièrement sensibles aux attaques par injection de faute.
Une expérience similaire a été menée dans le cas de l'instruction `ifne` avec des résultats équivalents.

Corruption *multithreadé* du contexte d'exécution. La seconde attaque visant le flot d'exécution des applications utilise une des particularités de la dernière version des spécifications Java Card (Java Card 3.0 Édition Connectée) : le support du multithreading (i.e. le support de l'exécution en parallèle de plusieurs applications). Le concept de cette attaque ainsi que son implémentation ont été réalisée en collaboration avec Hugues Thiebeauld et la description de cette attaque a été publiée dans les actes de la conférence CARDIS 2011 [BT11].

Outre le multithreading, le scénario d'attaque imaginé utilise également les capacités de communication de la plateforme. En particulier, il repose sur la séparation de l'interface bas-niveau gérant la communication et l'implémentation de l'ordonnanceur de thread, assignant tour à tour les ressources de la carte aux différentes applications concurrentes.

Le scénario d'attaque peut être résumé par la Figure 5.

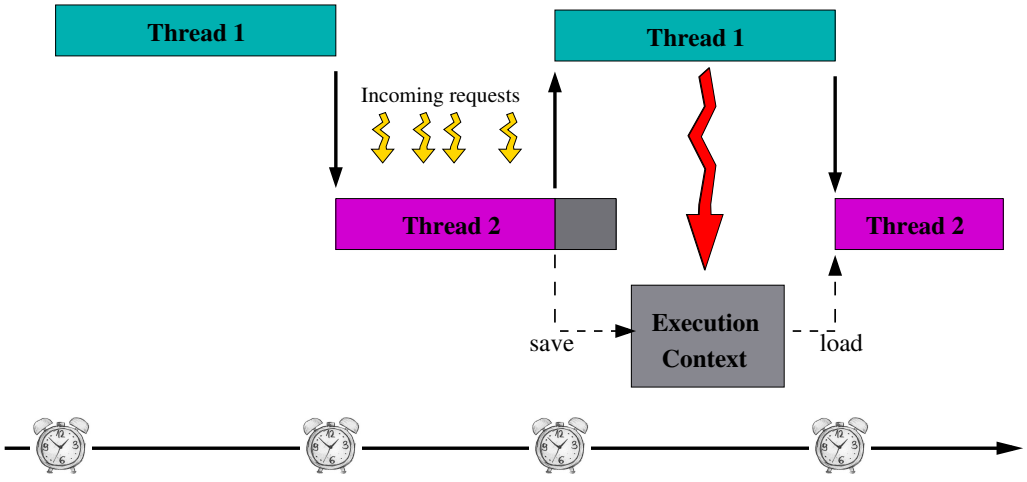


Figure 5: Le scénario d'attaque

D'après le scénario, l'attaquant peut parvenir à forcer le changement de thread par l'ordonnanceur à un instant t dans l'exécution du premier thread en envoyant de nombreuse requêtes sur la ligne de communication du composant.

Ceci a été validé en exécutant une application incrémentant un compteur pendant un laps de temps fixe, tout en envoyant de nombreuses requêtes de communication Ã la carte. Ainsi qu'escompté, la Figure 6 montre que la valeur atteinte par le compteur

décroit significativement avec l'augmentation du nombre de requêtes de communication envoyées.

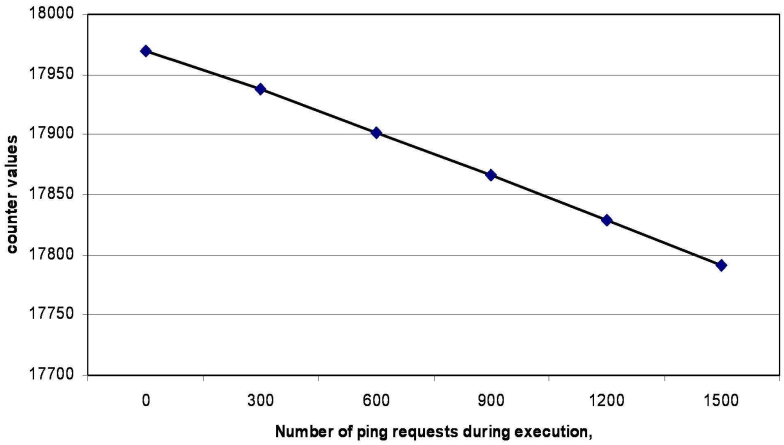


Figure 6: Influence de la communication sur le nombre d'instructions exécutées.

La seconde partie de l'attaque consiste à corrompre le contexte d'exécution du thread cible. Ceci peut être réalisé en modifiant le pointeur contenu dans un tableau de type `byte`, ainsi qu'illustré dans la Figure 7.

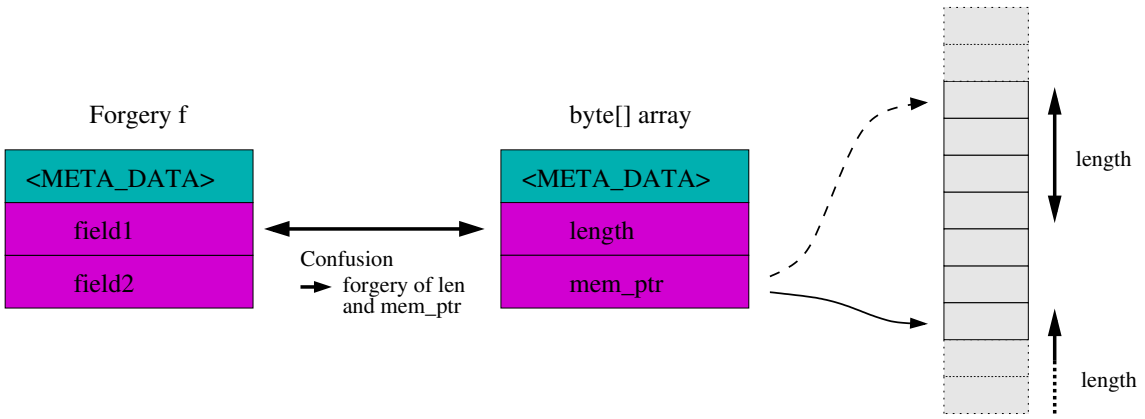


Figure 7: Confusion entre deux instances de classes pour forger l'adresse d'un tableau.

Ayant accès au contexte d'exécution de l'application ciblée, l'attaquant peut, à sa guise, modifier les valeurs des variables locales, les valeurs stockées dans la pile d'opérandes, voire le `java program counter` (pointeur vers la prochaine instruction à exécuter), et donc maîtriser le comportement de l'application.

Afin de démontrer la puissance de cette attaque, nous exposons comment elle peut mener à contourner des mécanismes de sécurité réputés robustes tels qu'une signature électronique.

Perturbation des mécanismes liés aux exceptions. Dans la dernière partie de ce chapitre, nous étudions les attaques contre les mécanismes liés aux exceptions. Ce travail a été mené avec Philippe Hoogvorst et Guillaume Duc et publié dans les actes de la conférence SECRYPT 2012 [BHD12b].

Le principe d'exception en Java permet d'associer à une erreur particulière, ou plus généralement à une situation particulière un traitement particulier (ou *handler*). La syntaxe de ces exceptions est donnée dans le Listing 6.

Listing 6: La syntaxe des exceptions

```
try {
      ...      // Code opérant une suite d'instructions susceptible
      ...      // de "jeter" différentes exceptions.
} catch (ExceptionType1 et1) {
      ...      // Une situation particulière a mené au jet d'une
      ...      // exception sp\'ecifique. Elle est "attrapée" ici.
} catch (ExceptionType2 et2) {
      ...      // Une situation particulière a mené au jet d'une
      ...      // exception spécifique. Elle est "attrapée" ici.
} finally {
      ...      // Code exécuté qu'une exception ait été jeté
      ...      // ou non, attrapée ou non.
}
```

Dans le format binaire des applications, les différents *handler* d'exceptions sont triés et listés dans une table pour chaque méthode définie. Ainsi lorsqu'une exception est jetée, la machine virtuelle a la responsabilité de rediriger le flot d'exécution vers le bon *handler* en cherchant dans cette table.

Notre étude a révélé que différentes perturbations durant les phases de jet d'exception et de recherche du handler approprié pouvaient permettre de corrompre le flot d'exécution de l'application.

En effet, dans la phase du jet d'exception, une perturbation de la référence de l'exception pourrait mener à rediriger le flot d'exécution dans le mauvais *handler*, créant par la même occasion une confusion de type entre le type de l'exception jetée et le type d'exception attendue par le *handler*.

De plus, un autre type d'erreur peut simplement consister à inhiber une exception en l'empêchant d'être jetée.

De manière similaire, dans la phase de recherche du handler approprié, une injection de faute affectant l'algorithme de recherche peut rediriger le flot d'exécution vers le mauvais *handler*.

L'Isolation Inter-Applications

Dans toute plateforme multi-applicative, il est crucial d'assurer une stricte isolation entre les différentes applications tant au niveau de leur code, que des données qu'elles manipulent. Le Chapitre 7 présente différentes attaques contre les mécanismes d'isolations fournis par la plateforme Java Card.

L'isolation du code et le chargeur de classe. La première attaque décrite dans ce chapitre remet en question le mécanisme d'isolation de code présent dans la derière version des spécifications Java Card par le biais de la notion d'objets partagés (SIOs, de l'anglais *Shareable Interface Objects*). Cette attaque a été présentée lors de la conférence E-SMART'09 [Bar09].

Le principe du partage d'objet repose sur l'exposition d'une interface partagée qui déclare les méthodes qui sont accessibles aux autres applications, i.e. aux clients. De son côté le serveur contient une classe implémentant cette interface partagée. La plupart du temps, en plus des méthodes définies dans l'interface partagée, le serveur définit des méthodes privées qui ne sont elles pas partagées.

L'attaque décrite vise à accéder aux méthodes privées du serveur, depuis une application cliente. Elle repose en premier lieu sur un confusion de type entre deux classes implémentant un même interface partagée. Cette confusion de type peut être provoquée par une des deux méthodes déjà introduites dans le Chapitre 5.

Grâce à cette confusion, l'attaquant gagne la visibilité des méthodes privées du serveur. L'isolation du code est alors mise en défaut, l'attaquant peut appeler la méthode non-partagée. Néanmoins, afin de parvenir à exécuter cette méthode, l'attaquant devra également contourner le mécanisme d'isolation des données, ou pare-feu applicatif.

Contournement du pare-feu applicatif par rejeu applicatif. Comme vu précédemment, le pare-feu applicatif est un mécanisme important de la sécurité des plateformes Java Card. La deuxième attaque présentée ici repose sur une nouveauté introduite avec la dernière version des spécifications Java Card : le mécanisme de ramasse-miettes (ou *garbage collector*). Cette attaque a été mise au point avec Philippe Hoogvorst et Guillaume Duc et a été publiée dans les actes de la conférence ESSOS 2012 [BHD12a].

Cette attaque repose sur une allocation linéaire des références d'objets telle que si les références allouées à un instant t : $(r_i)_{1 \leq i \leq n}$ sont telles que $1 \leq r_i \leq k$, la prochaine référence allouée sera r_{n+1} définie par :

$$r_{n+1} = \min\{r_i \text{ t.q. } \forall\, r_j < r_i,\, r_j \text{ est allouée}\} \quad (1)$$

En d'autres termes, la prochaine référence allouée sera la première référence disponible. Cette hypothèse a pu être testée avec succès sur différentes implémentations des spécifications Java Card.

Partant de cette hypothèse et en tirant parti de l'introduction du mécanisme de ramasse-miette, un attaquant est alors capable de prédire les références qui vont être utilisées. Cependant cette nouvelle capacité n'est pas l'unique responsable de l'attaque décrite ici. En effet, nous avons noté une différence importante entre les spécifications Java Card 3.0 Édition Connectée et ses prédecesseures concernant le mécanisme de suppression d'une application.

Là où les précédentes versions des spécifications recquiéraient que tous les objets d'une application soient effacés lors de la suppression de cette application, la dernière version est moins exigeante et ne requiert seulement que ces objets soient inaccessibles, i.e. susceptibles d'être effacés par le mécanisme de ramasse-miettes.

Ainsi, en combinant les capacités de prédire et de forger les références d'objets, un attaquant est capable de supprimer une application et d'empêcher la suppression de ces objets par la prochaine exécution du mécanisme de ramasse-miettes.

Sous l'hypothèse que le pare-feu applicatif fonctionne en assignant un identifiant à chaque application, on peut considérer qu'un attaquant aura la possibilité d'installer une application ayant le même identifiant que l'application supprimée dont les objets ont été conservés. L'attaquant pourra alors accéder légitimement à ces objets, contournant ainsi le pare-feu applicatif.

Utilisation des tableaux globaux. Les tableaux globaux sont des objets particuliers dans la plateforme Java Card qui ne sont pas soumis au mécanisme d'isolation inter-applicatif. Dans la dernière partie de ce chapitre nous exposons comment ce type de tableau peut être utilisé illicitement grâce à une injection de faute. Cette attaque permet en particulier de mettre en échec un protocole de communication sécurisé utilisé dans de nombreuses applications. Elle a été imaginée avec Christophe Giraud et Vincent Guerin et a été publiée dans les actes de la conférence SEC 2012 [BGG12].

L'injection de faute est utilisée dans cette attaque afin de permettre de contourner une restriction imposée par la plateforme qui empêche une application de stocker une référence vers un tableau global. Si réussie, cette injection de faute pourra donc permettre à l'attaquant de stocker la référence d'un tableau global comme le buffer de communication de la carte : le buffer APDU (de l'anglais *Application Protocol Data Unit*).

La suite de cette attaque montre comment l'utilisation de certains mécanismes des environnements permettant la gestion de la carte (GlobalPlatform) ou son intégration dans la téléphonie mobile (USIM Toolkit) peuvent permettre de tirer parti de cette situation.

En outre, une exploitation de cette attaque sur une plateforme supportant le multi-threading montre qu'un attaquant *embarqué* pourrait contourner la sécurité apportée par un canal sécurisé (*Secure Channel*) permettant d'assurer la confidentialité ainsi que l'intégrité des communications entre deux applications, l'une résidant sur la carte, l'autre sur un terminal.

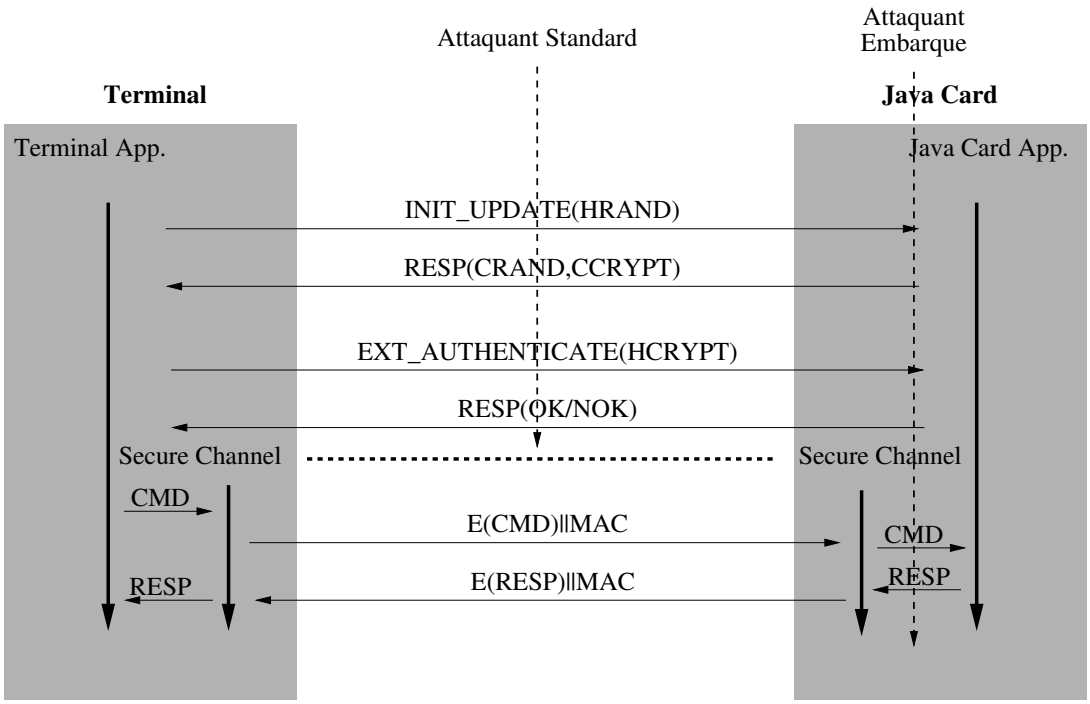


Figure 8: Espionnage et corruption de la communication par un attaquant embarqué.

Cette exploitation est illustrée par la Figure 8.

Contributions à la Sécurité des Java Cards

La Partie III de ce manuscrit a démontré que des attaques combinant perturbations physiques et logiciels malicieux représentent une menace pour la sécurité des plateformes Java Card. La 3ème partie décrit les différentes contributions de cette thèse à la sécurisation des plateformes Java Card.

Machine Virtuelle Offensive, Défensive et Défendante...

Dans un premier temps, le Chapitre 8 fait le point sur les notions de machines virtuelles dites offensives, défensives et défendantes.

Dans les machines offensives et défensives, la sécurité du système repose sur un outil d'analyse statique du code chargé dans la carte appelé *vérificateur de code* (*bytecode verifier*). Cet outil assure la validité du code chargé par rapport aux règles du langage. En particulier, un *bytecode verifier* parfait interdit le chargement de toute application mal-formée, contrecarrant implicitement toute attaque basée sur une telle application.

Cependant, nous avons vu qu'une perturbation physique du composant durant l'exécution d'une application bien formée peut permettre de la transformer en une application mal-formée (au moins temporairement). La protection apportée par le *bytecode verifier* dans ce cas n'est donc pas adéquate. D'autres outils, permettant par exemple de détecter une application "transformable" en application mal-formée, peuvent alors être envisagés. D'un autre côté, les applications légitimes doivent, dans le cadre d'une machine virtuelle défensive ou offensive, assurer elles-mêmes leur sécurité face à des attaques par injection de faute.

À l'opposé de ces machines virtuelles offensives et défensives, on trouve les machines virtuelles dites défendantes (de l'anglais *defending*). De telles machines virtuelles assurent dynamiquement la sécurité de la plateforme par des vérifications exécutées par la plateforme durant l'exécution d'une application. Ces vérifications sont, pour la majorité, redondantes avec les contrôles déjà effectués par le *bytecode verifier* et impliquent évidemment un coût additionnel en terme de temps de calcul et d'espace mémoire utilisé. Néanmoins, de telles machines virtuelles s'avèrent plus à même de détecter une attaque par injection de faute.

Une question qui se pose et à laquelle nous apportons quelques éléments de réponse est de savoir à quel moment il est judicieux d'exécuter ces vérifications supplémentaires afin de dégrader au minimum les performance de la plateforme. Il apparaît comme une solution avantageuse d'adjoindre ces vérifications à celle du pare-feu inter-applicatif, ainsi qu'aux instructions de rupture du flot de contrôle.

Assurer la Propriété de Bon-Typage

Le Chapitre 5 a exposé plusieurs attaques permettant de mettre en défaut la sécurité du typage au sein de la plateforme. Le Chapitre 9 expose, en réponse à une de ces attaques, différentes méthodes permettant de vérifier l'intégrité des valeurs empilées sur la pile d'opérandes Java.

Ces méthodes, publiées dans les actes de la conférence CARDIS 2011 [BDH11] sont :

- Une méthode basée sur un contrôle redondant simple des valeurs empilées,
- Une méthode utilisant les contrôles effectués par le mécanisme d'isolation applicative pour propager les éventuelles erreurs,
- Une méthode tirant partie d'une propriété invariante, au coût d'un octet de stockage supplémentaire.

Ces méthodes s'appliquent dans le contexte d'une machine virtuelle *défendante*, les contrôles d'intégrité étant implémentés au sein même de la machine virtuelle. Le coût en terme de temps d'exécution des différentes méthodes est décrit dans la Table 1.

Instructions	Simple	Propagation	Invariant
`aload+astore`	39.09 %	21.98 %	12.29 %
`aload+getfield+astore`	19.83 %	12.39 %	11.75 %
`aload+aload+putfield`	27.93 %	18.77 %	17.59 %
`aload+invokevirtual+return`	7.53 %	1.69 %	1.77 %
`aload+invokevirtual+areturn+astore`	8.82 %	3.26 %	2.38 %
`aload+putstatic`	18.60 %	11.58 %	8.89 %
`getstatic+astore`	19.18 %	10.76 %	10.21 %

Table 1: Impact des différentes contremesures sur le temps d'exécution de séquences de bytecodes (% d'augmentation par rapport à une implémentation initiale sans contremesure).

Étant données les limitations énoncées concernant la méthode par propagation, la méthode par invariant apparaît nettement la meilleure.

Dans la suite de ce chapitre, différentes approches permettant d'assurer la propriété de bon-typage dans un environnement perturbé sont discutées.

Assurer l'Intégrité du Flot d'Exécution

Augmenter la sécurité des instructions conditionnelles. Le Chapitre 10 décrit dans un premier temps une méthode permettant d'assurer la sécurité d'une application face à ce type d'attaque. Cette méthode mise au point en collaboration avec Christophe Giraud a fait l'objet d'un dépôt de brevet en France [BG11].

Si cette méthode est applicable pour n'importe quel type de machine virtuelle, elle est davantage adaptée au contexte des machines virtuelles offensives ou défensives. Ceci est dû au fait qu'elle repose sur un traitement statique de l'application en amont de son chargement. Le principe de cette méthode est d'ajouter automatiquement des instructions redondantes au niveau du code Java compilé. Le choix d'intervenir au niveau du code Java compilé est motivé par le fait que les informations permettant d'identifier les variables de type booléen sont encore présentes à ce stade, alors qu'elles n'apparaissent plus dans le fichier binaire chargé sur la carte.

La Figure 9 illustre les différentes étapes de la méthode proposée.

La méthode proposée apporte une meilleure sécurité que l'ajout d'instructions redondantes au niveau du code source de l'application généralement utilisé dans l'industrie. En effet, elle permet de se prémunir contre des attaques par simple ou double injections de fautes, à la fois dans le modèle de faute d'une modification de données empilées et dans le modèle de faute du saut d'instruction. Ceci est résumé dans la Table 2.

Elle permet en outre de préserver d'avantage les performances de l'application ainsi que son empreinte mémoire. Enfin cette méthode présente l'avantage d'être utilisable

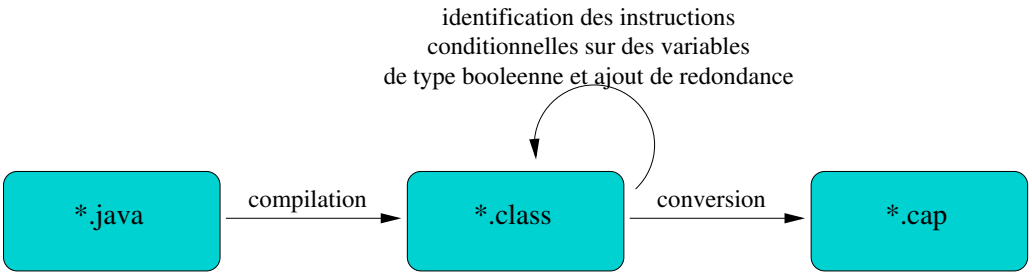


Figure 9: Exécution de la méthode d'ajout de redondance sur les instructions de branchement conditionnel opérant sur des booléens.

	Corruption de données		Saut d'instruction	
	Faute simple	Faute double	Faute simple	Faute double
Doublement à l'identique	Oui	Non	Oui	Non
Doublement à l'opposé	Oui	Non	Non	Non
Méthode proposée	Oui	Oui	Oui	Oui

Table 2: Sécurité apportée par la méthode proposée.

sur n'importe quelle machine virtuelle Java Card étant donné qu'elle n'utilise que des instructions définies par les spécifications.

Modification aléatoire du jeu d'instructions standard. Dans un second temps, une contremesure à l'attaque de Guillaume Bouffard *et al.* basée sur l'interprétation de code introduit dans la carte à l'aide d'un cheval de Troie est proposée. Cette contremesure a été développée avec Philippe Andouard et a été acceptée pour présentation lors de la conférence CHIP-TO-CLOUD 2012 [BA12].

À l'origine de cette contremesure se trouve l'observation que la disponibilité publique du jeu d'instruction supporté par la plateforme est un élément fondamental des attaques par "injection" de code. Mais la connaissance du jeu d'instruction ne peut pas être remise en question elle-même, sans quoi aucune application ne pourrait être développée.

La méthode proposée repose donc sur la génération d'une permutation aléatoire du jeu d'instruction de la machine virtuelle unique à chaque application chargée sur la plateforme. La répercussion de cette permutation doit donc nécessairement être appliquée au code de l'application qu'elle concerne.

Cette étape de "traduction" de l'application doit donc être effectuée au chargement de l'application. On peut noter que dans le cadre d'une machine virtuelle offensive, cette traduction peut être effectuée à moindre coût pendant la phase de vérification de l'application.

La Figure 10 illustre les différentes étapes de la méthode proposée.

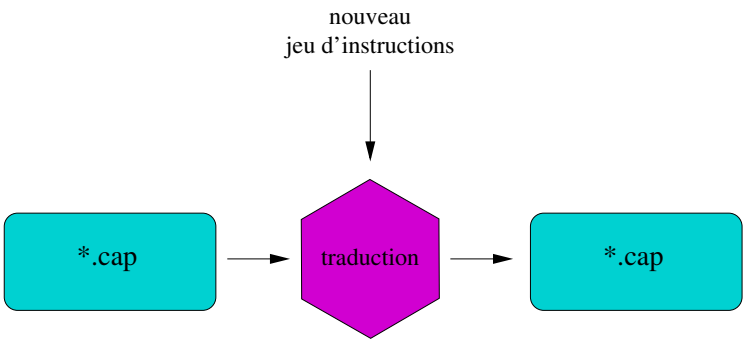


Figure 10: Exécution de la méthode de modification aléatoire du jeu d'instructions.

Il est également important de noter que si le coût de cette traduction n'est pas négligeable, il n'intervient qu'une seule fois dans le cycle de vie d'une application, lors de son chargement sur la carte. L'interprétation du code "traduit" peut se faire en revanche à un coût négligeable.

Conclusions et Perspectives

Depuis son introduction en 1996, la technologie Java Card a su s'imposer comme la technologie dominante dans le monde des cartes à puce. La même année que celle de l'invention de la technologie Java Card, Paul Kocher publia l'article qui mit en lumière l'importance des canaux auxiliaires. Peu après, les attaques par injection de faute furent introduites. Cependant ces deux types d'attaques n'ont longtemps été étudiés que vis-à-vis des implémentations embarquées d'algorithmes cryptographiques.

D'un autre côté, la facilité de développement et de déploiement d'applications offerte par la technologie Java Card a poussé l'étude de la sécurité des plateformes Java Card contre des attaques dites logicielles. Cependant, comme tout système embarqué, les plateformes Java Card ne sont pas intrinsèquement sûres face aux attaques par canaux cachés et injection de faute, dites matérielles. À l'inverse, un attaquant pourrait même parvenir à allier ces deux types d'attaques pour construire des chemins d'attaque plus puissants encore. Nous avons appelé ce type d'attaques des attaques combinées.

Ce manuscrit est dédié à l'évaluation et à l'amélioration de la sécurité des plateformes Java Card face aux attaques par injection de faute et plus particulièrement face aux attaques combinées.

L'analyse sécuritaire menée dans un premier temps nous a permis de conclure que plusieurs mécanismes sécuritaires des plateformes Java Card peuvent être contournés par une combinaison adéquate d'attaques logicielles et matérielles. En particulier, nous avons publié la première mise en pratique d'une telle attaque combinée contre une plateforme Java Card 3 Édition Connectée.

Au vu des résultats de notre analyse, nous avons proposé différentes contremesures contre certaines des attaques identifiées dans la seconde partie de ce manuscrit, ainsi que contre des attaques présentes dans la littérature.

Afin de contrecarrer des attaques combinées sur une plateforme Java Card, deux types d'approches peuvent être envisagées. D'un côté, les machines virtuelles offensives et défensives n'acceptent que des applications ayant passées avec succès différents contrôles effectués par des outils d'analyse statique, prouvant ainsi qu'elles assurent elles-mêmes une grande partie de la sécurité. De l'autre côté, les machines virtuelles dites défendantes acceptent *a priori* n'importe quelle application respectant les règles du langage Java Card et défendent ces applications contre certaines attaques.

Si la première approche peut s'avérer plus efficace si l'on considère l'exécution d'application de confiance dans un envionnement sécurisée, elle requiert dans la plupart des cas une somme considérable de vérifications du côté de l'application.

L'approche d'une machine virtuelle défendante nous apparaît donc plus appropriée au contexte des attaques combinées, ou simplement par injection de faute. Néanmoins, une telle machine virtuelle se doit de se tenir à l'état de l'art des attaques à la fois logicielles et matérielles.

L'introduction des attaques par injection de faute dans le contexte des Java Card est une évolution importante. Comme nous l'avons vu, elle remet en question une grande partie des mécanismes sécuritaires généralement utilisés. De plus, l'amélioration constante des techniques d'injection de faute, avec notamment la généralisation des attaques multiples ou bien l'émergence de nouveaux moyens, tel que les perturbations par champ électromagnétique, pourraient rendre de simples contrôles redondants obsolètes.

Jusqu'ici, les Java Cards, en dépit de leur capacité, sont restées des plateformes relativement fermées. Une évolution dans le sens des plateformes ouvertes pourrait également se révéler comme un défi important pour la communauté scientifique. De même, l'intégration des Java Cards dans l'écosystème des réseaux informatiques actuels est également un sujet d'étude important du point de vue de la sécurité.

Enfin, il est très probable que des attaques similaires à celles que nous avons décrites dans ce manuscrit puissent être portées sur d'autres plateformes Java, ou même sur d'autres machines virtuelles embarquées, notamment sur des téléphones mobiles.

Abstract

Smart cards play a key role in various applications we use on a daily basis: payment, mobile communication, public transports, *etc*. In this context, the Java Card technology has evolved since its introduction in the mid-nineties to become nowadays the world leading smart card platform. In the context of Java Card, researches on security have revealed that the possibility of loading malicious applications represents a real threat. In the meantime, the scientific community has also paid interest to the security of embedded cryptography, revealing that theoretically strong cryptosystems can be easily broken if their implementation does not take into account certain physical properties of the underlying hardware device. In particular, a part of the published attacks relies on the attacker's capacity to physically perturb the component during a cryptographic operation.

These latter fault attacks have been rarely considered in the literature in the Java Card context. In this thesis, we study and evaluate the security of Java Cards against the combination of fault and software attacks in order to enhance it.

First, we present various attack paths involving both hardware and software attacks and expose how these attacks allow to break various security mechanisms of Java Cards. In particular, our security analysis proves that the type-safety property, the control-flow integrity and the application isolation can be tampered with by the combination of adequate fault injections and malicious applications.

Then, with regards to the goal of this thesis and the results of our security analysis, we present different approaches allowing to improve the resistance of Java Cards against combined attacks. Thus we define several countermeasures against the attack we exposed as well as against some of the state-of-the-art attacks, always bearing in mind the strong constraints relative to smart cards.

Forewords

Smart cards are devoted to play a key role in the security of numerous protocols used throughout the world

The Java Card[2] is a particular kind of smart card which embeds a reduced Java Runtime Environment allowing it to execute applications written in a subset of the Java programming language. With about 6 billion cards delivered worldwide, Java Card is currently the most widespread smart card technology.

The security of Java Cards has been mainly threatened in the literature by malicious application attempting to exploit residual weaknesses (bugs) of a platform or to logically circumvent certain mechanisms it implements. However it is known for more than a decade that smart cards, and more generally embedded systems, are subject to particular types of attacks based on physical properties of the underlying hardware, namely Side-Channel Analysis and Fault Injection. The power of these attacks has been proven in many occasions, for instance by breaking theoretically strong cryptosystems, so to speak betrayed by their implementations.

This dissertation relates the research undertaken within the *Virtual Machine & Global Platform* and *Security* groups of Oberthur Technologies and the COMELEC department of Télécom ParisTech. The aim of this research has been to evaluate the potential consequences of hardware attacks against Java Card platforms by designing and analysing new attack paths involving physical perturbations of the device but also, and most importantly, to improve the robustness of Java Card platforms by proposing new countermeasures against such attacks.

In accordance with the fixed objectives, the contribution of this dissertation is twofold. On one hand we introduce novel attacks against Java Card platforms combining the assets of physical and logical attacks and which have been put into practice in Oberthur Technologies' security laboratory. On the other hand several methods and implementation techniques have been elaborated to counteract these attacks and enhance the security of Java Card platforms.

The first part of this dissertation sets the context of our research and introduces state-of-the-art attacks against smart cards and more particularly Java Card platforms. The second part presents the results of the security analysis led in the first step of our study with

[2]Java and Java Card are registered trademarks of Oracle Inc. in the United States of America and other countries.

regards to three particular notions which are the type safety, the execution flow integrity and the application isolation. Finally, the third part describes different countermeasures in response to our security analysis as well as to other attacks published in the literature and discusses more generic approaches to ensure the security of the platform.

Contents

Forewords **xxvii**

I Context and State of the Art **1**

Introduction to Part I **3**

1 Smart Cards in a Nutshell **5**
1.1 A Disputed Invention 5
1.2 From a Public Phone Card to a Digital Safe 5
1.3 The Architecture of a Smart Card 9

2 Hardware-Related Attacks **15**
2.1 Side Channel Analysis 16
2.2 Fault Attacks on Embedded Systems 19

3 The Java Card Technology and its Ecosystem **25**
3.1 Java-enabled Smart Cards 25
3.2 The Security of Java Card Platforms 35
3.3 The Ecosystem of Java Cards 45

4 State of the Art of Attacks against Java Cards **53**
4.1 The Open Platform Assumption 53
4.2 Malicious Application based Attacks 54
4.3 Ill-formed Application based Attacks 60
4.4 The Development of Combined Attacks and Mutant Applications 66

Conclusion to Part I **75**

II Analysis of the Security of Java Cards against Combined Attacks **77**

Introduction to Part II **79**

5 Security Analysis of the Type Safety Property **81**
5.1 Disruption of the `checkcast` Instruction 82
5.2 Disruption of the Operand Stack 87

5.3 Exploitations of the Type Confusions 92

6 Security Analysis of the Execution Flow Integrity 101
6.1 Disruption of the Operand Stack on Conditional Branching Instruction . . . 102
6.2 Multithreaded Corruption of the Execution Context 104
6.3 Disruption of `Exception`-Related Mechanisms 115

7 Security Analysis of the Application Isolation Mechanism 127
7.1 Questioning the Code Isolation by Abusing Class Loaders 128
7.2 Circumventing the Application Firewall with Application Replay 131
7.3 Abuse of Global Arrays 138

Conclusion to Part II 147

III Contributions to Enhance the Security of Java Cards 149

Introduction to Part III 151

8 Offensive, Defensive and Defending Virtual Machines 153
8.1 The Offensive and Defensive Virtual Machines 153
8.2 The Defending Virtual Machine 154
8.3 Sealing the Exposed Weaknesses in a Defending Virtual Machine 155

9 Ensuring Type Safety in the Presence of Faults 161
9.1 Protection of the Operand Stack 161
9.2 Towards Type-Confusion-Immune and Type-Safe Platforms 167

10 Securing the Execution Flow 169
10.1 Enhancing the Security of Conditional Branching Instructions 169
10.2 Preventing the Interpretation of Injected Code 177

Conclusion to Part III 183

Conclusions & Perspectives 187

Appendices 189

A Practical Validation of the Fault Model 189
A.1 The test application. 189
A.2 Experimental results. 189
A.3 Conclusions. 190

B Implementation of the Multithreaded Attack on Java Card 3 Connected Edition **191**
B.1 Array Forgery 191
B.2 Request Flooding Validation Thread 191

Publications **193**

Bibliography **195**

List of Figures

1 Représentation interne des instances de classes `B` et `C` ix
2 Accès au même objet en tant qu'instance de `B` ou de `C` ix
3 Exécution de l'instruction INS_ILLEGAL_CAST de l'applet x
4 Exécution perturbée de l'instruction INS_ILLEGAL_CAST de l'applet xi
5 Le scénario d'attaque xiv
6 Influence de la communication sur le nombre d'instructions exécutées. . . . xv
7 Confusion entre deux instances de classes pour forger l'adresse d'un tableau. xv
8 Espionnage et corruption de la communication par un attaquant embarqué. xix
9 Exécution de la méthode d'ajout de redondance sur les instructions de branchement conditionnel opérant sur des booléens. xxii
10 Exécution de la méthode de modification aléatoire du jeu d'instructions. . . xxiii

1.1 An ISO 7816-compliant smart card. 10
1.2 General architecture of the integrated circuit. 11
1.3 Bonding of the chip in the plastic card. 13
1.4 Decapsulated smart card ICs. 13

2.1 Depackaged smart card: back side (left) and front side (right) 16
2.2 Power consumption acquisition on smart cards 17
2.3 CPA of an AES implementation: Correlation curves for the different key hypotheses (left) and key hypotheses ranking (right). 18
2.4 Simple Electromagnetic Analysis of a modular exponentiation (with the corresponding leaking exponent bits). 19
2.5 Fault causes and consequences. [Ver11] 23

3.1 The Java Card architecture 29
3.2 The *frames* and the *stack of frame*. 30
3.3 The tree architecture of Java classes. 31
3.4 The application development process for Java Card *Classic* Editions. 34
3.5 The application development process for Java Card *Connected* Editions. . 35
3.6 Application data isolation with the application firewall. 39
3.7 The sharing mechanism on Classic platforms. 40
3.8 The sharing mechanism on Connected platforms. 41
3.9 The four protection profiles of the *Java Card Protection Profile Collection* [Tru06] 44
3.10 GlobalPlatform Card Architecture (Source: GlobalPlatform [Glo11a]) 47
3.11 Example of secure communication through a Secure Channel. 50

4.1 Power consumption of the card during execution of the applet List. 4.2 resampled and averaged over 10,000 samples [VWG07]. 56
4.2 Observed internal representation of an array in the system memory. 60
4.3 Assumed internal representation of `Fake` objects (left) and `byte[]` (right) . 62
4.4 Operation of the linker on `.CAP` file components. 64
4.5 Assumed card internal structure. 65
4.6 Field access in the confused object instance. 67
4.7 The `debit` method's control flow graph. 72
4.8 The `debit` method's tagged control flow graph. 72
4.9 Fault causes and consequences. 80

5.1 Internal representation of instance of `B` and `C` 84
5.2 Access to the same object either as `B` or `C` instance 84
5.3 Execution of the applet's INS_ILLEGAL_CAST instruction 86
5.4 Disturbed execution of the applet's INS_ILLEGAL_CAST instruction 86
5.5 Forgery of object `a`'s reference 87
5.6 Identification of `dummyMethod` and ill-formed code injection 95
5.7 The call of the `verify` method 96
5.8 Making the signature verification always successful 97
5.9 Assumed internal representation of a String object. 98
5.10 Java Card 3 authenticator classes and interfaces hierarchy. 100

6.1 Alleged architecture of the system. 105
6.2 Different requests handling 106
6.3 Normal (up) and curtailed (down) execution of a thread 107
6.4 The complete attack scenario 108
6.5 Confusion between instance of two classes in order to forge an array's address. 110
6.6 Memory dump from the forged array. 111
6.7 Influence of communication on instructions execution. 112
6.8 USB smart card acquisition module. 113
6.9 Execution of the two threads with various number of ping requests (respectively 0, 10, 30 and 40). T_1 and T_2 refer respectively to the attacker's and the target thread. 114
6.10 Memory dump from the forged array. 115

7.1 SIO client at initialisation `http://smartcard/sioclient/index`. . . . 129
7.2 SIO client get value `http://smartcard/sioclient/getvalue`. 129
7.3 Illustration of the class loading delegation hierarchy. 130
7.4 Breaking the Secure Channel with the Eavesdropping Restartable Task . . 146

A.1 Evolution of the operand stack content and of the top-of-stack (tos) along execution of lines 5 and 6 of the test applet. 190

Part I

Context and State of the Art

Introduction to Part I

The main topic of this dissertation is the study of fault injection attacks on Java Card platforms, of their combination with malicious applications and of their potential consequences in order to enhance the security of these systems against such attacks. The aim of this first part is to set the context of this work in order to let the reader apprehend the different notions as well as the problematic of the smart card security field and of the Java Card security in particular.

Chapter 1 describes the general context of the smart card field. Smart cards are nowadays used throughout the world on a daily basis, wherever the notion of digital security appears. This is due to the fact that smart cards have emerged as the privileged support for letting anyone ensure the confidentiality and integrity of their personal data thanks to cryptographic means. The evolution of various fields such as mobile communication, banking, pay TV or identity management has led to a massive deployment of these cards. Another important factor for the worldwide adoption of smart cards lies in the standardization effort started in 1987 which allows nowadays to use a single smart card in each and every card terminals around the world. Some of the characteristics of a smart card, mainly regarding the physical and communication aspects, are therefore specified by standardizing documents [ISO03, ISO98, ISO00]. On the other hand, the different components of the integrated circuit can vary from one card to another. However their general architecture and basic components are known.

Because of the sensitivity of the data they contain and of their inherent robustness, smart cards are the target of particular kinds of attack: hardware attacks. These attacks encompass both the so-called *Side Channel Analysis* (SCA) and *Fault Attacks* (FA). SCA aims at gaining information on the secret data manipulated by the system through a physical *leakage* "medium" which could be for instance the power consumption or the electromagnetic radiation of the system while manipulating the data. Usually a single observation of the *leakage* does not allow to recover the secret data, and statistical tools are required to extract information from a sample of *leakages*. FA are based on the attacker's ability to tamper with a device during the execution of a given program by modifying its physical environment. Such perturbations can have various origins detailed in Chapter 2 and may lead to disrupt the program execution flow and consequently to avoid the execution of a portion of code and/or to return erroneous values. Another possibility is that the physical perturbation corrupts the data being manipulated by the code which is being executed on-card, leading to roughly similar consequences. Both these attacks have been widely studied with regards to cryptographic algorithms implementations and

several countermeasures both at the hardware and software level have been developed. Chapter 2 gives a brief overview of these attacks and countermeasures.

Chapter 3 introduces the Java Card technology which is the center of this work. Java Cards have been introduced in 1996 and have rapidly become the world leading smart card platform. This fast adoption is mainly due to the progress allowed by this technology in terms of application deployment. Indeed, the Java Card technology allows the same Java Card application to run on all smart cards implementing the Java Card specifications, thus drastically reducing the application development cost observed when developing one native application several times for each and every different devices. This facility has also pushed the development of a full ecosystem around the Java Card technology allowing to easily deploy applications on the cards after they have been released on the field (*post-issuance*) and to integrate them in the various smart cards fields of application. In addition, the Java Card technology has introduced the notion of multi-application platforms, making it possible to store and execute several applications on a single smart card. Last but not least, the Java Card technology comes with several particular security features and Java Card applications can be considered with reasons as more secure than native ones.

Being the world leading technology, Java Card is particularly a target of interest for attackers wishing to challenge its security and extract the sensitive information it may contain. Chapter 4 shows how the attackers have used the ease of application development and deployment to run malicious applications on the platform. These applications can allow attackers to gain information on a given platform and/or other applications hosted on this platform, either logically or through SCA. In addition, they may explore and exploit possible security weaknesses either in the Java Card specifications or in a particular implementations of these specifications. But the largest part of the software attacks in the state of the art uses the notion of ill-formed applications allowing the attacker to bypass certain of the security mechanisms enforced by the platform. However with the generalisation of the use of static analysis tools allowing to detect these ill-formed applications, the notion of *Combined Attacks*, introducing hardware attacks in the Java Card community, has emerged.

Chapter 1

Smart Cards in a Nutshell

Contents

1.1 A Disputed Invention 5
1.2 From a Public Phone Card to a Digital Safe 5
1.2.1 Public Phone Cards, the First Mass Market 6
1.2.2 Modern Utilizations: Mobile Phone, Banking, Identity 6
1.2.3 Perspectives 8
1.3 The Architecture of a Smart Card 9
1.3.1 An Integrated Circuit within a Plastic Card 9
1.3.2 The Integrated Circuit 11

1.1 A Disputed Invention

The invention of smart cards is actually claimed by several persons. Some sources [RE03] state that as soon as the late 60s, Giesecke & Devrient engineers, Helmut Gröttrup and Jürgen Dethloff, invented an automatic chip card for which a patent was filed only in 1982 [GD82]. In 1970, PhD. Kunitaka Arimura patented a similar concept in Japan. In the meantime, two French inventors, Roland Moreno and Michel Ugon patented respectively the concept of memory card in 1974 [Mor74] and microprocessor smart card in 1977 [Ugo77]. These inventions are the basis of the future smart card industry and of its different actors. The different evolutions of these actors could be the subject of a dissertation alone, therefore no more details will be given on these aspects.

1.2 From a Public Phone Card to a Digital Safe

Advances in fields related to microelectronics have been outstanding from the late 60s to nowadays. Of course, the capabilities, and therefore the usage of smart cards has followed the evolution of the technologies they are made of. This section describes the different uses from the first memory cards used as public phone payment card in the early 80s to today's contact/contactless credit cards, evolved SIM (Subscriber Identity Module) cards or even embedded web servers integrated into USB (Universal Serial Bus) tokens.

1.2.1 Public Phone Cards, the First Mass Market

From 1978 to 1983, the French *Direction Générale des Télécommunications*[1] (Directorate-General for Telecommunications) developed the principle of chip cards as payment cards for public phones in order to cope with phone box vandalism. The German Federal Post Office started a similar project in Germany in 1984-85, although the used technology was different from the French cards. Soon, many other countries followed these initiatives.

The technology inside these chip cards was yet quite basic. For instance on French cards, the chip was indeed only aimed at manipulating a counter stored in its non-volatile memory. The memory bits were incrementally set with the consumption of a given unit of time. Consequently, when the user has consumed all the bought time units, all the memory bits were set and the card was not functional anymore. It is worth noticing that the first bits of the memory were set at manufacturing time in order to prevent the success of an attack aiming at resetting the whole content of the memory.

These cards did meet a great success and as much as several hundred million cards were deployed worldwide in the middle of the 90s.

1.2.2 Modern Utilizations: Mobile Phone, Banking, Identity

With the evolution of microelectronic technologies, the microprocessor, or *integrated circuit* (IC), cards emerged and new utilizations of these so-called smart cards were developed. The tamper resistance of the IC allows to store secrets in the card as if it were a tiny digital safe. The possibility to use this property to store cryptographic keys and even to operate the cryptographic computations inside the card opened a wide range of applications in various fields. This section describes the most popular of these fields of application: banking, mobile communication and identity-related applications. Furthermore, the evolution of contactless communication also brought new possibilities.

1.2.2.1 Debit/Credit Cards: *Carte Bleue* and EMV

In the early 80s several French actors joint there effort into the *GIE Carte Bancaire*[2] in order to develop a debit/credit smart card. This effort gave birth to the *Carte Bleue* which was publicly launched in France in 1992. Card users were then able to pay their purchases by the conjunction of their smart card and a secret code: the PIN, for Personal Identification Number, which was typed on the electronic payment terminal.

In 1993, Europay, Master Card and Visa, renowned payment companies started to develop smart card specifications for debit/credit cards [EMV11]. The first specifications were delivered in 1994 and the first stable release was published in 1998. The system maintenance is the charge of an entity called EMVco which JCB (for Japan Credit

[1] which became *France Télécom* in the meantime

[2] The CB Economic Interest Group (`http://www.cartes-bancaires.com/`)

Bureau) and American Express joined respectively in 2004 and 2009. Smart card payment is today really widespread. According to www.emvco.com, more than 1.3 billion EMV-compliant payment cards are deployed worldwide as of 2011's third quarter.

But the real catalyst for the explosion of the smart card market has been the introduction of mobile communication and of its SIM cards in the early 90s.

1.2.2.2 UICC for the GSM/UMTS Networks

The Global System for Mobile Communication (GSM for short) is a standard developed by the ETSI (European Telecommunications Standards Institute) that has been released in 1991. The Universal Mobile Telecommunications System (UMTS for short) is so to speak the successor of the GSM and has been introduced in 2002. It is not in the scope of this dissertation to introduce the basics of these mobile communication networks. However it is interesting to wonder why it has been chosen to integrate a smart card into the mobile phone of each user. The reason for this choice is indeed manyfold. First, the smart card stands as a secure module, allowing to store the secret data of the subscriber and allowing his authentication on the network, hence the name given to such cards: Subscriber Identity Module. Second, it is appreciable that a customer is able to transfer his operator account from one phone to another by simply shifting his smart card from one handset to another. And last but not least, it allows the operator to store its own applications and data on the card without requiring the phone manufacturer authorization.

Such cards are often referred to as SIM cards, because on 2G networks, the used smart cards were only embedding one application: the SIM application. Nowadays the generic term UICC (for Universal Integrated Circuit Card) is preferred since one single smart card is likely to contain a SIM application for GSM networks, a USIM application for UMTS networks and possibly other applications, such as an NFC (Near Field Communication) payment application for instance.

The UICC market is today the largest one in the smart card industry with 4.6[3] billion UICCs shipped worldwide in 2011.

1.2.2.3 Identity Cards, Passports and Access Control

Subsequently, it appeared that smart cards can also be used to prove the identity of its owner within a given system. The concept of identity cards can then be split into two subcategories.

The first category is that of national identity cards and passports, which are delivered by the governments of countries. The chip is here integrated in the national identity document (ID card, passport) and attest for the identity of a citizen. It can be used to pass borders, to log in for electronic public services, *etc*. The need for security is easy to understand regarding the perpetual war between governments and the organized crime

[3] http://www.smartcardalliance.org/pages/smart-cards-intro-market-information

concerning identity document frauds and forgeries.

The second category is that of company cards, which are used to access the building of a company, log into the IT system, or merely pay a lunch. Although the need for such a strong security concerning the lunch break is not obvious, accessing the confidential data of a company can turn out to be very profitable for the offender and disastrous for the company. The use of sophisticated security mechanisms is then again not exaggerated.

The use of smart cards is not limited to the applications depicted in the previous sections. Public transportation, e-health, loyalty programs or smart-metering are fields where the smart card is already implanted and will most probably prosper. Not to mention the pay TV field which is indeed very much exposed to piracy and where both the broadcasting networks and the smart card issuers have to step-up constantly with hackers in order to enhance the security of the whole system.

1.2.3 Perspectives

The growth of the number of smart cards deployed worldwide each year from 4.5 Billion in 2009 to 6.3 Billion in 2011 [4] is quite outstanding. The constant evolution of semiconductor technologies which has already led from the original 8-bit micro-controllers with a frequency of 7 MegaHertz (MHz), embedding a few kiloBytes (kB) of ROM (Read-Only Memory) and NVM (Non-Volatile Memory) and a few Bytes (B) of RAM (Random Access Memory) to current 32-bit micro-controller with a frequency of up to 66MHz and embedding several hundreds kB of ROM and Flash memory and a few kB of RAM let us foresee always more evolution in the smart card field.

Furthermore new challenges arise. Indeed, digital security and privacy are matters that are more and more discussed with regards to the evolution of the internet and mobile communications. The emergence of the contactless interface also represents an important vector of growth for the smart card market. Also, smart cards are likely to take a part in emerging fields such as Trusted Platform Modules, cloud computing, e-Health, smart-metering.

As illustrated in this section, smart cards are present in various fields of application. In each of these fields it has a crucial role to play, which is to ensure the security of both the secret data it holds and the system in which they are integrated by providing appropriate cryptographic credentials allowing to establish trust between the different entities involved. The necessity for the different actors to constantly challenge their security appears then obvious.

[4] Source Eurosmart: `http://www.eurosmart.com/index.php/publications/market-overview.html`

1.3 The Architecture of a Smart Card

This section describes the physical characteristics as well as the communication protocols standardized by the ISO/IEC in [ISO03, ISO98, ISO00]. Subsequently, it introduces the architecture of the *Integrated Circuit* (IC) itself and the role of its different components.

1.3.1 An Integrated Circuit within a Plastic Card

In order to allow a worldwide deployment and interoperability, a great effort has been taken since the late 80s to standardize the physical characteristics of smart cards, their communication facilities and the data organisation inside it. The following briefly presents these standards and their main requirements regarding the physical characteristics of the smart card and gives a more detailed view of the communication protocols standardized by the ISO.

1.3.1.1 Physical requirements.

A smart card is basically, an integrated circuit bonded within a plastic card. The ISO/IEC 7810 and 7816 standards [ISO03, ISO98] detail the physical characteristic of the plastic card, of the integrated circuit's contacts, as well as the communication interface and protocols between a card and a card reader. According to the ISO/IEC 7810 standard, the plastic card shall be between 85.47 and 85.72 mm in length and between 53.92 and 54.03 mm in height. Its thickness shall be between 0.68 and 0.84 mm. In addition, the ISO/IEC 7816-1 defines the physical resistance of the smart card (including the IC and its contacts) to external stress such as bending, torsion, high/low temperature, X-rays exposure, etc... The number and location of the contacts of the integrated circuit are also eligible to strict requirements as per the ISO/IEC 7816-2. An ISO 7816-compliant card shall provide 8 contacts according to the disposition illustrated in Figure 1.1, where:

- Contact C_1 is assigned to VCC (supply voltage);
- Contact C_2 is assigned to RST (reset signal);
- Contact C_3 is assigned to CLK (clock signal);
- Contact C_4 is Reserved for Future Use (RFU);
- Contact C_5 is assigned to GND (electric ground);
- Contact C_6 is assigned to VPP (variable supply voltage, *i.e.* programming voltage);
- Contact C_7 is assigned to I/O (data input/output signal);
- Contact C_8 is RFU;

It is worthwhile noting that C_4 and C_8 are more and more used as USB (Universal Serial Bus) contacts on smart cards supporting this protocol. Also, the VPP contact becoming obsolete, it is often used on contactless devices as an interface for the Single Wire Protocol (SWP) [Eur10]. The used contacts are then connected to the IC itself (see Section

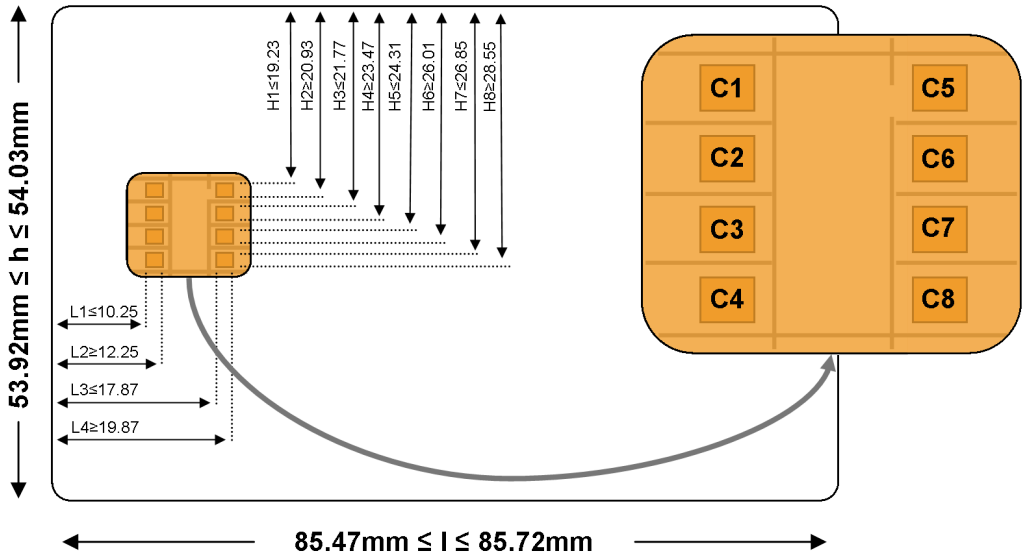


Figure 1.1: An ISO 7816-compliant smart card.

1.3.2) and are the support for the communication between the card and the reader.

In the case of a contactless card, the IC is connected to an antenna integrated within the plastic body of the card. The communication is achieved over the air thanks to the RFID (Radio Frequency IDentification) [ISO04, ISO00] technology.

1.3.1.2 Communication

The ISO 7816-3 standardizes the electronic signal and information protocol between a smart card and a reader. In particular, it describes the asynchronous character of the communication between the card and the reader and defines two transmission protocols:

- T=0, the protocol for half duplex transmission of characters,
- T=1, the protocol for half duplex transmission of blocks of characters.

Similarly, the ISO14443-4 standardizes the transmission protocol in contactless mode, often referred to as T=CL.

In addition, the ISO 7816-3 defines the Application Protocol Data Unit (APDU) which is the structure of the commands and responses received and emitted by the card. A particular interest is given here to this level of communication because it is at this level that the Java Card abstraction layer starts. That is to say, neither the JCRE, JCVM or Java Card applications have the hand on the protocol used by the card or the specific signal processed at the hardware level. On the other hand, every APDU exchanged between the card and the terminal is treated by the JCRE and possibly dispatched to one of the Java Card applications installed on the platform. These notions are more precisely defined in Chapter 3.

An APDU command is compound of two parts:

- A four-byte header denoted CLA INS P1 P2,
- A conditional part of variable length denoted [LC [DATA]] [LE], where LC stands for the length of the DATA field and LE stands for the Length of Expected response data.

The presence or not of these different fields in a particular command are referred to as particular *Cases* (as per [ISO98]). For instance, a command that do not expect to receive data from the card in the response APDU will not hold an *LE* byte and is referred to as a *Case 3* command.

Similarly, an APDU response is compound of two parts:

- A conditional part of length LE denoted [DATA],
- A two-byte status word generally reporting either the success or the failure of the command (with a specific error code in the latter case) denoted SW1 SW2.

As the name of *command* let expect the values contained in the different fields of an APDU command defines the behaviour of the card or of an on-card application when it receives it. Therefore, it also conditions the response APDU that will be returned by the card.

Until now this description has mainly focused on the external characteristics of the card. The following describes the integrated circuit that is bonded within the plastic card described above.

1.3.2 The Integrated Circuit

Unlike the communication protocols and contacts, the inside of a smart card is not much publicly documented. However, the general architecture of the IC is quite obvious with the presence of a microprocessor, memory cells, possibly cryptographic co-processors (*cf.* Figure 1.2).

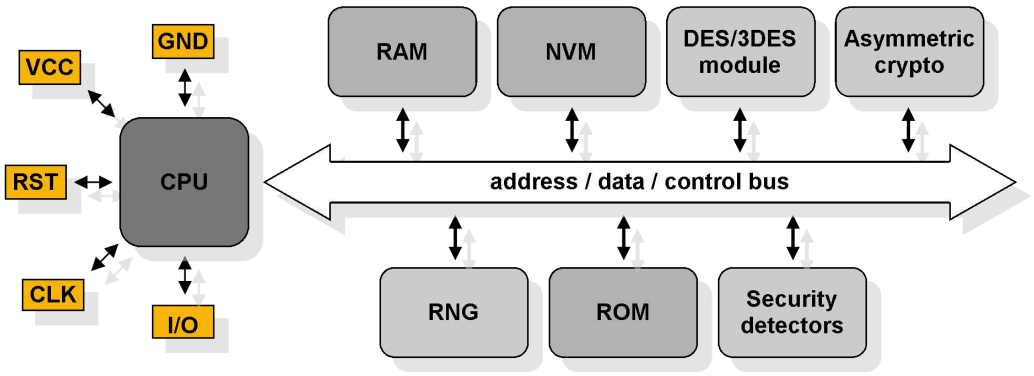


Figure 1.2: General architecture of the integrated circuit.

These different parts are more precisely defined hereafter:

- The Central Processing Unit (CPU) is the component that is in charge of processing data. It is considered as the brain of the smart card and is compound of different parts allowing it to handle data and addresses, to compute arithmetic and logic operations, to control the inputs and outputs, and to fetch, decode and execute the different instructions provided by the instruction set. Mainly two different CPU architectures can be found on smart cards:
  - The CISC architecture, for *Complex Instruction Set Computer*, commonly used in 8-bit micro-controllers such as Intel's 8051, which provides a large instruction set, but where each instruction requires several clock cycles.
  - The RISC architecture, for *Reduced Instruction Set Computer*, generally found in 16- or 32-bit micro-controllers such as ARM-based CPUs. This kind of architecture is nowadays becoming the most common one in the smart card industry.
- The Read Only Memory (ROM) is a particular kind of memory that can be written only once. It is typically written during the production process of the card with the basic operating system (OS) of the card. The historical term of *mask* comes from the fact that the card OS is initially "burnt" into the chip during the manufacturing of the chip by lithographic processes thanks to a mask.
- The Random Access Memory (RAM) is a writable memory that is very fast but which is volatile, *i.e.* the data it contains are lost when there is no more supplied power. It is therefore used to store temporary data such as local variables of a program for instance. This is the most expensive kind of memory in terms of required space on the silicon chip. Smart cards are consequently endowed with a little amount of RAM.
- The term Non-Volatile Memory (NVM), by opposition to volatile memory like RAM, refers nowadays to mainly two kinds of writable memory in the smart card field: Electrically Erasable Programmable Read Only Memory (E^2PROM) and Flash memory. It is likely that the Flash memory will outstand E^2PROM in the future because it offers better performance at an equivalent cost.
- The Cryptographic coprocessors: hardware DES (Data Encryption Standard) [Fed99] and AES (Advance Encryption Standard) [Fed01], asymmetric crypto, T/P-RNG (True-/Pseudo-Random Number Generator), are dedicated areas of the silicon chip that are only meant to cipher/decipher data or to produce random numbers for instance. These modules use particular registers referred to in manufacturers manual as SFRs, for Special Function Registers, which are cautiously dispatched on the silicon area in order to improve their robustness against hardware attacks.
- In response to state-of-the-art hardware attacks detailed in Chapter 2, IC designers have added several sensors and detectors allowing the IC to detect potential attacks. Light sensors or power/clock glitches detectors are nowadays common in such devices for instance.

The chip is subsequently integrated in the card itself as described in Figure 1.3. As stated in the previous section, the position of the IC on the plastic card and the electric contacts are precisely defined in the standardizing documentation. Figure 1.3 illustrate

how the ISO contacts are connected to the IC through the bonding wires.

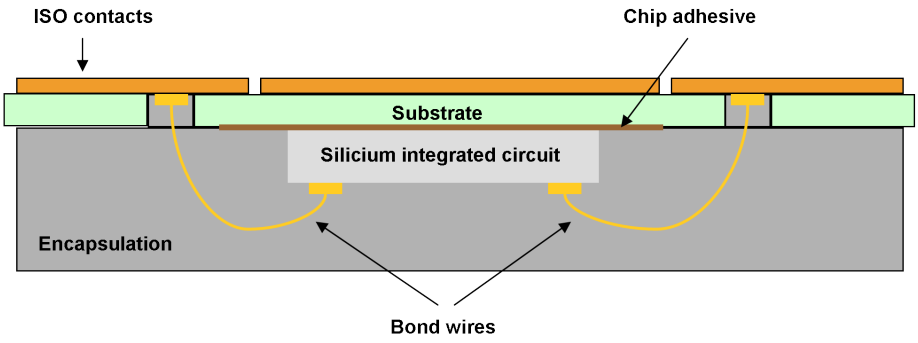


Figure 1.3: Bonding of the chip in the plastic card.

The exact design of a chip is always a secret preciously kept by manufacturers. It is therefore a typical target for various attackers willing to bypass certain security mechanisms provided by the chip. A first step of such attacks consists in identifying and locating the different components of the IC. Although these attacks are obviously not always publicly announced, the results presented by Christopher Tarnovsky and Karsten Nohl [TN11], which provides us with the following pictures (Figure 1.4) presenting decapsulated ICs on which the different parts can be identified, can be cited here.

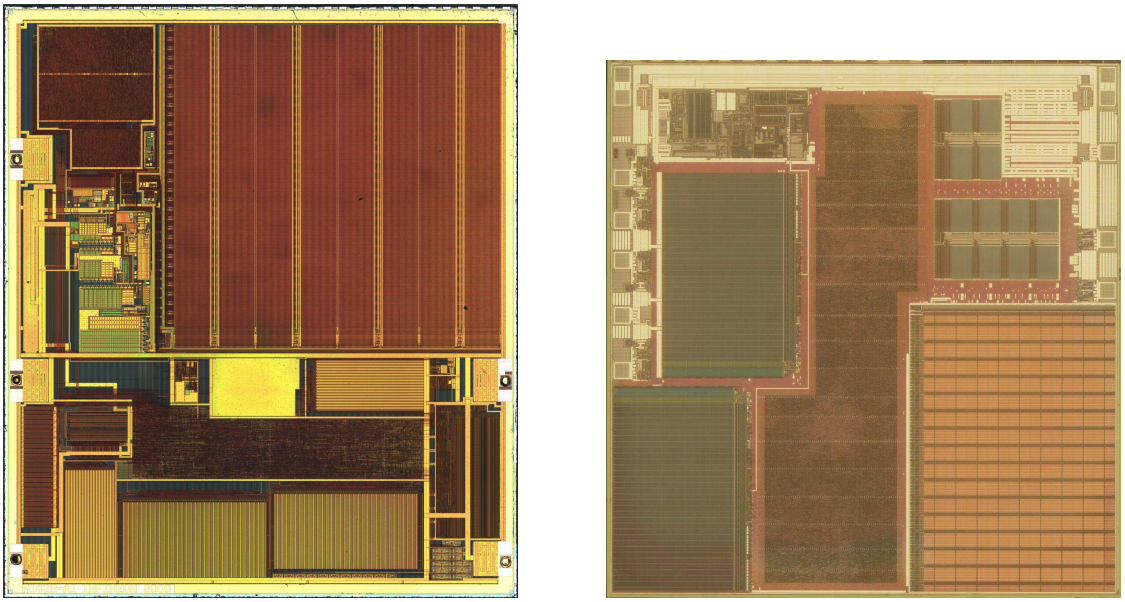


Figure 1.4: Decapsulated smart card ICs.

Such information also reveals crucial when intending to put into practice hardware attacks against smart cards, as will be exposed in Chapter 2.

Chapter 2

Hardware-Related Attacks

Contents

2.1 Side Channel Analysis 16
2.1.1 Brief History 16
2.1.2 Side Channel Leakage Media 16
2.1.3 Cryptographic Key Recovering and Reverse Engineering 17
2.2 Fault Attacks on Embedded Systems 19
2.2.1 Faut Injection Techniques and Fault Models 20
2.2.2 Fault Attacks against Cryptosystems and Countermeasures . . . 21
2.2.3 Fault Attacks Everywhere 22

Smart cards, and more generally embedded systems, are meant to operate in various environments once they are released. This is indeed one of the interesting thing with smart cards. But this also gives the opportunity to attackers to manipulate them at will, making the execution environment hostile in various considerations.

In this context, a large spectrum of attacks based on physical properties of the attacked device has been developed. In the literature (see [RE03]), these attacks are generally classified in different categories, whether they are:

Non-invasive meaning that the attacker does not alter the device under attack.

Semi-invasive meaning that the attacker alters the device under attack (see Figure 2.1), but leaves it in a functional state.

Invasive meaning that the attacker alters the device under attack, and is likely to leave it out of order.

and whether the attacker is considered to be:

Passive if she only observes the device under attack.

Active if she disrupts the behaviour of the component.



Figure 2.1: Depackaged smart card: back side (left) and front side (right)

The remainder of this chapter mainly considers two kinds of attack. The first one belongs to the *passive* and *non-invasive* (or *semi-invasive* in certain cases) categories and is called *Side Channel Analysis*, or *Side Channel Attacks* (SCA). The second one is an *active* and *semi-invasive* kind of attack, based on a fault induced during a given operation, hence its name: *Fault Attacks* (FA). Invasive attacks such as the use of focused ion beam for micro-probing purpose are out of the scope of this dissertation and will not be described here. Interested reader may refer to [RE03, §8.2] and [Gia04].

2.1 Side Channel Analysis

2.1.1 Brief History

It is difficult to state precisely when the first SCA against an automated cryptosystem was achieved. This uncertainty comes from the fact that the first entities active on this field appeared to be the secret services of different nations. Yet, published documents [Wri87, DC66, Kah96] tends to prove that both the British MI5 and the American NSA have used such attacks since at least the early 1950's.

It is in 1996 that Paul Kocher published an article [Koc96] presenting the first *Timing Attack* against different cryptographic algorithms implemented on embedded systems. In this work, the author exposes that observing the execution timing of a particular cryptographic process gives sufficient information to recover the secret key used. This seminal work has brought the interest of the scientific community and many other articles followed. Until today, this research field has been very active in developing novel attacks, either by using new leakage media, or new attack techniques, as well as countermeasures to state-of-the-art attacks [KJJ98, GMO01, BCO04], at either the software or hardware level.

2.1.2 Side Channel Leakage Media

SCA are based on the implicit assumption, that information leaks through the considered side channel.

Soon after the first published *Timing Attack*, the idea of using the power consumed by the embedded system during specific operations as a Side Channel leakage medium emerged [Koc96, KJJ98]. Similarly, it has been shown that other side channel leakage

media are likely to give useful information to an attacker, such as the electromagnetic (EM) radiation of the system [GMO01, QS02]. Other more exotic leakage media have been described in the literature, such as the acoustic noise or the heating of the system [ST04, KJJR11]. In all cases the observed leakage depends then on observable physical characteristics of the attacked component, based on its very basic components.

For instance, it is known that the energy consumed by a system depends on the operation it executes. By measuring the power actually consumed by the device, an attacker gains a macroscopic view of its behaviour. On the other hand, the electromagnetic side-channel can provide the attacker with much more localized information [QS01, SGM09], and therefore valuable information on the chip itself and furthermore on the manipulated values. On smart cards, this measurement can be operated as described in Figure 2.2 with regards to the power consumption of the device.

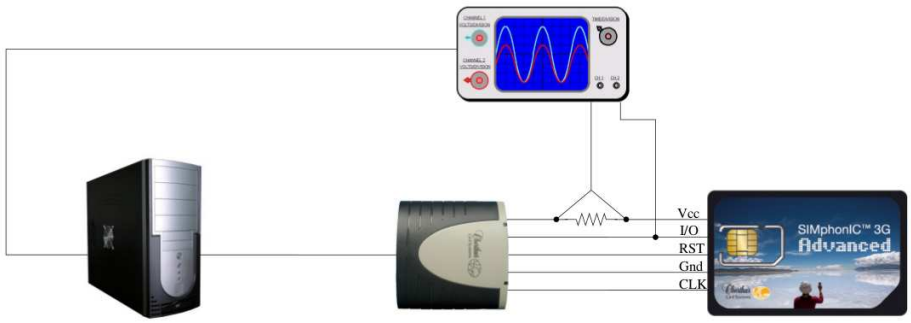


Figure 2.2: Power consumption acquisition on smart cards

The SCA relies on the assumption that the observed leakage media depends on the instruction that are actually being executed by the chip and on the values that are being manipulated. Different attacks can be mounted whether the attacker observed particularly the instruction flow or the processed data. The following gives an overview of these attacks.

2.1.3 Cryptographic Key Recovering and Reverse Engineering

In order to expose the information contained in the side channel leakage, the use of several statistical tools on the measured observations have been considered. The various attacks published [Koc96, KJJ98, BCO04, DPRS11] mainly vary in assumptions on the data leakage and in the distinguisher used as part of the statistical processing of the side channel leakage sample (usually power consumption and electromagnetic radiation curves). However, the general idea is always to highlight a dependency between the observed side channel and (a part of) the secret that is being manipulated.

Figure 2.3, for instance, depicts the result of a Correlation Power Analysis (CPA) intending to exhibit a correlation between the power consumption of the card and a temporary byte which value depends on a secret key byte of the executed AES (Advanced

Encryption Standard) [Fed01].

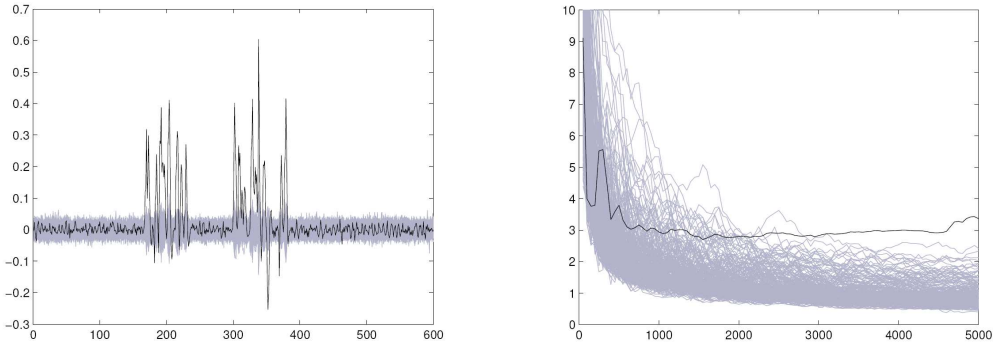


Figure 2.3: CPA of an AES implementation: Correlation curves for the different key hypotheses (left) and key hypotheses ranking (right).

The left graph of Figure 2.3 shows the correlation between the power consumption of the card and hypotheses on the temporary byte depending on the key. One can observe that the correct hypothesis (depicted in black in the figure) presents a strong correlation with the power consumption of the card at a given time. This correlation peak therefore reveals the correct key hypothesis. The right graph of Figure 2.3 shows the number of acquired power consumption curves necessary to have one particular key hypothesis standing out. This correlation convergence curve shows here that 3,000 acquisitions were required to exhibit the correct hypothesis and thus to retrieve the secret key byte.

To thwart such attacks, the most used countermeasure consists in breaking the dependency between the manipulated data and the secret key. Such a *masking* is usually operated by combining the manipulated data with random data, typically with a XOR operation [GP99, AG01]. The notion of High Order SCA (HOSCA) [SP06, CPR07] subsequently appeared, allowing to attack implementations using the masking countermeasure. However, these attacks are more difficult to mount, and the notion of high order masking has been developed against these HOSCA.

Another possibility for an attacker to take advantage of the side channel leakage is to analyze it in order to reverse engineer the instructions executed by the observed system. The first example of such attacks, given in [KJJ98], shows that the observed side channel allows to determine the instruction flow, which in turn reveals the private key of an RSA cryptosystem [RSA78]. The success of this attack relies on the implementation of the *Square-and-Multiply* algorithm to operate the modular exponentiation as part of the RSA encryption. This algorithm operates a bitwise exponentiation where a squaring is executed if the exponent bit is 0 and a squaring and a multiplication are executed if the exponent bit is 1. As exposed in Figure 2.4, the Simple Electromagnetic Analysis allows to identify the exponent's bits by identifying the *Square* and *Multiply* pattern.

Finally, several publications [AFV07, GSM$^+$10] have exposed how similar techniques

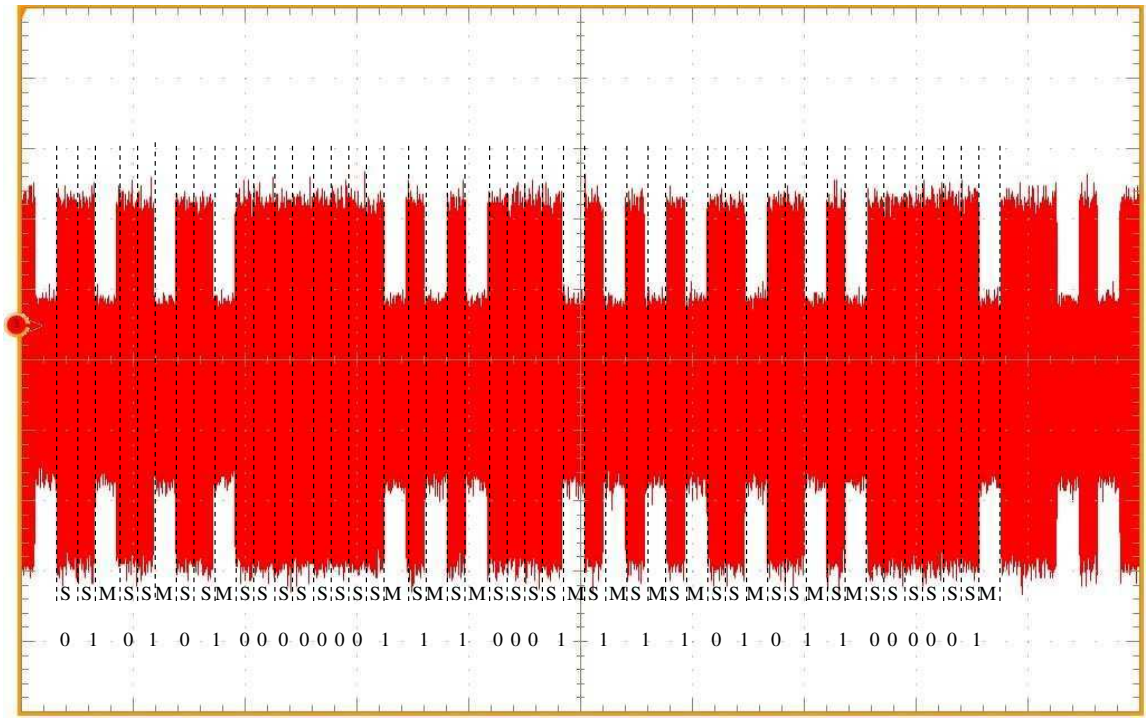


Figure 2.4: Simple Electromagnetic Analysis of a modular exponentiation (with the corresponding leaking exponent bits).

could also be used to reverse engineer secret cryptographic operations. On the other hand, Thomas Einsenbarth *et al.* have exposed in [EPW10] that a thorough side channel analysis of the different instructions supported by a device allows an attacker to characterize these instructions with regards to the leakage and consequently to reverse engineer any embedded native application.

2.2 Fault Attacks on Embedded Systems

The question of the behaviour of an electronic module exposed to a critical environment has been widely studied in field where the fault resilience is crucial, such as in the aerospace engineering. The aim of researchers in this field is to ensure that a system will still run in the case of a particular natural event such as cosmic rays. Different techniques were then developed in order to simulate the effects of these natural events, one of them being the use of lasers [Hab65, SMF97].

Since Dan Boneh *et al.* published an article [Bel96, BDL97] proving that embedded cryptography can be broken by fault attacks, the use of these very same techniques has rapidly spread to analyze the vulnerability of embedded cryptosystems. Consequently, new attacks were built and specific countermeasures to these attacks were designed, both hardware and software.

This section introduces the basis of Fault Attacks and exposes the results that have been observed on smart cards, most often with regards to embedded cryptosystems.

2.2.1 Faut Injection Techniques and Fault Models

Inducing Faults The main idea behind the fault injection techniques against embedded systems is to perturb the context of execution of the system. As part of this context of execution, and considering a smart card, we can outline several variables. Some of these variables are internal, such as the location of the different component of the IC (CPU, RAM, ROM, NVM). Others are external such as the signals provided by the terminal (clock, I/O) or the physical environment. This explains why there exists various means of provoking an error on a smart card:

Power glitches which consist in applying a very short voltage peak on the power supplied to the chip.

Clock glitches which consist in provoking a very short modification in the clock signal sent to the chip.

Visible light which consists in exposing the chip to a short flash of visible light.

Invisible light which consists in exposing the chip to a short flash of infra-red light.

Electromagnetic radiation which consists in exposing the chip to a specific electromagnetic field.

In the context of vulnerability analysis on smart cards, the use of the optical induction of errors in the IC through laser beams is widely spread, mainly because it allows attackers to be sufficiently precise with regards to both timing and location of the fault injection. Also the use of electromagnetic radiation generator is spreading nowadays to induce faults in ICs.

The first FA based on optically induced errors was presented by Sergei Skorobogatov *et al.* in [SA02]. In this article the authors describe how a low-cost equipment such as a camera's flash can be sufficient to perturb the behaviour of a smart card. Since then, attacks have evolved and the use of laser beams applied on either the front side or the rear side of the chip is nowadays common in evaluation laboratories. On the other hand, both IC and smart card manufacturers have developed several countermeasures to protect their products from such attacks, adding light sensors and developing fault-aware implementations for instance [RE03, §8.2].

Such optical fault injection does work because of the current induced by the light beam's photon into the circuit by photoelectric effect [Sor87]. The photons of the laser beam impinge against the semiconductor and are absorbed in a so-called *interband absorption across the semiconductor gap* (*i.e.* electrons jump from the valence to the conduction band if the energy of the photons is greater than the semiconductor gap) which leads to the generation of electron-hole pairs and then of a current source. The fault itself occurs because the IC components such as memory cells and logical gates are sensitive to this induced current.

Fault Models An interesting question to answer when considering fault injection is then: what is the consequence of the fault on the system?

Indeed, and considering laser-induced faults, the consequence will depend on a couple of parameters:

- The location of the laser shot on the circuit. That is to say, whether the laser beam is applied on the circuit logic, on memory cells, on data or address buses, ...
- The time of the laser pulse. Obviously, powering on the laser beam at different moments in time will have different consequences.
- The wavelength of the laser beam. Since the depth of the propagation of the energy brought by the laser beam depends on its wavelength.
- The duration of the laser pulse. That also allows to modulate the energy added in the circuit.
- The size of the laser beam when hitting the system. It allows to precisely target a specific area on the chip's surface.
- The intensity of the laser pulse.

In order to have a theoretical approach of the induced error, the literature generally refers to the different fault models listed in Table 2.1.

Fault model	Precision	Erroneous value
precise-bit-chosen-error	A chosen bit	A chosen value (0 or 1)
precise-bit-random-error	A chosen bit	A random value (0 or 1)
random-bit-chosen-error	A random bit	A chosen value (0 or 1)
random-bit-random-error	A random bit	A random value (0 or 1)
precise-byte-chosen-error	A chosen byte	A chosen value (0x00 or 0xFF)
precise-byte-random-error	A chosen byte	A random value (between 0x00 and 0xFF)
random-byte-chosen-error	A random byte	A chosen value (0x00 or 0xFF)
random-byte-random-error	A random byte	A random value (between 0x00 and 0xFF)

Table 2.1: The different fault models considered in the literature

2.2.2 Fault Attacks against Cryptosystems and Countermeasures

As already stated, the notion of FA has been introduced in the literature as a mean to break embedded cryptography. The following gives an outline of such attacks.

Differential Fault Analysis Differential cryptanalysis pays attention to the propagation of differences between inputs within a cryptographic algorithm. Therefore modern algorithm are generally constructed in order to resist to such attacks. The idea of Differential Fault Analysis (DFA) is to create such a differential bias thanks to an error introduced

during the execution of the targeted algorithm.

DFA attacks have turned out to be very effective against various algorithms (DES, AES, RSA) [BS97, JLQ99, DLV03, PQ03, Gir04]. Embedded cryptographic modules and implementations therefore use state-of-the-art countermeasures to ensure (at least) that erroneous results are not returned to the outside, so that a potential attacker cannot use them.

Safe-Error Attacks Another kind of attack has also emerged: Safe-Error Attacks [YJ00, JQYY02]. These attacks do indeed take advantage of the countermeasures ensuring the validity of the returned values. The point is that if an attacker can target a single bit and stuck it at 0 for instance, then she will know the actual value of this bit whether the execution returns the correct result or not. Therefore, provided the injected fault is easily reproducible for all bits, an iterative attack will allow her to fully recover the used key.

To counteract fault attacks, smart cards usually implement countermeasures consisting in dummy or redundant calculations or using the key and its complement, either at the hardware or software level.

2.2.3 Fault Attacks Everywhere

FA are definitely not intrinsically limited to embedded cryptosystems as exposed by Christophe Giraud et al. in [GT04] and in [BECN$^+$06b]. Since these attacks directly target the underlying hardware, they are likely to be used against any layer of the attacked system, from the card OS to the highest-level applications. This state of fact can be illustrated by Figure 2.5 presented by Ingrid Verbauwhede in [Ver11], considering the arithmetic operations, cryptographic primitives, algorithms and protocols as the upper layers where consequences of the fault happen and the transistors, logic gates, flip-flops and the register-transfer level (RTL) as the lower layer where the fault is actually induced. The introduction of Part II shows that this diagram can be adapted to the Java Card context.

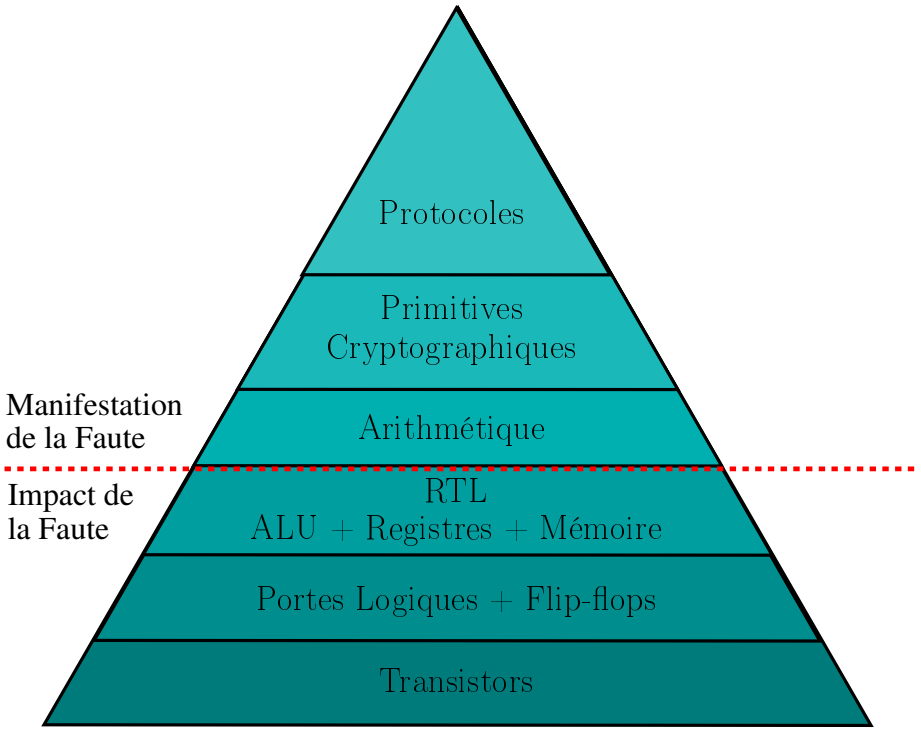


Figure 2.5: Fault causes and consequences. [Ver11]

Chapter 3

The Java Card Technology and its Ecosystem

Contents

3.1 Java-enabled Smart Cards 25
3.1.1 Genesis: Putting Java in a Smart Card 26
3.1.2 The Java Card Standard Evolution 26
3.1.3 The Architecture of a Java Card 28
3.1.4 Developing and Executing Java Card Applications 31
3.2 The Security of Java Card Platforms 35
3.2.1 Inherited Security Mechanisms 36
3.2.2 The Need for Specific Security Features 38
3.2.3 The Java Card Protection Profiles 43
3.3 The Ecosystem of Java Cards 45
3.3.1 GlobalPlatform 45
3.3.2 The (U)SIM/Card Application Toolkit 50

3.1 Java-enabled Smart Cards

Smart cards have strong constraints as for computing power and memory size. Besides, Java is well-known to be relatively slow in workstations. The combination of these two technologies was consequently not obvious in the first place. However, solving several problems until then inherent to smart cards, Java-enabled smart cards have been largely accepted and are today the most used platforms in the smart card field. We first introduce the basic concepts of the Java Card technology, its evolution and explore the reason of its present success as well as the future challenges.

3.1.1 Genesis: Putting Java in a Smart Card

The Java programming language has been developed by James Gosling from Sun Microsystems Inc. (now merged with Oracle Inc.) and publicly released in 1995 [GJSB05b]. It is an object-oriented programming language in which programs are actually executed thanks to an abstraction layer provided by the Java Virtual Machine (JVM) [LY99]. This abstraction layer is indeed the reason for the today well-known moto of the Java language: *write once, run any(every)where*, meaning that Java applications run independently of the underlying hardware and system on which they are actually executed. The point is that if a JVM has been developed for a particular system, then any Java application will be able to run on this system. This property is enforced by a test suite provided by Oracle Inc. called TCK (Test Compliance Kit) which tests the correct implementation of the instruction set and of the required APIs (Application Programming Interfaces). Any virtual machine must pass the TCK in order to be a certified Java Virtual Machine. The JVM will actually interpret the Java application's binaries (.CLASS files) and their Java instructions (the so-called *bytecode array*) and execute the appropriate native instructions on the underlying system.

Soon after the release of the Java programming language, the idea of adapting this philosophy to smart cards appeared. Before this release, the development of applications for smart cards was very expensive since every time a new device was released, application developers had to write their application again, as each device presents hardware specificities and exposes a particular instruction set. The idea behind Java-enabled smart cards was to allow a single Java application to run on any smart card supporting Java.

In 1996, a team from the smart card unit of Schlumberger started working on the future Java Cards. An interesting narration of the history of Java Cards is given by Bertrand du Castel in [dC12]. In the first quarter of 1997, the first Java-enabled smart card: "Java Card 1.0", was produced. Given the strong constraints on the available memory in small devices such as smart cards, this card did only support a subset of the standard Java language, later referred to as the Java Card language. Soon after the first card was produced, the Java Card Forum[1] was created. The Java Card Forum (JCF) is a consortium of the different actors of the Java Card industry, *i.e.* Sun Microsystems (and now Oracle) plus the regular actors of the smart card industry such as card issuers, card vendors and IC manufacturers which are either *members*, *strategic partners* or *observers* depending on their involvement. The aim of the JCF is to discuss the Java Card specifications and directions. It is therefore responsible for the different versions of the Java Card standard described below.

3.1.2 The Java Card Standard Evolution

The JCF published the successor of Schlumberger's Java Card 1.0 specification, named Java Card 2.0, in November 1997 [Sun97]. The Java Card 2.0 specifications present the principles and concepts of Java Card applications, the so-called Java Card *applets*, as well as an Application Programming Interface (API). Yet, they do not specify how the

[1] www.javacardforum.org

applets should be distributed.

In March 1999, three documents are released to specify the version 2.1 of the Java Card technology. This specification trilogy has endured until today. These are:

- The Java Card 2.1 Virtual Machine (JCVM) specification [Sun99c] which defines an adaptation of the JVM meant for smart cards, as well as the Java Card programming language, a subset of the standard Java programming language. A good example of this reduction of the programming language is the removal of the `int` type, which, being coded on 32 bits, did not fit the most common 8-bit ICs of smart cards for memory constraint reasons.

- The Java Card 2.1 Application Programming Interface (API) [Sun99a] which provides the basic blocks of the Java Card programming through various packages of classes that must or may be supported by any smart card compliant with the specifications. This API is obviously drastically reduced compared to the standard Java API.

- The Java Card 2.1 Runtime Environment (JCRE) specification [Sun99b] which defines the runtime behaviour of the system, such as the applet management on the platform, or different notions relative to object instantiation for instance.

These three entities are described in detail in Section 3.1.3.2. Subsequently this version was slightly updated with the version 2.1.1 of the specifications released in May 2000 [Sun00c, Sun00a, Sun00b].

After being first thought as the version 2.1.2 of the platform, the Java Card 2.2 specifications [Sun02c, Sun02a, Sun02b] were released in June 2002. Their main contributions to the evolution of the technology were the introduction of the notion of *logical channel* and of *Remote Method Invocation*. Another novelty of this version is the optional support of the `int` type and of a mechanism similar to the *garbage collector* in Java which deletes object instances that are not referenced (i.e. not used anymore) on the platform. We can also note an update concerning the specified cryptographic API and precisions concerning applet management. The versions 2.2.1 [Sun03c, Sun03a, Sun03b] and 2.2.2 [Sun06c, Sun06a, Sun06b] followed, respectively in October 2003 and March 2006, each updating the API, mainly regarding the cryptography.

The latest evolution of the Java Card technology came with the version 3.0 of the specifications, publicly released in April 2009. This version comes in two editions:

- The so-called *Classic Edition* [Sun09g, Sun09a, Sun09d] which stands as a regular update of the Java Card 2.2.2, mainly concerning the API again.

- The *Connected Edition* [Sun09h, Sun09b, Sun09e, Sun09c], which appears as a major evolution of the Java Card technology. It is detailed in the following section.

3.1.2.1 Java Card 3 *Connected Edition*

The Java Card 3 *Connected Edition* is a web-oriented version of the Java Card technology. Consequently it integrates features to support communication within IP (the Internet Protocol [The81]) networks as well as to conform to web services architecture. The traditional Java Card applet model is replaced in this edition by *servlets*, *filters* and *listeners*, well-known in the web service development community and referred to as the "web application model".

In addition to the legacy features of the Java Card technology, it offers many more facilities to both developers and end-users, such as:

- A 32-bit VM, providing a *de facto* support for the `int` type and an easier support for the `long` type;
- Dynamic Java class loading at runtime;
- Multithreading, *i.e.* the concurrent execution of different applications;
- On-card bytecode verification (OCBV);
- A true automatic garbage collector;
- An embedded web server, allowing the card to serve various resources, either static or dynamic, through HTTP(S) ((Secure) HyperText Transfer Protocol [BLFF96, FGM+99, Res00, FKK11]);
- Communication over USB (Universal Serial Bus) or MMC (MultiMedia Card);
- The Java Generic Connection Framework, providing an API for network connection and communication;
- Support of the `String` class, to handle character strings;
- Multi-dimensional arrays, and collection structures such as `Vector` or `List`;
- Security-related components and features such as a policy-based access control (either based on user roles or on permissions), web application client and card holder authentication, transport-level security (SSL/TLS) [DR06, DR08, FKK11].

As the following of this dissertation exposes, each of the evolutions of the Java Card technology has solved certain security issues. But new facilities often cause new security breaches. As the Java Card 3 Connected Edition brings a lot of new features, it will be discussed at large in the rest of this work.

3.1.3 The Architecture of a Java Card

As exposed in the previous section, the Java Card technology is compound of different layers on top of the smart card hardware, introduced in Chapter 1, and a basic operating system. The applets can subsequently be installed on top of this architecture (Figure 3.1).

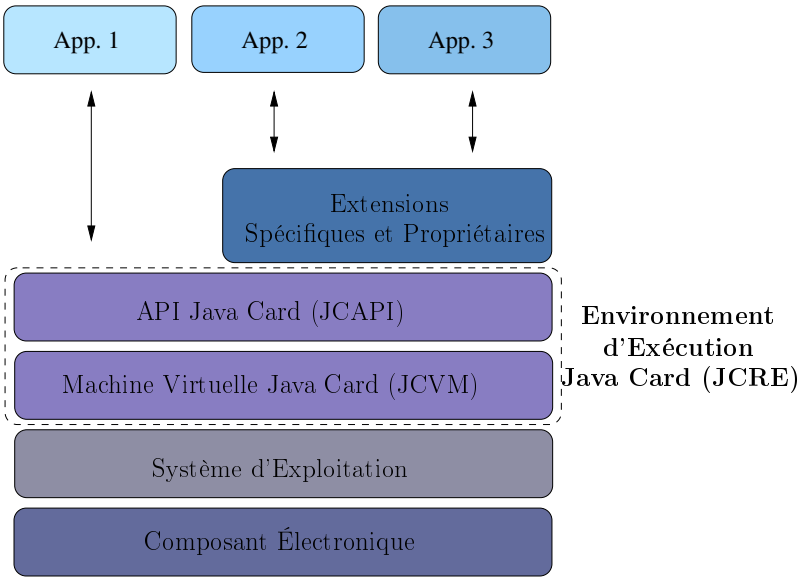


Figure 3.1: The Java Card architecture

3.1.3.1 The Card Operating System

The card Operating System (OS) is responsible for interacting with the smart card hardware. This concerns mainly the memory management, the communication protocols, the access to the cryptographic co-processors, and more generally all the low-level mechanisms.

3.1.3.2 The Java Card Runtime Environment

As illustrated on Figure 3.1, the JCRE is compound of the Java Card Virtual Machine and the Java Card Application Programming Interface. These two parts are described hereafter. Additionally, the JCRE is in charge of the management of the card initialization, power-up and reset, as well as of several security-related mechanisms which are described in Section 3.2.

The Java Card Virtual Machine The JCVM can somehow be seen as an abstract processor with its own instruction set and internal mechanisms. It *interprets* the *bytecodes* into which Java Card applications are translated by the Java compiler. The *interpreter* is an important part of the JCVM which reads the bytecodes and executes the appropriate native instructions. This mimics the *fetch/decode/execute* cycle of a common CPU.

One of the internal mechanisms of the JCVM mentioned above is the management of *frames* of execution relative to each executed method. These frames are organized in a stack (the so-called *stack of frames*), corresponding to the nested calls of several methods (*cf.* Figure 3.2). A frame of execution is a structure of data containing the information necessary to the execution of a method, namely the local variables, the Java

program counter designating the next bytecode instruction to execute, and the operand stack used to pass parameters to the instructions and to read their result. The presence of the operand stack is indeed required because the JCVM does not use *registers* like the processors introduced in Chapter 1. This is the reason why the JCVM (and other Java VM) is called a stack-based machine, by opposition to register-based machine.

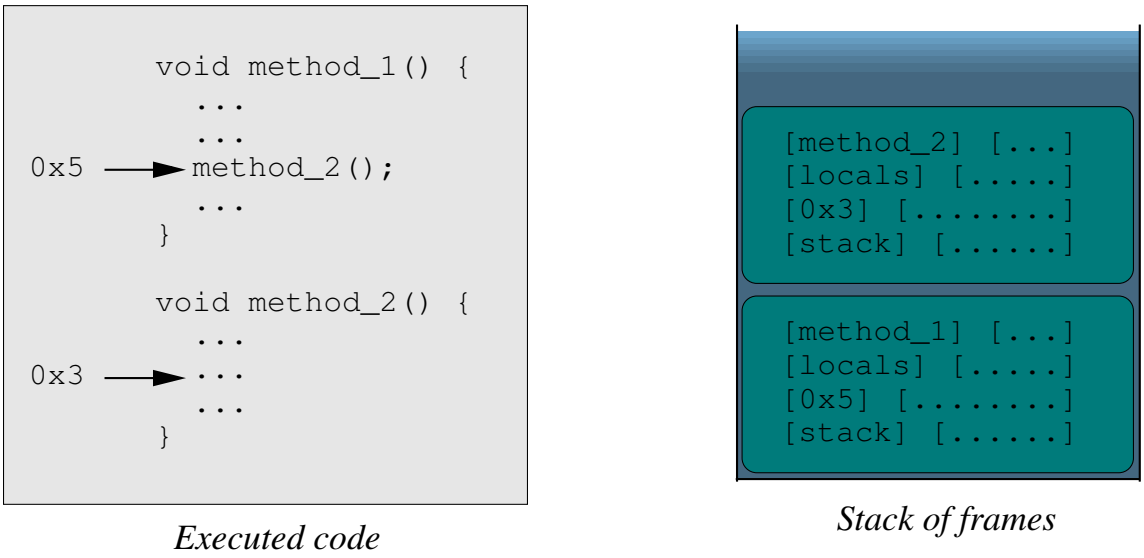


Figure 3.2: The *frames* and the *stack of frame*.

Another role of the JCVM is the management of the Java *heap*, which is the memory containing the Java objects *instantiated* at a given time. Such a mechanism is obviously dependent on both the JCVM implementation and the embedded OS.

The Java Card Application Programming Interface The JCAPI is the part that has the most regularly evolved along the different specifications updates. The role of the JCAPI is in all aspects similar to that of standard Java's. It provides numerous classes in order to allow the developer of an application to focus on the functionality of his application rather than on either basic operations (such as copying the content of an array in another array), complex but "standard" operations (such as generating the digital signature of a given input according to the Digital Signature Algorithm - DSA), or hardware-related operations (such as receiving and sending APDUs).

The classes organization can be illustrated in a tree structure with the `java.lang.Object` class as its root. This architecture allows to define subclasses (child nodes) *extending* existing superclasses (parent nodes). In addition, a class may *implement* one or several interfaces, which is the only way to multiple inheritance in the Java language.

The classes defined by the JCAPI (if not specified as optional) are implemented on a Java Card platform, together with the JCVM, either in native or Java code, or even both,

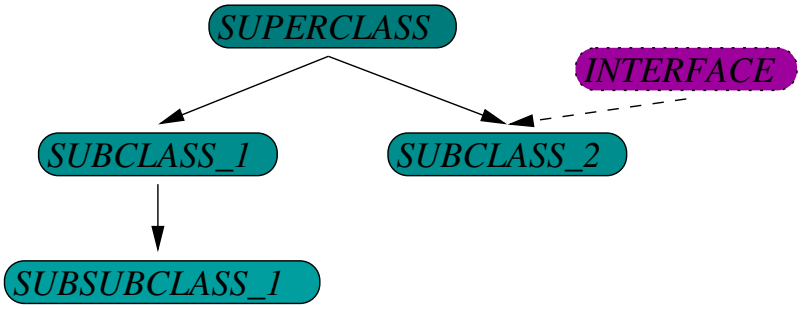


Figure 3.3: The tree architecture of Java classes.

depending on the card vendor and to a certain extent to the class itself. The role of the JCAPI with regards to the security of the system is detailed in Section 3.2.

3.1.3.3 Java Card Applications

The language in which the Java Card applications are written is a subset of Java, thus an object-oriented language. Therefore an application consists in one or more *packages* containing *interfaces* and *classes*. These classes and interfaces define fields and methods that are either statically linked to the class or interface itself (using the keyword `static`), or dynamically linked to object instances (by default). A field can be of one of the basic types supported by the Java Card language, such as `boolean`, `byte`, `short`, `int`, `char`, `long` (depending on the JCVM version and options), or a reference to an instance of a given class. A method is declared by

- a signature specifying :
  - its name,
  - the type of its parameters,
  - the type of the returned value,
- the sequence of instructions to execute (the bytecode array, once compiled).

Listing 3.1 depicts a classic Java Card applet, while Listing 3.2 depicts a connected Java Card servlet (Listing 3.2).

3.1.4 Developing and Executing Java Card Applications

It was mentioned at the beginning of this chapter that the key factor for the success of the Java Card technology has been the outstanding reduction of both the applications' development and deployment costs. This section describes the different steps of the development of an application and of its execution on the platform.

Listing 3.1: A sample Java Card classic applet

```
package mypackage;
import javacard.framework.*;

public class Wallet extends Applet {
  private static byte CLA_WALLET  = (byte) 0x20;
  private static byte INS_CREDIT  = (byte) 0x10;
  private static byte INS_DEBIT   = (byte) 0x14;
  private static byte INS_HIST    = (byte) 0x18;

  public static void install(byte[] bArr, short bOff, byte bLen) {
    // ...
  }

  protected Wallet() {
    register();
    // ...
  }

  public void process(APDU apdu) {
    if (selectingApplet())
      return;

    byte[] buf = apdu.getBuffer();
    if (buf[ISO7816.OFFSET_CLA] == CLA_WALLET) {
      switch (buf[ISO7816.OFFSET_INS]) {
        case INS_CREDIT:
          // ...
        case INS_DEBIT:
          // ...
        case INS_HIST:
          // ...
        default:
          ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
      }
    } else {
      ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    }
  }
}
```

3.1.4.1 The Development Process

As for the Java Card technology, the development process of an applet splits into three different steps:

1. The first step is obviously that of writing the source code of the application in the Java Card language, producing .JAVA files.

2. These .JAVA source files are then compiled by a standard Java compiler. This operation produces the .CLASS files, binary representations of the .JAVA files, following a strict format specification given in [LY99] which can be executed by any Java Virtual

Listing 3.2: A sample Java Card connected servlet

```
package mypackage;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class WebWallet extends HttpServlet {
  private static final String PARAM_OP  = "op";
  private static final String PARAM_VAL = "value";
  private static final String CREDIT    = "credit";
  private static final String DEBIT     = "debit";
  private static final String HIST      = "history";

  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
              throws ServletException, IOException {
    String op = req.getParameter(PARAM_OP);
    if (op == null)
      throw new ServletException("Operation parameter not supported.");

    if (op.equalsIgnoreCase(CREDIT)) {
      // ...
    }
    else if (op.equalsIgnoreCase(DEBIT)) {
      // ...
    }
    else if (op.equalsIgnoreCase(HIST)) {
      // ...
    }
    else {
      throw new ServletException("Unknown operation.");
    }
  }

  public void init(ServletConfig config) throws ServletException {
    // ...
  }

  public void destroy() {
    // ...
  }
}
```

Machine. At this point, the development process is not different from that of standard Java. The next steps are specific to the Java Card Classic technology (*cf.* Figure 3.4). On Java Card Connected Edition platforms, the .CLASS files are packed in a Java ARchive file (.JAR file) together with the archive file's manifest (named *manifest.mf*) and .XML deployment descriptor files (named *javacard.xml* and *web.xml*) before being sent to the card (*cf.* Figure 3.5).

3. The so-called converter transforms the .CLASS files containing the just compiled application into a .CAP file, for Converted APplet. This *converter* is generally referred to in the literature as the off-card part of the JCVM because the link resolution it

operates is operated by the VM on standard Java platforms. The .CAP file's structure is specified by the Java Card Virtual Machine specification [Sun09f]. It consists in different *components* relative to specific information, some contained in the .CLASS file, other added by the converter. For instance, the *method component* contains the information relative to the methods declared in the source code (the code itself, the number of arguments, of local variables, ...). In order to link with classes defined outside of the current package, the converter must be given in parameter the list of *export* files (.EXP) to use, typically the directory containing the export files of the JCAPI. Similarly, the converter can output an export file for the converted package, to allow another application to use it, if necessary.

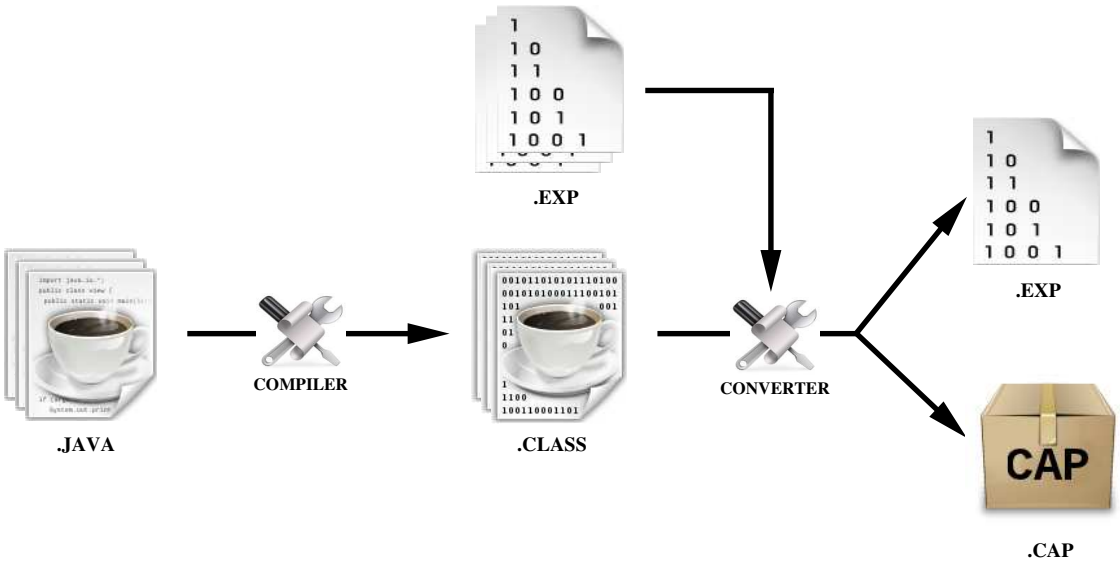


Figure 3.4: The application development process for Java Card *Classic* Editions.

The .CAP files can be downloaded into the smart card, with a possible step of bytecode verification, either off-card (*i.e.* before the loading process) or on-card (*i.e.* as part of the loading process). This verification step is one of the security features of the Java Card technology. It will be described in Section 3.2 which exposes the Java Card security mechanisms. Once the .CAP files are loaded, the applet they contain can be installed and registered on the platform. They are then ready to be executed. The loading and installing phases are described in Section 3.3.1.2.

3.1.4.2 Execution of an application on a Java Card 3 *Connected Edition* Platform

The operations leading to the execution of an application depend on the application type: an applet or a web application. In both cases, the entity in charge of the life cycle of the application is called a *container* (respectively applet container and web container).

Classic and Extended Applets The difference between *classic* and *extended* applets on a Java Card 3 Connected Edition lies in the possibility of using the various novelties

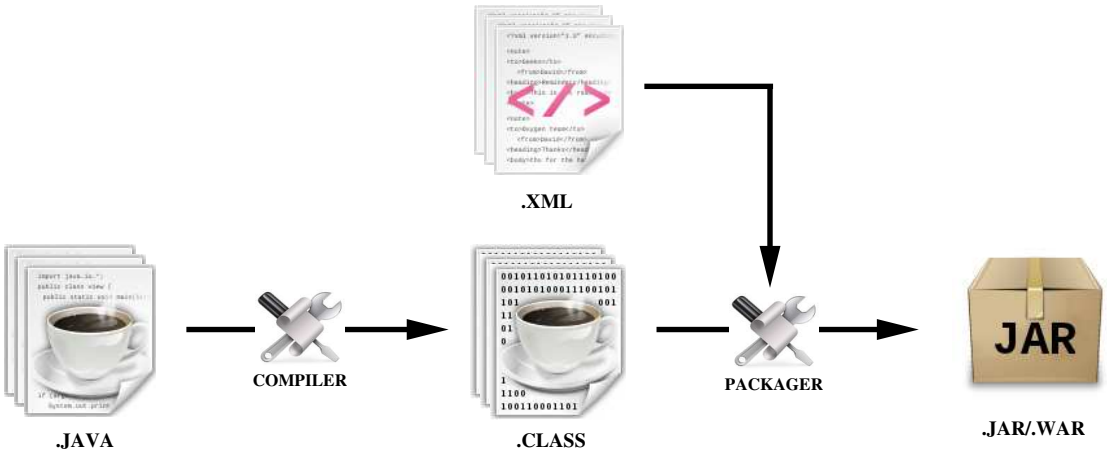


Figure 3.5: The application development process for Java Card *Connected* Editions.

of the Connected specifications. However, in both cases, as the card may host more than one application, an applet must be *selected* before being executed. This is done by sending a particular APDU command (Section 1.3.1.2), called the SELECT command, specified in [ISO98] with the DATA field set to the Application IDentifier (AID) of the application, which is defined during the application development process. This first step is necessary to identify the application the subsequent APDU command will be sent to. After this selection, all APDU commands received by the the card will be dispatched to the `Applet.process(APDU apdu)` method of the just selected application, which is part of the `Applet` interface of the JCAPI and must thus be implemented by every class implementing the `Applet` interface, i.e. by every applet.

Web Applications Similarly to the applet identification by an AID, web applications are identified on the platform by their *context path*. This context path defines the URL (Unified Resource Locator [BLMM94]) namespace of the application. Each request sent to the card will be dispatched by the web container to the application whose context path matches the request's URL. There is thus no explicit selection in this application model. The selection of an application is implicitly done by the container during the dispatching phase thanks to the URL of the incoming request, which higher part defines the concerned application.

The following section details the different security mechanisms coming with this technology.

3.2 The Security of Java Card Platforms

The Java Card technology provides several mechanisms enforcing the security of the system and allowing the hosted applications to provide security features which can be divided into two categories :

- the security mechanisms inherited from the standard Java specifications ;
- those specific to Java Card platforms.

3.2.1 Inherited Security Mechanisms

Java Card is a subset of Java. Consequently it inherits several security mechanisms from the Java language itself and from the standard Java specifications.

3.2.1.1 The Java Language: Access Modifiers and Type Safety

The main security characteristics of the Java language are the access modifiers associated to classes, fields and methods and the type safety enforced by the system.

Access Modifiers The Java language specifications defines a couple of keywords allowing to restrict the access to classes, fields and methods. These keywords, called *access modifiers*, are listed hereafter with their respective meanings:

- `private`: "can only be accessed from within the defining class".
- `package`: "can only be accessed from within the defining class or the package containing the defining class". It is the default modifier.
- `protected`: "can only be accessed form within the defining class, the package containing the defining class or subclasses of the defining class".
- `public`: "can be accessed from any class".

The access modifiers allow then to isolate certain data. These modifiers are enforced by both the compiler and the bytecode verification process described in Section 3.2.1.2.

In addition to these access modifiers, the keyword `final` allows to define classes that cannot be extended, methods that cannot be redefined by a subclass and fields that cannot be modified.

Type Safety Types are important on a Java Card platform from a functional point of view since they correspond to classes defining the behaviour of instantiated objects. But they are even more important when considering the security of the whole system. The type safety property is meant to ensure that each and every object instance complies to the contract defined by the class it is an instance of. For instance it ensures that a variable of a given type `A` is never used as if it were a variable of another type `B` that would not be compatible with type `A`. More particularly, it ensures that no C-like pointer arithmetic is used, since object instances and integral values are not compatible. It is therefore not possible to increment an object reference for instance.

The compliance of an application to the type safety property can partially be statically verified. This is a part of the aim of the bytecode verifier described hereafter. However, in certain circonstances, the correctness of a typecasting can only be decided at runtime.

Such a situation comes with the use of explicit typecasting in the application code for which the compiler generates *checkcast* instructions. The JCVM is then responsible for this decision, through the execution of the *checkcast* instruction. An example of a situation requiring an explicit typecasting involving an interface and a class implementing it is given in Listing 3.3.
Since method `methodNotDeclaredInInterface` is not visible considering the

Listing 3.3: An explicit typecasting

```
MyInterface mi = methodReturnsInterface();
mi.methodDeclaredInInterface();
// Explicit typecasting of the interface object into the implementing class
MyInterfaceImpl mii = (MyInterfaceImpl) mi;
mii.methodNotDeclaredInInterface();
```

`MyInterface` type, the explicit typecasting into the `MyInterfaceImpl` type is required to use this method.

3.2.1.2 The Bytecode Verification Process

The Java compiler produces `.CLASS` files that are compliant with the language rules defined in the previous section. However, the class file format as well as the instruction set being public, it is rather easy to modify a `.CLASS` file in order to produce a class that would not comply to these rules. The bytecode verifier is in charge of detecting such infringements. Such a situation does not necessarily betray the bad intention of the class developer since it can be due to a single class recompilation within a package after various modifications for instance.

On standard Java platforms, the bytecode verification process is executed by the virtual machine at linking time. Historically, this option was not viable on Java Card platforms because the verification process is very costly. The bytecode verifier was then executed off-card, as well as the converter executing the linking phase. The on-card execution of the bytecode verifier is still only optional on *Classic* platforms. The Java Card 3 *Connected Edition* is the only one for which on-card bytecode verification is mandatory, as this standards targets high-end devices with enhanced capacities and as the linking phase is also executed on-card, thanks to the dynamic class loading feature.

The bytecode verifier operates in several steps, which allow to recursively check that for each method defined in the class file and each possible branch of the data-flow :

- Local variables never contain an illegal value, with regards to the type safety property;
- There are sufficient values on the operand stack for each executed instruction (*i.e.* there is no stack underflow);
- There is sufficient room on the operand stack when a new value is pushed onto it (*i.e.* there is no stack overflow);

- The destination of control transfer instruction is valid, *i.e.* it is in the current method's body and points to an instruction.

Objects initialization and exception handlers are treated separately with specific verifications.

An implementation of the bytecode verifier is provided with the Java Card Development Kit [Ora]. However, it is likely that every actor of the Java Card industry has developed its own bytecode verifier for internal use and *a fortiori* for embedded verification. These implementations can use the various approaches that have been mentioned in the literature (such as data flow analysis, model checking or proof-carrying code) [Nec97, Ler01, Ler02, RR98, BD04, CBR02, MNTT02].

3.2.2 The Need for Specific Security Features

In addition to the inherited security mechanisms several security features have been added to the Java Card technology to meet all the requirements of a secure multi-application embedded platform. These additional security mechanisms mainly come from the constraints imposed in the context of smart cards such as the impossibility to run one instance of the JCVM for each application or the possible loss of power supply on card-tears for instance.

3.2.2.1 Application Isolation

On standard Java platforms, whenever a Java application is launched, a particular instance of the JVM is executed and exits together with the application. To ensure the isolation between the various applications, each instance of the JVM is executed in a so-called *sandbox*. However the constraints existing on Java Card platforms prevent such a behaviour. A single instance of the JCVM runs on the card and never exits. This unique JCVM instance is then responsible for interpreting the multiple applications loaded on the card. Consequently it is also responsible for ensuring the data and code isolation between these various applications.

Application Data Isolation: the Application Firewall This application isolation relies on the concept of context and is enforced by the so-called *application firewall*. At load time, each *application group* (*i.e.* Java package) is assigned a context[2]. Each object subsequently created within an application group is said to be *owned* by the application group and inherits its context. In addition, the JCRE itself is also assigned a specific context. The application firewall then forbids any access between different contexts that would not be explicitly authorized, except if the accessor context is that of the JCRE.

These explicit authorizations are granted through the *sharing mechanism*. The sharing mechanism relies on a particular interface of the API: `javacard.framework.Shareable`. By implementing this interface, a class indicates its intention to be shared and allows its instance (called SIO, for *Shareable Interface*

[2]One can note that this principle is similar to that enforced on early IBM/360 computers [**?**]

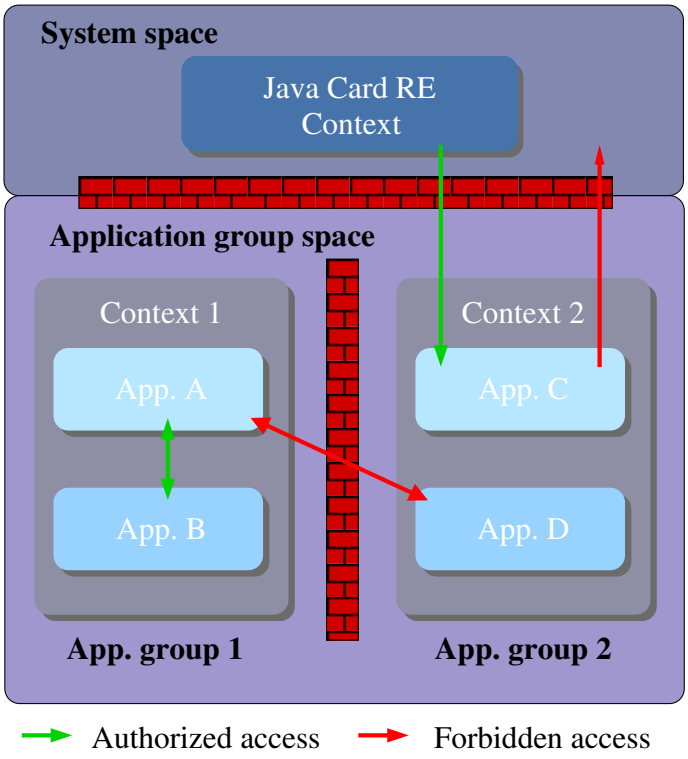


Figure 3.6: Application data isolation with the application firewall.

Object) to pass through the application firewall.

The mechanisms allowing an application to access a shared interface are different in the *Classic* and *Connected Editions* of the specifications. Both these mechanisms are illustrated in Figures 3.7 and 3.8. In the *Classic Edition*, access across the application firewall is operated as follows (with references to the different steps described in Figure 3.7) :

- An applet `B` exposes a shareable interface `SI` (1) and contains a class `O` implementing this interface (2).
- The client applet `A` also contains the interface `SI` (3) and sends a request to access a SIO by calling the `JCSystem.getShareableInterfaceObject(AID serverAID, byte parameter)` static method, with `serverAID` the AID of the applet `B` exposing the SIO and `parameter`, optional data (4).
- The JCRE forwards this request to applet `B` by calling its `Applet.getShareableInterfaceOBject(AID clientAID, byte parameter)` virtual method, with `clientAID` the AID of the applet demanding access to the SIO and `parameter`, the parameter byte used in the previous step (5).
- Applet `B` decides whether it accepts or not to share the requested SIO with applet `A` and returns either the SIO or the `null` object (6).

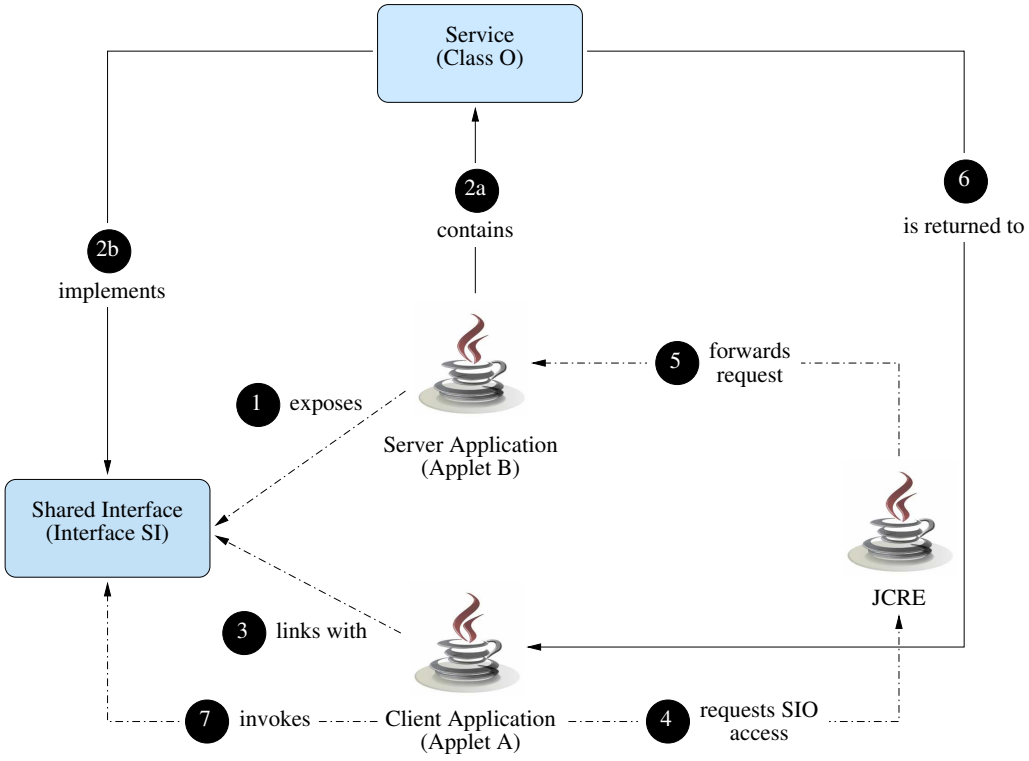


Figure 3.7: The sharing mechanism on Classic platforms.

- Applet `A` receives the SIO as of type `Shareable`. It must then cast it into the shared interface type `SI` in order to call the methods defined by this interface (7). Note, that even if the actual class of the SIO provides additional methods, applet `A` does not have the visibility on these methods (*i.e.* it cannot link with these methods) and can therefore not use them.

In addition to the legacy sharing mechanism of the *Classic Edition*, the *Connected Edition* offers a sharing mechanism based on a so-called *service registry* owned by the JCRE. Access across the operation firewall through this service registry is operated as described hereafter (with references to the different steps described in Figure 3.8) :

- An application `B` contains a class `O` implementing an interface `SI` extending the interface `Shareable`.
- In order to expose the shared service, application `B` registers a *service factory* (i.e. a class providing a create method returning instance(s) of class `O`) under a given URL with a particular scheme, `sio:` and within its own namespace (1).
- The client applet `A` also contains the definition of interface `SI` and sends a request to access the exposed service by calling the `ServiceRegistry.lookup(String serverURI, String serviceURI, Object parameter)` virtual method on the service registry instance obtained by a call to the `ServiceRegistry.getServiceRegistry()` static method, and with parameters `serverURI`, the application URI of the server of the service to lookup,

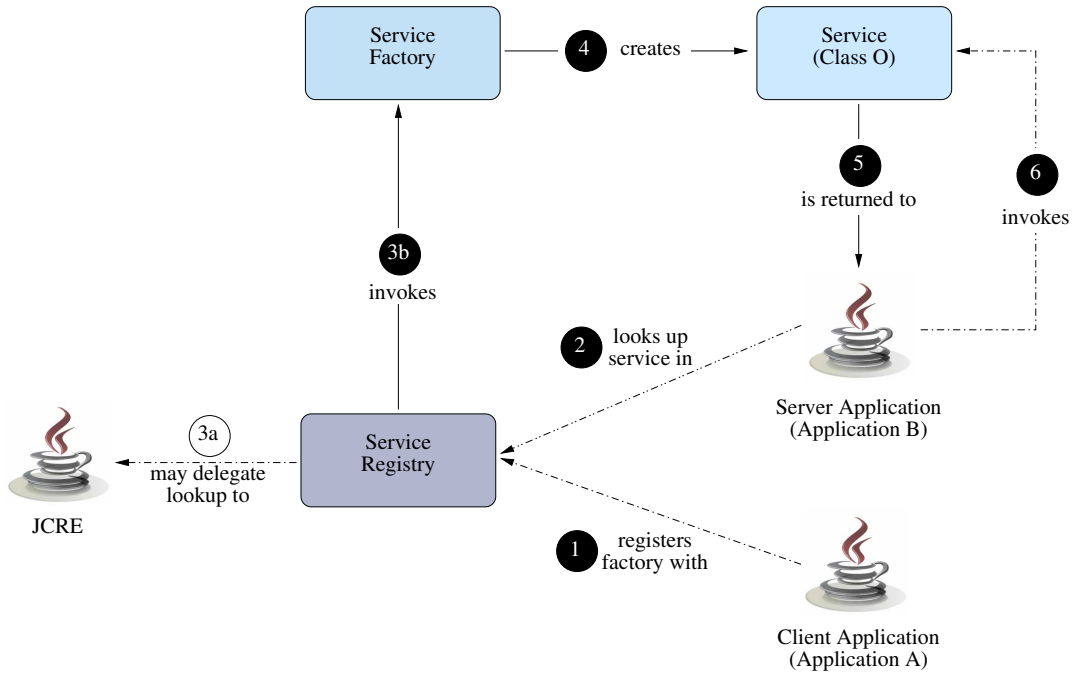


Figure 3.8: The sharing mechanism on Connected platforms.

`serviceURI`, the URI of the service to lookup and `parameter`, an optional parameter (2).

- The JCRE forwards this request to the service factory provided by `B` and associated with the given server and service URIs (3).
- Application `B` decides whether it accepts or not to share the requested SIO with application `A` and returns either the SIO or the `null` object (4).
- Application `A` receives the SIO as of type `Shareable` (5). It must then cast it into the shared interface type `SI` in order to call the methods defined by this interface (6). Similarly, even if the actual class of the SIO provides additional methods, application `A` does not have the visibility on these methods and can therefore not use them.

In addition the ability for an application to register, unregister or lookup services is subject to permission checking (see Section 3.2.2.3).

Finally, we can denote a exceptions to the application isolation mechanism concerning *global arrays*, static fields and methods. As defined in the JCRE specifications, global arrays are owned by the JCRE itself but are likely to be accessed by any application. The typical example of such arrays is the byte array contained in the `APDU` object instance, referred to as the APDU buffer. Similarly, classes' static fields and methods are not subject to firewall checks. They can therefore be accessed by another application, provided that application can link with this class. Using a static field in an application can therefore allow an "easy" sharing mechanisms. On the other hand developers should be conscious that it is also a serious security weakness since any application is likely to access it. The use

of static fields to store sensitive data is therefore generally proscribed with regards to the attacks described in Chapter 4.

Application Code Isolation On *Classic Edition* platforms, the code isolation is implicit since the linking phase is executed off-card by the Java Card converter. The simplest way to ensure that an application does not use another application's code is to keep this application's export file confidential.

On the other hand, *Connected Edition* platforms support dynamic class loading. The JCRE specification mandates that the code isolation be implemented through a class loader delegation hierarchy. This hierarchy ensures that:

- The classes of an application group are only visible to that application group.
- Classic library classes are only visible to classic applet applications.
- Extension library classes are only visible to extended applet and web applications.
- Shareable interfaces are visible to all applications.
- JCRE classes are visible to all applications and libraries.

When an application requests access to a specific class, the JCRE then asks this class to the different class loaders, going from the application group class loader to the bootstrap class loader. If none of the class loader can load this class, a `ClassNotFoundException` is thrown by the system. Regarding the numerous vulnerabilities of standard Java platforms based on user-defined class loader [DFW96], it is important to highlight the absence of such a facility on Java Card platforms.

3.2.2.2 Transactions and Atomicity

The atomicity of an operations is the fact that an operation is either completed or not performed at all. This principle must be enforced to avoid a situation in which the state of the card would be unknown, somewhere between the beginning and the end of an operation. In order to enforce this property, the Java Card specifications describes the notion of *transaction*. A transaction stands for a block of operations for which the atomicity is required. The transaction mechanism is based on three methods provided by the `JCSystem` class of the JCAPI:

- `beginTransaction()`, starting a transaction block;
- `commitTransaction()`, ending a completed transaction block;
- `abortTransaction()`, breaking a transaction block and unrolling it.

This mechanism had once lead to a security breach which is described in Section 4.2.2.1.

3.2.2.3 Permission- and Role-Based Security Policy

Another novelty of the Java Card 3 *Connected Edition* specifications is the introduction of permission- and role-based security policies. These mechanisms allow to restrict access to certain facilities offered by the platform or certain services offered by applications based on a given security policy. The access decision is based either on:

- a set of permission which an application or an on-card client is explicitly granted or denied based on a *protection domain* (set of permissions) defined by both the platform security policy and the card management security policy. These permissions are checked by the platform's access controller: `AccessController.check(Permission p)`.
- or a role for which the user or a client-application is authenticated. The authentication may be required declaratively within the deployment descriptor of a web application or programmatically by calling the `isUserInRole` method of either the `JCSystem` class or `HttpSerlvetRequest` interface depending on the requested data.

The security features presented in this section allow the Java Card to be considered as a secure platform. However, Chapter 4 will show that these mechanisms have been challenged in many occasions. To ensure a good level of security for both card issuers and final card holders, several certification schemes are followed in the smart card field. The following introduces the specificities of Java Card platforms in the certification context.

3.2.3 The Java Card Protection Profiles

In the framework of *Common Criteria for IT Security Evaluation*[3] certification, the company *Trusted Logic* has prepared, on behalf of *Sun Microsystems Inc.*, a document specifying a set of *protection profiles* for Java Card platforms: the *Java Card Protection Profile Collection* [Tru06]. A protection profile is a document allowing to reduce the cost of a security evaluation by defining security objectives and requirements the platform, called Target Of Evaluation (TOE) in this context, should meet in order to reach a certain *Evaluation Assurance Level* (EAL).

The *Java Card Protection Profile Collection* defines four protection profiles, depending on the *configuration* of the TOE. The defined configurations are:

- *The Minimal Configuration*: a multi-application platform where it is not allowed to load application post-issuance (***i.e.*** **after the card has been issued to a card holder)**.
- *The Java Card System Standard 2.1.1 Configuration*: extends the previous configuration by adding security requirements relative to post-issuance application loading, provided the application has passed a ***trusted*** bytecode verifier. This includes the *loader* and *installer* entities, but neither the bytecode verifier, nor the deletion process nor the card management.

[3] Refer to `http://csrc.nist.gov/cc/` and `http://www.commoncriteriaportal.org/`

- *The Java Card System Standard 2.2 Configuration*: extends the previous configuration by adding requirements relative to the *Remote Method Invocation* mechanism, the logical channels management, applet and object deletion processes. Similarly to the previous configuration, the bytecode verifier and the card management are not considered as part of the TOE.
- *The Defensive Configuration*: extends the previous configuration by adding requirements relative to the on-card execution of the bytecode verifier. The card management is still not considered as part of the TOE.

These configurations are illustrated in Figure 3.9, taken from [Tru06].

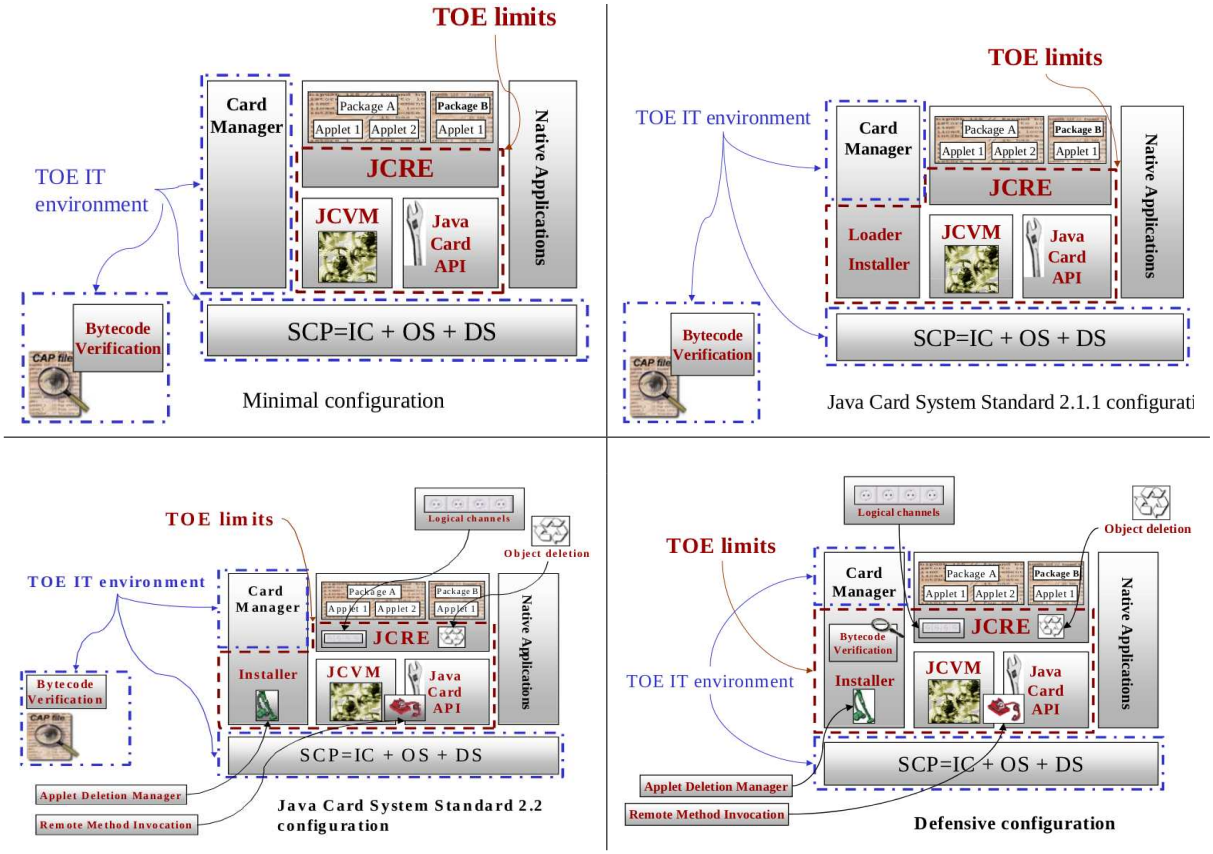


Figure 3.9: The four protection profiles of the *Java Card Protection Profile Collection* [Tru06]

The assets identified in the protection profiles and to which the specified security requirements protect from monopolization and/or either unauthorized disclosure and/or modification are:

- On-card applications and libraries' code;
- Sensitive data of the applications, such as the data contained in an object, a class static field, a local variable of a method, or a value in the operand stack;

- The PIN of any end-user;
- Any biometric template (for Java Cards 2.2.2 and later);
- The Java Card platform's code;
- The internal runtime data areas, such as the stack of frame and its content, the value representing the class of an object, the length of an array, any pointer used in internal structures;
- The runtime security data of the JCRE, such as the AIDs of installed applets, identification of the applet currently selected, the information used by the application firewall to identify an object's owner and the current context of execution;
- Sensitive data of the API, such as its private fields;
- Cryptographic keys, such as those used for application loading, or created by any on-card application;
- Cryptographic data used in cryptographic operations, such as a seed used to generate or derive a key.

Meeting the requirements described in the chosen protection profile against state-of-the-art threats demonstrates the security of a Java Card system. The considered threats are also described in the protection profile. In addition, the Joint Interpretation Library Working Group produces documents [Lib12] defining the kind of attacks to consider during security evaluation and aiming at unifying their respective quotation with regards to their potential consequences and the knowledge, equipments and facilities they require.

3.3 The Ecosystem of Java Cards

The leadership of the Java Card technology has naturally led to the development of a full ecosystem around it. In the following I introduce the two most important frameworks tightly connected with the Java Card technology, namely the GlobalPlatform framework and the (U)SIM/Card Application Toolkit.

3.3.1 GlobalPlatform

GlobalPlatform is a consortium to which participates the majority of the actors of the smart card industry in its widest sense (chip and card manufacturers, evaluation bodies, card issuers, but also actors of the payment and communication industry). The main goal of GlobalPlatform is to bring solutions to the different issues raised by multi-applications card, mainly:

- The downloading and installation of an application,
- The validation of a given application regarding the security of both the host platform and the other hosted applications,

- The identification and authentication of an on-card application, of the terminal with which it communicates and of the end user interacting with the application through the terminal.

The GlobalPlatform Specifications[4] define a standard for managing multi-applications smart cards which takes into account these issues. Indeed, there exists different specifications, dedicated to the various industries represented within the GlobalPlatform organization. In the framework of this dissertation, we are particularly interested in the *GlobalPlatform Card Specification Version 2.2.1* [Glo11a] which details the architecture of a GlobalPlatform card, the specified operations of card content management and the security features of the framework.

3.3.1.1 The GlobalPlatform Card Architecture

The GlobalPlatform card architecture aims at providing an interface allowing the management of several applications, potentially with various origins, that is independent of the underlying hardware and software system. However it is historically close to the Java Card standard and its architecture. The resemblance between them attests of such a fact (Figure 3.1 and 3.10).

As illustrated in Figure 3.10, GlobalPlatform defines several notions and actors, and establishes the responsibility of each actor. The following gives an outline of these notions and actors. See [Glo11a] for a thorough description and analysis of the different responsibilities.

Card Issuer. The *card issuer* is the entity owning the card and which is responsible for its behaviour. For instance, a bank is the card issuer of a card, a mobile phone operator is the card issuer of a (U)SIM card.

Application Provider. The *application provider* is the entity owning the application and which is responsible for its behaviour. For instance, the application provider of the NFC payment application on a (U)SIM card is the bank, although the card issuer is the mobile phone operator. We can note that this situation has been a obstacle to the deployment of the NFC since it raises several liability issues.

Security Domains. A *security domain* is an application providing secure services for the management of other applications, such as cryptographic services and key management. It stands as the representant of an off-card authority. There are three categories of security domains:

- The Issuer Security Domain (ISD) represents the card administrator, typically the card issuer. Its presence is mandatory.
- Supplementary security domains represent additional authorities, typically the application providers.

[4] Available at `http://www.globalplatform.org`

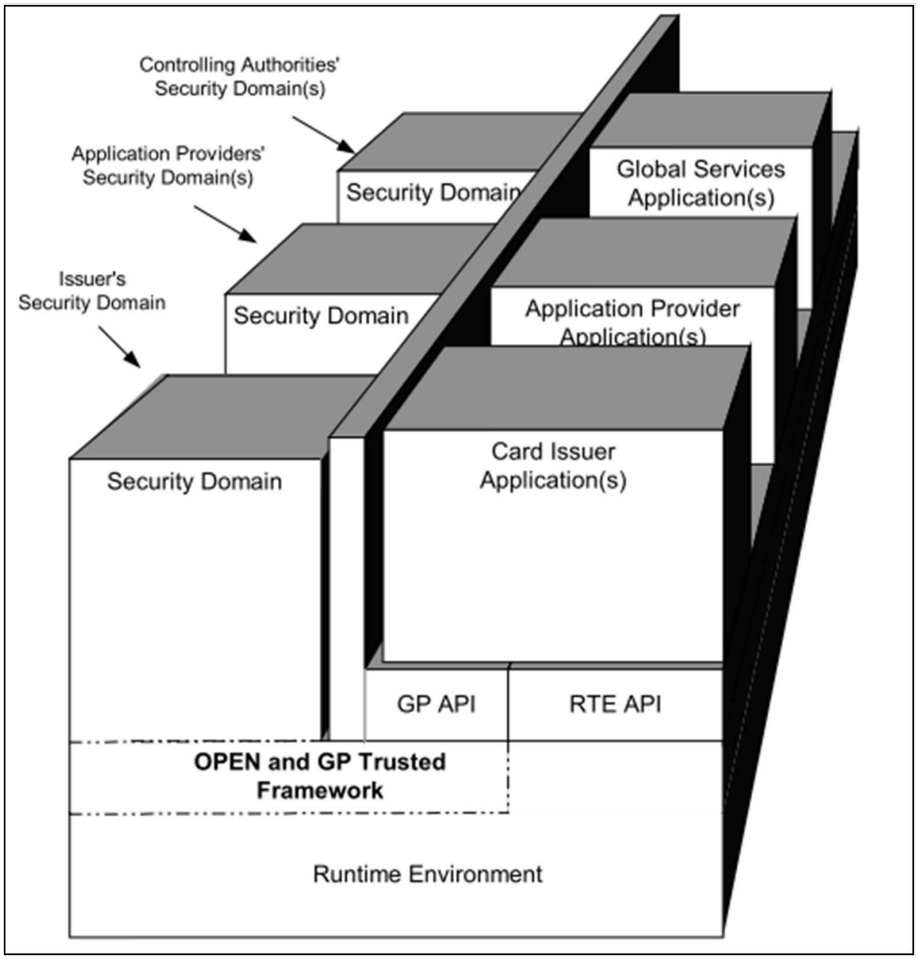


Figure 3.10: GlobalPlatform Card Architecture (Source: GlobalPlatform [Glo11a])

- Controlling authority security domains are particular types of supplementary security domains, hence also optional. Their aim is to enforce the defined security policy for all applications loaded on-card.

Global Services Application. Such applications are meant to provide services to all applications on-card. The typical example of a Global Services Application is an application providing Cardholder Verification Method (CVM) services. The GlobalPlatform specifications standardizes only one CVM: the global PIN.

Runtime Environment. The card runtime environment defined by the GlobalPlatform specification should provide a hardware-independent API, a mechanism ensuring the isolation of the different on-card applications and communication facilities with the off-card environment. The JCRE meets all these requirements.

Trusted Framework. A trusted framework is a part (or extension) of the runtime environment which is aimed at ensuring the communication between the different on-card applications. Regarding the JCRE as the GlobalPlatform runtime environment, this corresponds to the sharing mechanism described in Section 3.2.2.1.

GlobalPlatform Environment. The GlobalPlatform environment (OPEN, for Open Platform ENvironment[5]) is responsible for:

- Dispatching the incoming commands,
- The selection of an application,
- Managing *logical channels* which allows to communicate with several applications in parallel,
- The management of the card content,
- Exposing a dedicated API to on-card applications.

These different features may be, for a part, already proposed by the card runtime environment, which is the case with the JCRE. In this case, the OPEN should only complete the missing features or the features that are not compliant with the GlobalPlatform specification. Basically, the OPEN stands as an implementation of the different mechanisms stated above, based on a registry: the *GlobalPlatform Registry*. This registry stores data relative to the state of both the system and the loaded applications.

GlobalPlatform API. The GlobalPlatform API is meant to provide various services to the on-card applications such as card content management, cardholder verification or personalization for instance. The GlobalPlatform API for Java Card platforms is specified in [Glo11a].

Card Manager. The card manager is the central administrator of the card. It is not a single entity, but the reunion of the OPEN, the ISD and the CVM services.

3.3.1.2 Card Content Management

According to [Glo11a], card content management operation are the loading, installation, extradition (i.e. association with a different security domain), registry update and removal of Card Content. In order to let the card issuer delegate the management of a part of the card content to the application providers, GlobalPlatform provides it with means to:

- Let an application provider manage all card content,
- Let an application provider have full control over its own card content,
- Let an application provider isolate its own security domain(s) and application(s) from other application provider(s) and from the card issuer itself.

[5]The Open Platform is somehow the ancestor of GlobalPlatform.

In all cases, card content management operations are subject to authorization, based on the privileges of the different security domains.

The OPEN is responsible for the physical loading and installation of card content. In addition it is in charge of preventing a card content management operation depending on the card life cycle state and may prohibit concurrent card content management operations. The security policy applied during a card content management operation is enforced by the concerned security domain. This security policy must have been validated by the card issuer during the security domain installation.

An application provider security domain may be given the *DAP* (*Data Authentication Pattern) Verification* privilege. This privilege provides a service allowing to check for authenticity and integrity this application provider's application code, on behalf of the application provider's security domain, as part of their loading process. Also, the controlling authority's security domain may be given the *Mandated DAP Verification* privilege. Similarly, this provides a service allowing to check for authenticity and integrity all application code, on behalf of the controlling authority's security domain, as part of their loading process. The DAP Verification and Mandated DAP Verification processes consist in verifying the validity of a digital signature associated with the application code. The particular key and algorithm used are assumed to have been previously loaded and thus known by the concerned security domain.

The GlobalPlatform specification defines other security mechanisms such as the *Load File Data Block Hash* and the notion of *Token* relative to *Delegated* and *Authorized Management*. The reader is referred to [Glo11a, §C.2 and C.4] for more details on these notions.

3.3.1.3 Secure Communication

In order to ensure the security of communication between a GlobalPlatform card and a card reader, the specification defines the notion of *Secure Channel* and *Secure Channel Session*. In addition, it defines several *Secure Channel Protocols* (SCPs): SCP01 (deprecated), SCP02, SCP10 and SCP03 (specified in [Glo09]). This various protocols differ from the services they expose and the cryptographic algorithm they use (Triple DES, RSA, AES). The SCPs may be used to ensure:

- Mutual authentication of the off-card and on-card entities;
- Integrity of the data transmitted between the off-card and on-card entities and authentication of its origin;
- Confidentiality of the data transmitted between the off-card and on-card entities, including with regards to other on-card entities.

The *Security Domains* are responsible for supporting none, one or several SCPs.

Figure 3.11 gives an example of a secured communication where:

- The secure channel is initiated by the `INIT_UPDATE` and `EXTERNAL_AUTHENTICATE` commands which allow a mutual authentication of both the card and the terminal.

- The commands and responses are secured by both encryption and checksum computations ensuring respectively the confidentiality and the integrity of the transmitted data.

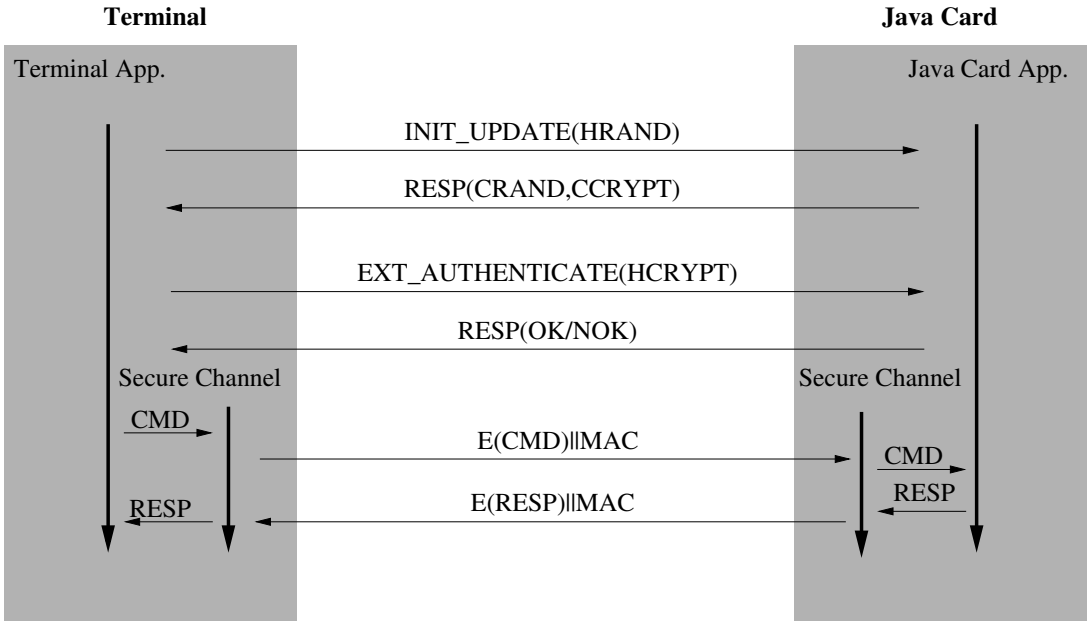


Figure 3.11: Example of secure communication through a Secure Channel.

As exposed in this section, the GlobalPlatform framework plays a key role in the Java Card ecosystem since it is widely used on the field to effectively deploy applications on-card. It is definitely a part of the Java Card technology's success and of its inherent security thanks to the secure application loading and communication protocols it provides. Chapter 4 explains the assumptions made in consequence in state-of-the-art attacks to circumvent the GlobalPlatform's security.

3.3.2 The (U)SIM/Card Application Toolkit

With the development of mobile communication emerged the need for a generic framework for *Network Access Application* (NAA) embedded within the *Integrated Circuit Card* (ICC), themselves integrated in the *Mobile Equipments* (MEs). The European Telecommunications Standards Institute (ETSI), together with the American Alliance for Telecommunications Industry Solutions (ATIS), the Japanese Association of Radio Industries and Businesses (ARIB) and Telecommunication Technology Committee (TTC)), the China

Communications Standards Association (CCSA) and the South-Korean Telecommunications Technology Association (TTA), within the framework of the 3^{rd} Generation Partnership Project (3GPP), have provided a great effort to standardize not only the underlying networks and protocols for a worldwide mobile communication, but also several *Application Toolkits* adapted to these networks, such as the Global System for Mobile Communication (GSM) and the Universal Mobile Telecommunication System (UMTS) for instance, and their respective NAA, such as the Subscriber Identification Module (SIM) application and the Universal Subscriber Identity Module (USIM) application.

3.3.2.1 The Standardized Application Toolkits

There are three standardization documents defining the different application toolkits:

- The Card Application Toolkit (CAT) [The11a], the most generic one;
- The SIM Application Toolkit (SAT, or more commonly STK) [The11e];
- The USIM Application Toolkit (USAT) [The11c].

These documents specify the interfaces between the ICC and the terminal (usually the ME), as well as terminal procedures that are compulsory. They define commands and protocol for the various networks and access technologies ensuring the interoperability between the ICC and the terminal regardless of both the respective manufacturers and the network operator.

3.3.2.2 The Standardized Java Card APIs

In order to integrate these toolkits in the Java Card ecosystem, the standardization bodies have also specified specific APIs:

- UICC API [The11b], relative to the CAT;
- (U)SIM API [The11d], relative to the (U)SAT.

These APIs provide an interface for the Java Card application developer to take advantage of the mechanisms implemented by the toolkit. In the case of the (U)SAT, Java Card applets using these mechanisms shall implement the `Toolkit` interface and are referred to as *Toolkit Applets*.

Chapter 4

State of the Art of Attacks against Java Cards

Contents

4.1 The Open Platform Assumption 53
4.2 Malicious Application based Attacks 54
4.2.1 Testing and Attacking Java Cards 54
4.2.2 Abusing Java Card Features 58
4.3 Ill-formed Application based Attacks 60
4.3.1 Unverified Applications 60
4.3.2 Type Confusion: Arrays and Objects 61
4.3.3 Attacks against Static Links in the Application 63
4.4 The Development of Combined Attacks and Mutant Applications . . 66
4.4.1 A Fault Attack against a Java Virtual Machine 66
4.4.2 Combined Attacks in Theory 67
4.4.3 Mutant Applications Detection 69

4.1 The Open Platform Assumption

In this chapter I introduce the state-of-the-art attacks against Java Cards. It is therefore necessary to set the general context of these attacks. Talking about software attacks, the main requirement is that an attacker is actually able of executing her own application on the system. The Java Card platform allowing post-issuance loading of applications, it is generally admitted that an attacker is likely to use this facility, although application loading on *on-the-field* cards is not that obvious, as exposed in Section 3.3.1.2. The second assumption concerns the execution of the bytecode verifier described in Section 3.2.1.2. From an attacker's point of view, two different assumptions regarding the verification process can be distinguished. They are listed below.

Hypothesis 1. *The application does not go through the off-card bytecode verification process, or the application goes through and passes an assumedly flawed bytecode verification process, either off-card or on-card.*

Hypothesis 2. *The application goes through and passes a sound bytecode verification process, either on-card or off-card.*

These assumptions have consequently lead to different kinds of attacks which are described in the following sections.

4.2 Malicious Application based Attacks

The first introduced attacks were based on the execution of malicious, but yet legal, applications loaded by the attacker. These works are therefore considering the most restrictive assumption concerning the execution of the bytecode verifier, *Hypothesis 2*. This kind of attacks can be subdivided into two categories:

1. Applications aiming at retrieving information on the platform itself and other hosted applications;
2. Applications attacking the platform, trying to circumvent the security mechanisms by misusing various features.

The following exposes these attacks and how they have been concealed by the specifications and secure coding guidelines.

4.2.1 Testing and Attacking Java Cards

I expose in this section the publications showing how one can use the multi-application support and the post-issuance loading facility of Java Card platforms to gather information on the attacked one and on the other applications on-card. In addition, such applications can try to discover weaknesses in the platform, or its environment.

4.2.1.1 Gaining Information on the System and Hosted Applications

First, the works published by Serge Chaumette *et al.* [CHS03, CS05] and Damien Sauveron's PhD thesis [Sau04] have demonstrated that if an attacker can load and install an applet on-card, then this applet would be able, from the inside, to:

- identify the available services, thus gaining information on the platform itself, but also on the other applications since they can only use existing services;
- collect information on the card in order to perform hardware attacks on the other applications.

To illustrate the first statement, we can consider the applet's `process` method described in Listing 4.1 exploring the cryptographic algorithms supported by the host platform.
This applet is totally legal, there is therefore no reason for a bytecode verifier to reject it. Yet, the information it provides the attacker with is valuable.

Listing 4.1: Fuzzing the cryptographic services

```
/**
 * This methods brute forces the Cipher.getInstance(byte algorithm,
 * boolean externalAccess) method in order to retrieve the supported
 * algorithms, stored in aSupported.
 * Each supported algorithm corresponds to a byte value specified
 * by the JCAPI.
 * For instance: ALG_DES_CBC_ISO9797_M1 corresponds to 0x2.
 */
public void process(APDU apdu) {
  // ...
  for (short algo = 0x0; algo <= 0xFF; algo++) {
    try {
      Cipher cip = Cipher.getInstance((byte) algo, false);
      aSupported[(byte) algo] = (byte) 0x1;
    }
    catch (CryptoException ce) {
      try {
        Cipher cip = Cipher.getInstance((byte) algo, true);
        aSupported[(byte) algo] = (byte) 0x2;
      }
      catch (CryptoException ce) {
        aSupported[(byte) algo] = (byte) 0x0;
      }
    }
  }
  // ...
}
```

Regarding the second statement above, we can also consider the work published in the framework of Dennis Vermoen's MSc thesis [Ver06a, VWG07] in which the authors propose a method based on *Power Analysis* to reverse engineer on-card applets. The exposed applet reverse engineering is actually based on two steps.

The first step is to associate a unique power consumption pattern (template) to the various JCVM instructions. This step is then achieved by applying statistical treatment to curves representing the power consumption of the card during the execution of a particular applet made to exhibit a given instruction, or sequence of instruction. The application loading facility is then crucial in this phase. For instance, the authors give the example of the applet described in Listing 4.2 to exhibit the bytecode sequences

```
sload, sload, sadd, s2b, sstore
```

and

```
sload, sload, smul, s2b, sstore
```

corresponding to the repeated Java code lines

```
p = (byte) (a+d);
```

and

```
p = (byte) (a*d);
```

Listing 4.2: Constructing power consumption patterns [VWG07]

```
public class TestApplet extends javacard.framework.Applet {
  public void process(javacard.framework.APDU apdu) {
    byte a = (byte) 0x04, d, p;
    byte buffer[] = apdu.getBuffer();
    short len = apdu.setIncomingAndReceive();
    d = buffer[(short) (javacard.framework.ISO7816.OFFSET_CDATA)];
    p = (byte) (a+d);
    p = (byte) (a*d);
    p = (byte) (a*d);
    p = (byte) (a+d);
    p = (byte) (a*d);
    p = (byte) (a+d);
    p = (byte) (a+d);
    p = (byte) (a*d);
  }
}
```

After several power consumption acquisitions, resampling and averaging, they obtain the following power consumption pattern where the different instructions can be identified (Figure 4.1).

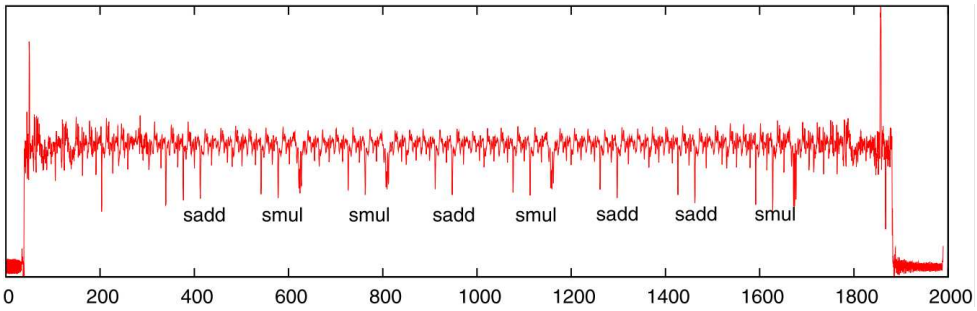


Figure 4.1: Power consumption of the card during execution of the applet List. 4.2 resampled and averaged over 10,000 samples [VWG07].

The second step consists in acquiring the power consumption during an unknown applet execution in order to recognize the different instruction templates built in the previous step. The pattern matching itself is done by computing the Pearson's correlation coefficient between the unknown instruction and the various built templates. The instruction corresponding to the template leading to the highest correlation coefficient can reasonably be considered as a good hypothesis on the actually executed instruction. Table 4.1 shows the results obtained with their technique.

The results exposed show uncertainties and a mismatch. However, by combining the information gained from the template matching with the strong entropy of the Java language, due to the fact that every sequence of bytecode instructions is not necessarily legal, they manage to enhance these results.

Expected	*Recognized*	*Alternatives*
sload	sload (93%)	aload (89%)
sload	sload (92%)	aload (91%) & sconst, sstore (57%)
sadd	sadd (91%)	sload (55%) & aload (51%)
s2b, sstore	s2b, sstore (91%)	sload (51%)
sload	sload (92%)	aload (78%) & sconst, sstore (54%)
sload	**aload (92%)**	**sload (91%)**
sadd	sadd (90%)	sload (54%) & aload (53%)
s2b, sstore	s2b, sstore (90%)	sload (53%)

Table 4.1: Results of the pattern matching phase on various instructions [VWG07].

Finally, the works published in the scope of the MESURE project [GPV06, CNA06] are also of interest since they allow to easily test a Java Card and get the timing of various API methods and VM instructions. The general philosophy of this benchmark tool is to compare the execution timing of a method executing the measured operation and a dummy method exactly similar to the first one, except for the measured operation. Again, these information can reveal very interesting from an attacker's point of view, although the aim of the project was definitely not to propose new attack techniques.

4.2.1.2 Searching for Weaknesses

An attacker can also load test applications that will search for potential weaknesses in certain security mechanisms of the platform.

Wojciech Mostowski *et al.* [MP07] provide a complete suite allowing to test several specific mechanisms of open Java Card systems, including, and particularly, the application firewall and related specified rules. Yet this work has not revealed severe flaws in the tested cards. This is not very surprising since similar tests are performed on all cards through the Java Card Test Compliance Kit (TCK), provided by Oracle, validating the conformance of a platform to the Java Card specifications before it is released. However the TCK is not public and is only available to Java Card licensees. This initiative is therefore still interesting.

The construction of a sound bytecode verifier has been widely studied, in particular through the use of formal methods []. However, Émilie Faugeron *et al.* have exposed a weakness in the implementation of a particular off-card verifier without revealing its origin [FV10]. This weakness lies on the order in which the different `case` blocks of a `switch` statement are verified and leads to an unverified sequence of bytecode instructions. In this context we can also outline the ongoing work of Aymerick Savary aiming at producing complete test suites for bytecode verifier based on a formal model of the programming language [Sav11].

4.2.2 Abusing Java Card Features

To a lesser extent, it has been shown that the implementation of certain functionalities of the Java Card platform can lead to security weaknesses. Such vulnerabilities are exposed in by Marc Witteman in [Wit03] and Wojciech Mostowski *et al.* in [MP08]. These vulnerabilities concern two different mechanisms: object sharing and the transaction feature.

4.2.2.1 Abusing the Sharing Mechanism

In [Wit03, MP08], the authors relate an exploit of a bug in the implementation of the sharing mechanism found on several platforms. This exploit is based on the use of two different versions of the shared interface in the server and client applets, which are potentially built and loaded separately, i.e. in different `.CAP` files and using different `.EXP` files for the linking phase.

For instance they exhibit the example of the two interfaces described respectively in Listings 4.3 and 4.4, differing only in the type of the argument of the `typeAttack` method.

Listing 4.3: The server's shared interface definition

```
public interface TypeAttackInterface extends Shareable {
  public void typeAttack(short[] buf);
}
```

Listing 4.4: The shared interface definition used on the client side

```
public interface TypeAttackInterface extends Shareable {
  public void typeAttack(byte[] buf);
}
```

Consequently, the server applet will assume the parameter of the `typeAttack` method to be of type `short[]` whereas the client will give a parameter of type `byte[]` to the `typeAttack` method. As a result, the server applet will treat an array of `byte` as if it were an array of `short`. Since byte and short values are respectively coded over 8 and 16 bits, the server may then read n bytes in the parameter array (assuming n is the length of this array) and n more bytes in the system memory. Note that the type confusion between the `byte[]` and `short[]` types described here is only one of the many options offered in terms of type confusions.

4.2.2.2 Abusing the Transaction Mechanism

Another functionality abuse is exposed in [MP08] concerning the transaction mechanism. This attack is based on an incorrect implementation of the roll-back operation occurring when a transaction is aborted, by a call to the `abortTransaction` method provided in the JCAPI. Indeed, amongst other tasks, this mechanism is supposed to deallocate any

object instance created during the aborted transaction and reset to `null` the references to these object instances.

The authors discovered that on some cards, this mechanism was not fully compliant with the specifications and did not reset to `null` local variables used to store objects instantiated during the transaction. In addition, they assume that references released during the roll-back are subsequently reused. They consider the code in Listing 4.5 which would create a type confusion between types `byte[]` and `short[]`, respectively through the instance field `arrayB` and the local variable `localArray`.

Listing 4.5: Transaction abuse

```
short[] arrayS; // instance field
byte[] arrayB; // instance field
// ...

  short[] localArray = null;     // local variable
  JCSystem.beginTransaction();
  arrayS = new short[1];
  localArray = arrayS;
  JCSystem.abortTransaction();

  // arrayB gets the same reference arrayS used to have
  arrayB = new byte[10];

  // this can be tested
  if((Object) arrayB == (Object) localArray)
    // this is true!

// ...
```

This code was subsequently executed on eight different cards among which only the half was immune to the attack. It is still worth noticing that both the JCRE and JCAPI specifications encourage programmers not to use the `abortTransaction` from within a transaction that creates new objects, allowing the JCRE to mute the card in such circonstances.

Hogenboom *et al.* also explore the possibilities offered by the very same type confusion in [HM09]. In this article, the authors claim that they manage to identify the internal representation of an array on a given card, crossing the bounds of a `byte[]` by accessing it as a `short[]`. This representation is illustrated in Figure 4.2.

They subsequently modify both the `<ARRAY_LENGTH>` and `<ARRAY_DATA_POINTER>` values, parsing all possible values. It turns out that not all `<ARRAY_DATA_POINTER>` values allowed their applet to read data, but they still manage to access 48 kilobytes of data in the system memory. After analysing the obtained data, they "*suspect*" the memory organization depicted in Table 4.2.

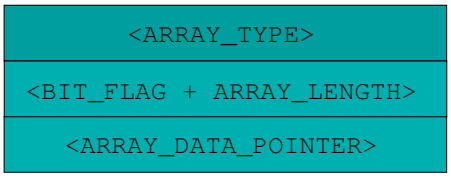


Figure 4.2: Observed internal representation of an array in the system memory.

Address range	(Suspected) Content
0020 – 0F20	System data
3017 – 3717	RAM, System data
4000 – FFFF	Applet data and code, CardManager data and code

Table 4.2: Suspected memory organization.

An important point to notice in this attack is that the application firewall implemented on the tested card has failed protecting the data outside of the attacking applet. This is probably due to the use of the internal data pointer which is apparently not considered by the application isolation mechanism on this card.

4.3 Ill-formed Application based Attacks

The second type of attacks I will describe is based on the execution of malicious *ill-formed* applications loaded by the attacker. Indeed the term *ill-formed* simply denotes that such applications do not respect the Java Card language rules and would consequently be rejected by any decent bytecode verifier. The works presented in this section are therefore considering the least restrictive assumption concerning the execution of the bytecode verifier, *Hypothesis 1*.

4.3.1 Unverified Applications

Although such applications may be produced by mistake, for instance by recompiling separately a given class within a given package, they are more likely produced by manipulation of the application's binary file (.CAP or .CLASS file). As previously stated, the Java Card language forbids the use of C-like pointer and, even more so, pointer arithmetic. Statements such as

```
Object obj = 0x1234;
```

or

```
obj++;
```

are therefore not accepted by the Java compiler. The question raised by the binary file manipulation is then: *how about carved bytecode instructions sequence?*

Indeed, it is rather easy to compile (and convert if need be) the following Java source code:

```
short addr = 0x1234; Object obj = null;
```

and to modify the resulting bytecode sequence in the output binary file:

```
sipush 0x1234; sstore_1; aconst_null; astore_2;
```

where:

`sipush 0x1234` pushes the `short` value 0x1234 onto the operand stack;

`sstore_1` stores the short value on top of the operand stack in local variable #1;

`aconst_null` pushes the `null` reference onto the operand stack;

`astore_2` stores the reference on top of the operand stack in local variable #1.

Into:

```
sipush 0x1234; astore_2; nop; nop;
```

where:

`nop` is the dummy instruction doing nothing.

Which corresponds to the forbidden Java source code:

```
Object obj = 0x1234;
```

The exposed binary file manipulation is very simple. But other manipulation can reveal much more complex, especially when dealing with values used in the linking phase or referring to offsets in various parts of the binary file. In order to make any manipulation as simple as the one described above, several tools have been developed within the Java (Card) security community, either publicly available or not [Sau01, Sau04, Sma, reJ, BCE, ASM, SER].

The following describes state-of-the-art attacks using ill-formed applications to question the security of Java Card platforms.

4.3.2 Type Confusion: Arrays and Objects

In [MP08], Mostowski *et al.* expose various type attacks against Java Cards (some of them taken from [Wit03]) and test their success against several cards.

The first of these attacks consist in a type confusion between a `byte` array and a `short` array, which has already been described in Section 4.2.2.2.

Secondly, they expose attacks involving several type confusions between integral value arrays and object instances. The first of these type confusions involves an instance of class `Fake` defined in Listing 4.6 and a `byte` array.

The underlying assumption of this attack is that instances of the `Fake` class and `byte` arrays have a similar representation in the system memory, such as illustrated in Figure 4.3.

Listing 4.6: Transaction abuse

```
public class Fake {
  short len = 0x7FFF;
}
```

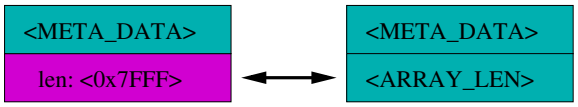


Figure 4.3: Assumed internal representation of `Fake` objects (left) and `byte[]` (right)

If this hypothesis is verified, then, once the type confusion has been created, the attacker would be able to access as much as 32,767 bytes (32 kilobytes) of data in the system memory. Yet, the attack failed on all of the tested cards.

The failure of the previous type confusion pushes the authors to conclude to another correspondance between the internal representation of object instances and arrays. They then build a second class, described in Listing 4.7, in order to create a type confusion between an instance of this class and a `short` array.

Listing 4.7: Transaction abuse

```
public class Test {
  Object r1 = new Object();
  short s1 = 10;
}
```

Once an instance of this class confused with a `short` array `a`, they obtain the following results when reading the array content:

- `a.length = 2`,
- `a[0] = 0x09E0`,
- `a[1] = 0x000A`.

They conclude that these values correspond to:

- the number of instance fields,
- the reference of object `r1` (*i.e.* the first instance field),
- the value of `s1` (*i.e.* the second instance field).

In particular, this allow them to forge (the authors use the term *spoof*) object references by writing in `a[0]`. However some cards they tested apparently perform runtime checks and prevent certain modifications or data access. They then explore the potential consequences of reference switching, and AID modification which may eventually lead to applet impersonation.

4.3.3 Attacks against Static Links in the Application

Another widely studied kind of attacks aims at targeting the linking phase when loading an applet on-card. Such attacks usually involve three components of an applet's .CAP file:

The method component which describes the methods declared in the package represented by the .CAP file;

The constant pool component which represents a list of all classes, methods and fields referenced in the *method component* of the .CAP file;

The reference location component which contains a list representing the mapping between items referenced in the *method component* and their corresponding entry in the *constant pool component*.

Konstantin Hyppönen describes in [Hyp03] an attack using static class field linkage. The author assumes that an on-card entity referred to as the *linker* performs the resolution of the method component's contant pool references to raw memory addresses when loading a .CAP file on-card. This operation is processes as described below:

- The linker parses the *reference location component* to read the offsets in the *method component* where a resolution is needed;
- For each offset listed, the linker reads the index of the reference to resolve in the *constant pool component* and resolve it.

The principle of the attack is then to remove an item from the *reference location component* in order to avoid the resolution of the corresponding reference in the bytecode array of the *method component*. As a consequence, the reference to the *constant pool component* in the bytecode array would then be considered as a raw memory address. The author then suggests that it would be possible to read any arbitrary memory address by applying this technique to the `getstatic_a` bytecode instruction which is specified in [Sun09g] as follows:

Format:

`getstatic_a`
`indexbyte1`
`indexbyte2`

Description:

The unsigned `indexbyte1` and `indexbyte2` are used to construct an index into the constant pool of the current package, where the value of the index is $(indexbyte1 << 8)|indexbyte2$. The constant pool item at the index must be of type `CONSTANT_StaticFieldref`, a reference to a static field. The item must resolve to a field of type reference. The item is resolved, determining the field offset. The item is resolved, determining the class field. The value of the class field is fetched. The value is pushed onto the operand stack.

By fixing the values indexbyte1 and indexbyte2 and removing the instruction offset in the *reference location component*, an attacker would consequently have access to the address $(indexbyte1 << 8)|indexbyte2$. The attack process is illustrated in Figure 4.4.

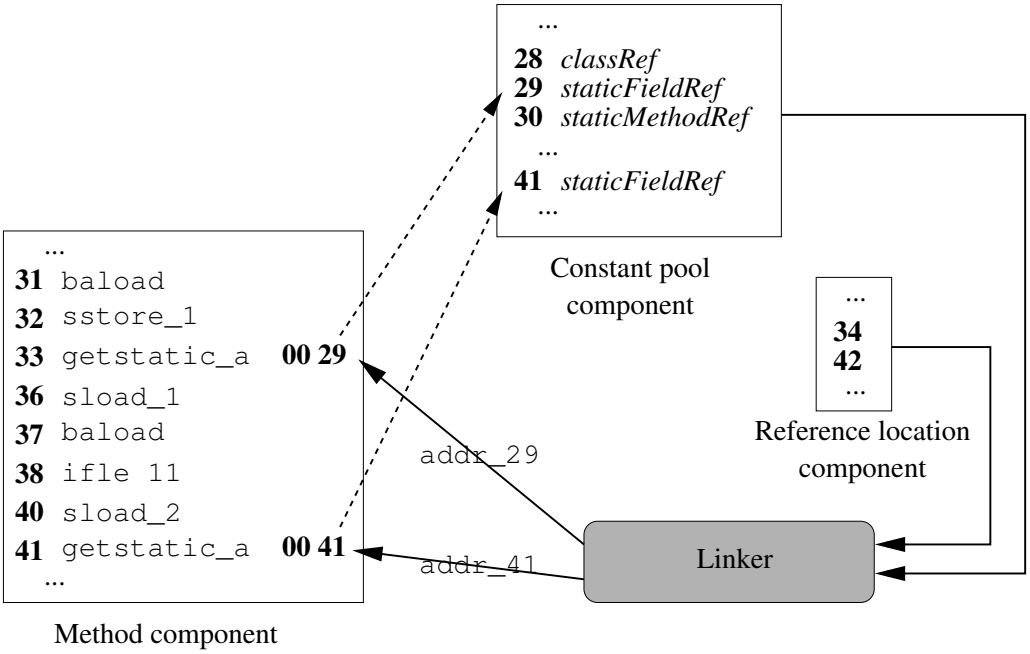


Figure 4.4: Operation of the linker on .CAP file components.

In [ICL10] Iguchi-Cartigny *et al.* describe an attack that is very similar in its conception but that may have even more serious consequences. They actually target the execution of an `invokestatic` instruction. Indeed, they prove that it was possible, under certain assumptions, mainly regarding the internal structure of on-card applets (*cf.* Figure 4.5) to search for the address of a `byte` array declared in an applet and to write its address as parameter to an `invokestatic` instruction in the bytecode array of a method.

Their attacking applet plays then the role of a Trojan horse, allowing the execution of any bytecode sequence that would be written in the `byte` array. Furthermore, since they can update the content of the array itself, they produce a kind of mutable code. This mutable code can for instance take advantage of the `getstatic_<t>` attack described above by modifying the parameters given to this instruction. The authors expose then that the Trojan is likely to search any code pattern in the system memory and to modify it, for instance replacing the call to the `OwnerPIN.check` method by `nop` instructions rendering thus the PIN useless. Listing 4.8 describes the Trojan applet.

In another article [BICL11], Guillaume Bouffard *et al.* show that another modification of the on-card code of an applet can give sensitive information to an attacker. This attack is based on the lack of runtime verification of the local variable indices given in parameter of certain bytecode instructions. Indeed, they show that using a local variable index greater than maximum number of local variables in a given *frame* of execution as the

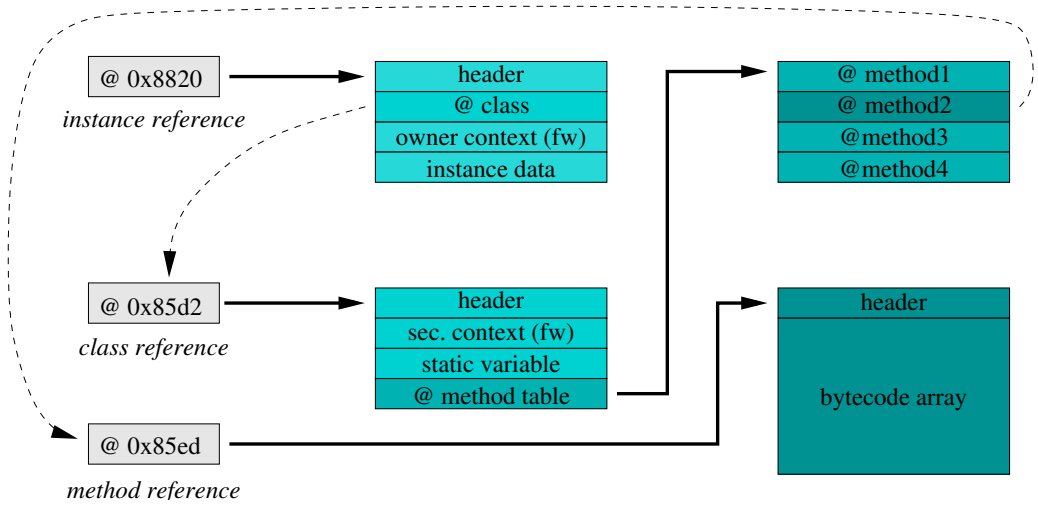


Figure 4.5: Assumed card internal structure.

Listing 4.8: Excerpt from the Trojan applet

```
public class EMANApplet extends Applet {

  byte[] codeD = {(byte) 0x01, (byte) 0x00, (byte) 0x7d, (byte) 0x00,
                  (byte) 0x00, (byte) 0x78};

  public static short functionToReplace(){
    return ad;
  }

  public void executeMyArray() {
    /**
     * Easy to detect instruction pattern to replace reference.
     */
    functionToReplace();
  }

  public void updateArray(byte[] code) {
    Util.arrayCopy(code, 0, code.length, myArray, 0);
  }

  public boolean searchAndReplace(byte[] pattern, byte[] newCode) {
    // ...
  }
}
```

parameter of a `ssload` instruction. The direct consequence of this attack is then the access to data stored in the frame of execution, in the case exposed, this allowed for instance to modify the return address redirecting the control flow at the end of the current method.

4.4 The Development of Combined Attacks and Mutant Applications

In order to circumvent the security brought by the bytecode verification process, the idea of combining fault injection techniques with software attacks emerged. Surprisingly, although fault attacks have been mainly studied in the embedded system context, the pioneering work in this field exposes a combined attack against a standard Java Virtual Machine executed on a Personal Computer. This seminal work has, since then, inspired several publications demonstrating the potential consequences of such attacks in the Java Card context. The first practical achievement of such attacks as well as several instances of this class of attacks are described in the following parts of this dissertation. The development of *Combined Attacks* has also lead to the introduction of the so-called *Mutant Application* detection problematic. The notion of Combined Attack is indeed at the center of this work.

4.4.1 A Fault Attack against a Java Virtual Machine

This attack is presented by Govindavajhala *et al.* in [GA03]. In this article, the authors describe how an error induced in the RAM allocated to the JVM execution can be exploited by an attacker to create a type confusion. The attack is based on the two Java classes defined in Listing 4.9.

Listing 4.9: The classes to confuse

```
public class A {
  A a1; A a2; B b; A a4; A a5; int i; A a7;
}

public class B {
  A a1; A a2; A a3; A a4; A a5; A a6; A a7;
}
```

The attack application can then allocate one object of type `A` and as many objects of type `B` as possible, with all fields of type `A` pointing to the unique `A` object instance. Then it only has to wait for a random error to flip at least one bit of some word (*i.e.* one of the `aX` fields) in one of the B objects to exploit a type confusion between types `A` and `B`. The attack process is depicted in Listing 4.10.

The type confusion can subsequently be exploited to read/write arbitrary memory address. This is done by setting the address in the field `i` of the `A` object (`r.i = address;`) and accessing the addressed memory through the same field in the field a6 of the B object (`q.a6.i`). Note that this is possible only if fields `i` of type `A` and `a6` of type `B` reside at equal offsets in their respective class structure. The type confusion is illustrated in Figure 4.6.

Listing 4.10: The attack application

```
public A a;
public B b000, b001, ... ;

public A r;
public B q;

public static void main(String[] args) {
  a = new A();
  b000 = new B(); b001 = new B(); ... ;

  while (checkFieldTypes());

  r = getErrFieldAInErrInstB();
  q = r.b;

  // Then q is considered as a type B instance.
  // Yet it is an instance of type A.
}

public boolean checkFieldTypes() {
  // returns false if a type error is detected,
  // true otherwise.
}
```

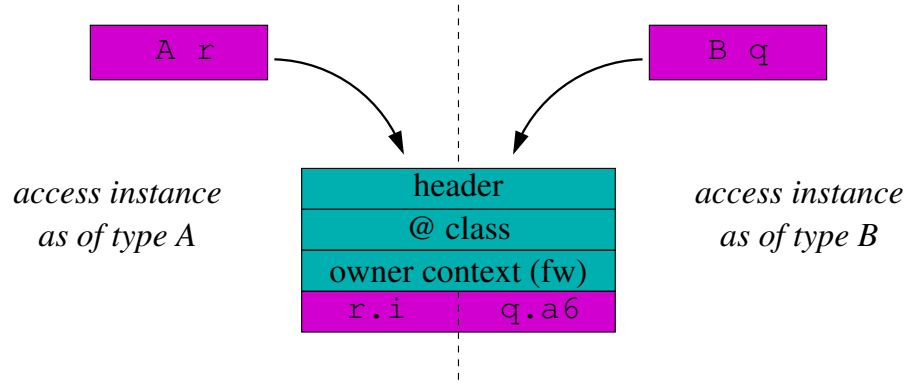


Figure 4.6: Field access in the confused object instance.

4.4.2 Combined Attacks in Theory

The possibility to combine hardware and software attacks against Java Card platforms has been evoked in various publications [Gad05, Ver06b, MP08, BICL11, BL12].

In [Ver06b], Olli Vertanen proposes different attack scenarii based on the use of power glitches to jump the execution of bytecode instruction(s) and provoke a type confusion. These are:

- Jumping (or zeroing) a sequence of instructions, thus forcing the store of an operand of a given type into a local variable of incompatible type;

- Jumping (or zeroing) a `checkcast` instruction supposed to control dynamically the correctness of a particular typecasting;
- Jumping (or zeroing) an instruction byte, thus forcing the execution of its parameter byte as a bytecode instruction.

Guillaume Bouffard *et al.* expose in [BICL11] that a fault injection allowing to stuck one chosen byte to zero may be used to redirect the control flow into an array of `byte`. For that matter, they consider the class defined in Listing 4.11, after having characterized the targeted card's memory management.

Listing 4.11: Attack on the `goto_w` instruction

```
public static byte[] codeD = // Located after the applet code
                             // according to memory mapping.
                           {(byte) 0x00, (byte) 0x00, (byte) 0x00,
                           ...,
                           (byte) 0x00, (byte) 0x00, (byte) 0x00,
                           (byte) 0x11, (byte) 0x17, (byte) 0x12,
                           (byte) 0x00, (byte) 0x00, (byte) 0x00,
                           (byte) 0x8d, (byte) 0x6f, (byte) 0xc0,
                           (byte) 0x00, (byte) 0x00, (byte) 0x00,
                           (byte) 0x00, (byte) 0x00, (byte) 0x00,
                           (byte) 0xfe, (byte) 0xdc, (byte) 0xba};

public void test(short n) {
  byte foo, bar;
  for (short i=0; i<n; i++) {
    foo = (byte) 0xba;
    bar = foo;  bar = foo;
    ...
    bar = foo;  bar = foo;
  } // At the end of the for loop, a goto_w instruction
    // is generated by the compiler to re-loop.
}
```

Since they know the memory mapping of data on-card, the authors can tune the size of the for loop so that a laser-induced perturbation sticking at `0` one byte of the offset given in parameter to the `goto_w` bytecode force a jump within the `byte` array `codeD`. Yet they do not provide much information on how they obtain the memory dump allowing them to learn the memory mapping.

Apart from the works presented in the following parts of this dissertation [Bar09, BTG10, Bar10, BHD11, BDH11, BT11, BHD12a, BGG12, BHD12b], the only publication presenting a practically achieved *Combined Attack* is that of Éric Vétillard *et al.* [VF10]. In this article, the authors present a *Combined Attack* and introduce the notion of *Mutant Applications*, meaning applications which behaviour will change once they have been loaded (possibly because of a fault injection modifying their bytecode instructions). Their attack is achieved in two steps and requires two fault injections. Yet the two fault injections do not have to be achieved in a raw thanks to the persistency of class fields. The first step of

their attack targets the method described in Listing 4.12.

Listing 4.12: Vétillard *et al.*'s attacked method

```
byte KEY_ARRAY_SIZE = (byte) 0x77;

Key getKey(short index) {
  if (index < KEY_ARRAY_SIZE)
    return keys[index];
  else
    return null;
}
```

The attack aims at disrupting the bytecode at offset 2 in the bytecode array associated to this method in order to set it to 0, corresponding to the `nop` instruction, and thus to modify the method return value, as exposed in Listing 4.13 and 4.14.

Listing 4.13: Original bytecode of the attacked method

```
sload_1                (1D)
bspush 0x77            (10 77)
if_scmpge +08          (6D 08)
getfield_a_this <1>    (AD 01)
sload_1                (1D)
aaload                 (24)
areturn                (77)
aconst_null            (01)
areturn                (77)
```

Listing 4.14: Interpreted bytecode after the attack

```
sload_1                (1D)
→ nop                  (00)
  areturn              (77)
```

By giving the reference of a `Key` object owned by another applet as `index` parameter to their `getKey` method, the authors can get a valid reference on this key. They subsequently manage to disrupt by a second fault injection the firewall check operated when they call the `Key.getKey` method of the API allowing to retrieve the actual key bytes of a `Key` object. As a matter of fact, they managed to disclose the cryptographic key used by another applet on the platform, which is obviously compromising the system.

4.4.3 Mutant Applications Detection

The detection of such mutant applications is a field that has been mainly tackled in the scope of Ahmadou Séré's PhD. thesis [Ś10]. As pointed out in [SICL09] and [VF10], the first countermeasure against mutant applications is either to modify the value associated to the `nop` instruction in order to make 0 an impossible bytecode or at least to double-check the read bytecode when it turns out to be 0. The following describes the other detection methods exposed in [SICL09, SLIC10, SLIC11]: the field-of-bit method, the basic-block method and the path-check method.

Bytecode Instructions	Bytecode Values	Field of Bit
`sload_1`	`0x1D`	X
`bspush 0x77`	`0x10 0x77`	X R
`if_scmpge +08`	`0x6D 0x08`	X R
`getfield_a_this <1>`	`0xAD 0x01`	X R
`sload_1`	`0x1D`	X
`aaload`	`0x24`	X
`areturn`	`0x77`	X
`aconst_null`	`0x01`	X
`areturn`	`0x77`	X

Table 4.3: Association between a bytecode array and a field of bit.

4.4.3.1 The Field-of-Bit Method

This first method [SICL09, Section 6.3] is based on the fact that the modification of a byte in a method's bytecode array is likely to modify the *nature* of the different bytes: an instruction byte becoming a parameter one and *vice-versa*. The authors propose then to add a *custom component* in the concerned package's `.CAP` file (as authorized by the JCVM specification [Sun09g]) keeping track of the *nature* of each byte.

The method consists in statically building the *custom component* where each method defined in the *method component* is associated a field of bit associating to each byte a bit-value depending on the *nature* of the byte: either executable (X) or readable (R), as depicted in Table 4.3.

At runtime, the JCVM is then responsible for checking that each interpreted bytecode is consistent with the associated bit. The principal drawback of this method is then that it will not detect a fault if it does not modify the *nature* of a byte, such as turning an `sload_1` into an `aaload` for instance.

4.4.3.2 The Basic-Block Method

The second proposed method [SICL09, Section 6.4] use the notion of *basic blocks*.

Definition 1. A **basic block** is a sequence of instructions which starts at a single entry point and ends at a single exit point.

The execution of a basic block can therefore only start at its entry point and end at its exit point. The point is that, at runtime, each basic block will be either fully executed or not executed at all.

The method consists then in statically determining the basic blocks composing each method defined in the *method component* of the package's `.CAP` file and computing a checksum for each basic block. The authors propose to use a simple XOR as checksum operation. They subsequently add a *custom component* made of a table containing for each basic block:

- The offset of the entry point in the method's bytecode array,
- The offset of the exit point in the method's bytecode array,

- The value of the XOR checksum.

At runtime, the JCVM is then responsible for dynamically computing the basic blocks, updating incrementally the XOR checksum and checking the coherency with the stored entry point, exit point and checksum. That is to say, it should check:

- When a basic block is entered, if the entry point is known (*i.e.* is in the table);
- When a basic block is leaved, if the exit point is known and matches the last seen entry point and if the checksum is correct.

Although this method leads to a relatively high computing overhead, it would detect any single-byte error in the bytecode array, unlike the previous one.

4.4.3.3 The Path-Check Method

Finally, the last proposed method [SLIC10] somehow enhances the previous *basic-block* method. This method also uses the notion of basic blocks but integrates them in the Control Flow Graph (CFG) representing the execution flow of a given method. For instance, the execution flow of the method `debit` defined in Listing 4.15 (from [SÍ10, §A.1]) and which basic blocks are given in Listing 4.16 (from [SÍ10, §A.2]) with its bytecode array can be represented by the CFG in Figure 4.7 where each vertex correspond to a basic block and each oriented edge to a jump from one basic block to another.

Listing 4.15: A sample debit method in a Java Card e-Wallet applet

```
private void debit(APDU apdu) {
  // access authentication
  if (pin.isValidated()) {
    byte[] buffer = apdu.getBuffer();
    byte numBytes = (byte) (buffer[ISO7816.OFFSET_LC]);
    byte byteRead = (byte) (apdu.setIncomingAndReceive());
    if ( (numBytes != 1) || (byteRead != 1) )
      ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);
    // get debit amount
    byte debitAmount = buffer[ISO7816.OFFSET_CDATA];
    // check debit amount
    if ( (debitAmount > MAX_TRANSACTION_AMOUNT) || (debitAmount < 0) )
      ISOException.throwIt(SW_INVALID_TRANSACTION_AMOUNT);
    // check the new balance
    if ( (short) (balance - debitAmount) < (short) 0)
      ISOException.throwIt(SW_NEGATIVE_BALANCE);
    balance = (short) (balance - debitAmount);
  }
  else {
    ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
  }
} // end of debit method
```

The method proposed consists then in statically computing and encoding the valid paths in the CFG with the following convention:

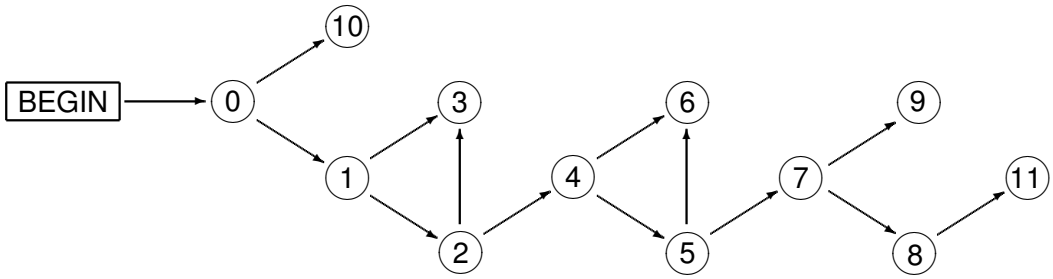


Figure 4.7: The `debit` method's control flow graph.

- The tag 01 denotes the beginning of a path;
- An edge is denoted 0 if it links a basic block ending by a branch instruction to another basic block;
- An edge is denoted 1 else (*i.e.* the entry point of the next basic block directly follow the exit point of the current basic block).

Note that the length of the tag depends on the logarithm of the number of vertices to link to. With regards to the same debit method, this leads to the tagged CFG depicted in Figure 4.8.

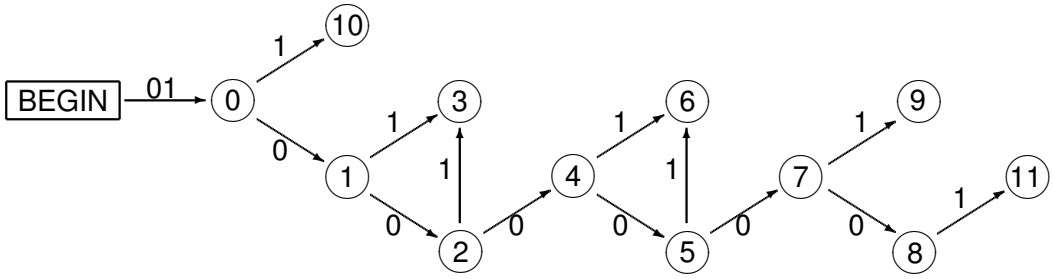


Figure 4.8: The `debit` method's tagged control flow graph.

The list of valid paths can then be stored in binary format in a *custom component* of the .CAP file. For instance 011, 010001 and 010000011 are valid paths in the exposed case.

At runtime, the JCVM is then responsible for dynamically computing the basic blocks, encoding the path it executes and checking the path whenever entering in a new basic block. Indeed, if the JCVM produces a path that is not one listed in the custom component, an error is detected.

Listing 4.16: A sample debit method in a Java Card e-Wallet applet (compiled bytecode)

```
 (0) aload_0
 (0) getfield #4 <fr/xlim/ssd/wallet/Wallet.pin>
 (0) invokevirtual #18 <javacard/framework/OwnerPIN.isValidated>
 (0) ifeq 98 (+91)
 (1) aload_1
 (1) invokevirtual #11 <javacard/framework/APDU.getBuffer>
 (1) astore_2
 (1) aload_2
 (1) iconst_4
 (1) baload
 (1) istore_3
 (1) aload_1
 (1) invokevirtual #19 <javacard/framework/APDU.setIncomingAndReceive>
 (1) i2b
 (1) istore 4
 (1) iload_3
 (1) iconst_1
 (1) if_icmpne 37 (+9)
 (2) iload 4
 (2) iconst_1
 (2) if_icmpeq 43 (+9)
 (3) sipush 26368
 (3) invokestatic #13 <javacard/framework/ISOException.throwIt>
 (4) aload_2
 (4) iconst_5
 (4) baload
 (4) istore 5
 (4) iload 5
 (4) bipush 127
 (4) if_icmpgt 60 (+8)
 (5) iload 5
 (5) ifge 66 (+9)
 (6) sipush 27267
 (6) invokestatic #13 <javacard/framework/ISOException.throwIt>
 (7) aload_0
 (7) getfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
 (7) iload 5
 (7) isub
 (7) i2s
 (7) ifge 83 (+9)
 (8) sipush 27269
 (8) invokestatic #13 <javacard/framework/ISOException.throwIt>
 (9) aload_0
 (9) aload_0
 (9) getfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
 (9) iload 5
 (9) isub
 (9) i2s
 (9) putfield #20 <fr/xlim/ssd/wallet/Wallet.balance>
 (9) goto 104 (+9)
(10) sipush 25345
(10) invokestatic #13 <javacard/framework/ISOException.throwIt>
(11) return
```

Conclusion to Part I

Along the four last decades, microelectronics and telecommunication have spectacularly evolved. Consequently, the capacities of smart cards have followed these evolutions. In addition, the growing needs for digital security, trust, privacy have offered these devices multiple applications throughout the world. This worldwide deployment is in a sense owed to the huge effort that has been taken to standardize the communication interfaces and protocols and to ensure the security of the embedded data. To ensure the required security, industrial actors (including IC manufacturers, evaluation laboratories, card manufacturers and application issuers) and academic actors are constantly questioning the implemented security mechanisms at both hardware and software levels.

Hardware-related attacks have proven to be very powerful to disclose secrets embedded in the card. In order to thwart such attacks, both hardware and software countermeasures are implemented in smart cards nowadays. For instance, clock jitter, random delays, redundant control modules, high-order masking, double and fake computations, are features that are common in modern smart cards. However, similarly to the attacks which have mainly focused on cryptographic calculation, the countermeasures are usually restricted to protect the same calculation and *in fine* the embedded secret cryptographic keys.

The integration of the Java technology within resource-constrained devices such as smart cards was not an easy bet. However, the Java Card technology has met a global success, in a first time in the mobile communication industry, and nowadays in all smart cards application fields. This success has been driven by the standardization and specification contribution of the different involved entities cited in this chapter: the Java Card Forum, the GlobalPlatform Inc., the ETSI and other 3GPP members, but also the OpenPlatform (ancestor of GlobalPlatform), the Smart Card Alliance[1], and EMVCo, amongst others. Furthermore, it tends to prove the need of these industries for fast and secure application development and deployment processes allowed by this technology, as well as for multi-application support. By defining a generic platform and by using a subset of a widespread language, it has drastically reduced the cost of embedded applications development and deployment.

Nevertheless, the eased application development and deployment is also an opportunity for attackers. As exposed, state-of-the-art logical attacks relies on the open

[1] http://www.smartcardalliance.org/

platform assumption, with either hypothesis 1 or 2 regarding the bytecode verification of the loaded application. Such an assumption is indeed a strong one since it assumes that the attacker knows the load key required by the embedded GlobalPlatform framework. In the real life, such a situation would only happen if the attacker is an evaluator from a certification laboratory, if she bought *white* cards on the internet, but in this case it cannot be assumed that other applications are loaded on the card, or if the load key has been disclosed, for instance by social engineering.

With the generalization of on-card byte code verifiers and the improvements in the verification process allowed by the various studies in this field, the combination of hardware attacks with malicious application has emerged as the only viable attack path. Surprisingly, although the seminal work introducing these so-called *Combined Attacks* was released in 2003 [GA03], no practically achieved attack have been publicly exposed until the talk we gave at E-SMART in 2009 [Bar09]. However, this work has seemingly revived the interest of both the academic and the industrial community interest for this topic, as attested by the various related works released since then [VF10, SLIC10, SLIC11, BICL11] and by the introduction of these attacks in standard Java Card platforms security evaluation processes [Lib12].

Part II

Analysis of the Security of Java Cards against Combined Attacks

Introduction to Part II

As exposed in Chapter 4, up to 2009, the majority of the attacks against Java Card platforms were relying on the execution, and therefore the loading, of ill-formed applications.

These attacks were then generally based on the corruption of the binary representation of a Java Card application (*.cap* or *.class* file) into a so-called ill-formed application before it is loaded on-card. Such modifications aim at circumventing certain controls enforced by the JCVM. But in most cases, they also make the application illegal with regards to the Java Card specifications. Therefore the modified application should not be able to pass static analysis tools such as the Java bytecode verifier. The bytecode verification being a costly process, it is generally executed off-card on Java Card 2.2.2 and earlier, as a part of the application development tool chain. The usual philosophy of Logical Attacks is then to skip this step and to directly load unverified applications on platforms that allow it.

The recently released Java Card 3.0 Connected Edition specifications, has made mandatory the on-card execution of the bytecode verification. Therefore loading ill-formed application appears more difficult. Attackers then need to find another way to have applications running on the card with few regards for the Java Card language rules. The use of fault injection, well known from the embedded cryptography's field, to provoke incorrect behaviours appears then as a viable solution. Combining a malicious but well-formed application with an adequate fault injection may allow an attacker to provoke the same behaviour she used to have with her ill-formed application by perturbing various parts of the platform. Figure 4.9 echoes Figure 2.5 and presents the different layers that can be targeted by a fault injection on a Java Card.

This state of fact has given a push to the introduction of the combination of Logical Attacks with Fault Attacks into the Java Card field and practical applications have been published over the last two years. In these works Fault Attacks are used to bypass certain security mechanisms in order to allow a Logical Attacks. The so-called Combined Attacks allow then to take the benefits of both Fault Attacks and Logical Attacks. Indeed, they are more realistic than Logical Attacks since they do not rely on an unverified application loading and potentially more powerful than Fault Attacks since the malicious application can make permanent changes and act like a trojan inside the card.

Although the concept of Combined Attacks have been introduced a few years earlier in theory, we presented the first practically achieved Combined Attack at E-SMART in September 2009 [Bar09]. The second part of this dissertation describes the works

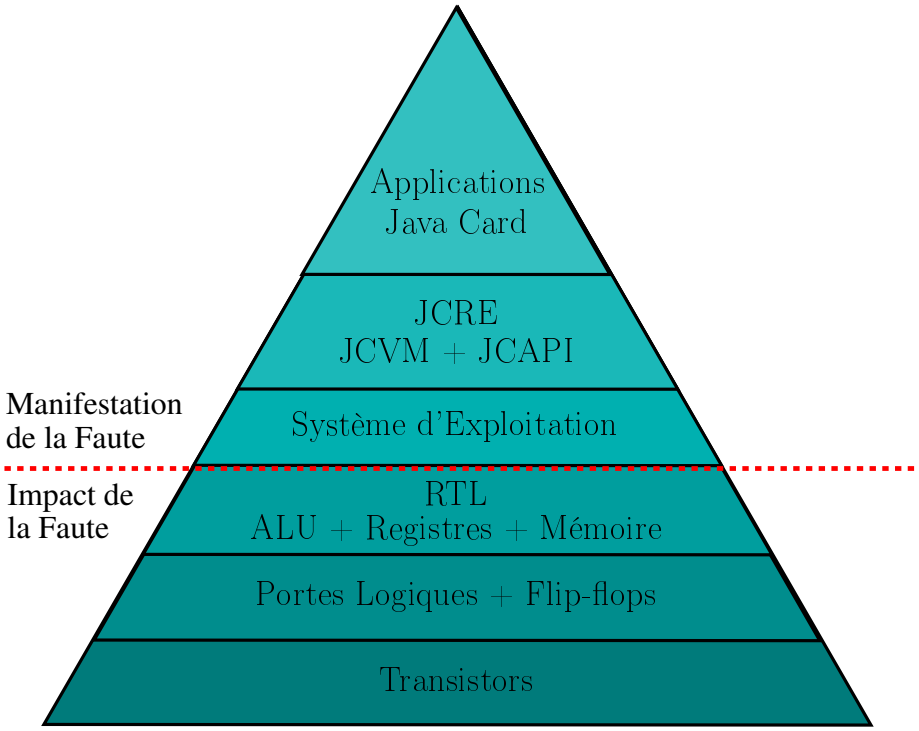


Figure 4.9: Fault causes and consequences.

achieved in order to evaluate and analyse the actual security of Java Card platforms against Combined Attacks. This security analysis has consisted in searching for different attack paths based on the combination of a well-formed application (or even several applications) and a fault injection. This study reveals that several properties enforced by the JCRE are likely to be disrupted or circumvented by Combined Attacks.

Chapter 5

Security Analysis of the Type Safety Property

Contents

5.1 Disruption of the `checkcast` Instruction 82
5.1.1 Recalls on Type Conversion 82
5.1.2 From Theory to Practice 83
5.2 Disruption of the Operand Stack 87
5.2.1 The Operand Stack, a Central Element of the JCVM. 88
5.2.2 The Selected Fault Model 88
5.2.3 Yet Another Way to Type Confusion 89
5.2.4 Instance Confusion 89
5.3 Exploitations of the Type Confusions 92
5.3.1 Exploitations of the `Class` Forgery 92
5.3.2 Servlet Impersonation (The `String` Theory) 97
5.3.3 Instance Confusion: Impersonating an Authentication Service . . 99

The type safety property is the cornerstone of all Java systems, including Java Cards, and particularly regarding the security. It is therefore natural that attackers trying to tamper with the system would try to break this property by provoking a type confusion. This chapter describes two different techniques attackers may be able to put into practice to create such a type flaw and exposes a couple of ways these type confusions could be exploited on recent Java Card platforms. The attacks presented here implicitly assume that the attacker has the opportunity to load and execute her own application on the platform. Because of the limitations coming with the GlobalPlatform framework introduced in Section 3.3.1, this privilege is far from obvious on released products. However such attacks must be considered in the context of platforms allowing post-issuance application loading like Java Cards.

The first section hereafter describes the first published attack (concomitantly with the article of Éric Vétillard *et al.* [VF10]) combining a fault injection and a malicious application allowing to defeat the security of a Java Card. This work was led with Hugues Thiebeauld

and Vincent Guerin and appeared in the proceedings of CARDIS 2010 [BTG10]. The novelty of this work is indeed twofold:

- firstly, such a Combined Attack had never been exploited until then. It turns out to be a very efficient way to tamper with a device like a Java Card.
- secondly, it shows that even if the Java Card 3 standard appears to have been designed with a real concern for security, it is still possible to attack devices embedding a straightforward Java Card 3 specifications implementation.

The demonstration that the attack is not only theoretical is given by exposing how it was successfully put into practice on a recent chip.

Another way of tampering with the type safety property is to disrupt the operand stack which is massively used by the JCVM. The second section of this chapter investigates this path and describes how a fault injected in the operand stack can lead to a type confusion. The results of this investigation carried with Guillaume Duc and Philippe Hoogvorst are published in the proceedings of CARDIS 2011 [BDH11]. Similarly experimental results are given, proving thus the practicability of the attack.

Finally, various exploitations of different type confusions are exposed, proving the variety of attacks that can be built upon these. The first exploitations were presented in the two previously cited publications whereas the last one was presented during E-SMART'10 [Bar10].

5.1 Disruption of the `checkcast` Instruction

In this section, an attack based on the disruption of the code executed during the interpretation of the `checkcast` instruction is described. After presenting this instruction, both the malicious application and the physical perturbation involved in this Combined Attack are detailed.

5.1.1 Recalls on Type Conversion

As previously stated, Java objects' types (classes) organization forms a hierarchy. Each class is a subclass of another class, except for the `Object` class on top of the hierarchy. The hierarchy defined by the Java classes architecture enforces the principle of conversion which allows an object of type `T1` to be used as if it were an object of type `T2` through a so-called *typecasting* operation. A type conversion can be explicitly requested in the source code by the use of the cast operator: `()`.

For obvious type safety reasons, such conversions must be checked. Conversions proven incorrect at compile time, or during bytecode verification, result in a error. But in certain cases, the check will happen at runtime *via* the `checkcast` instruction produced by the compiler and executed by the VM. Such an example is given below:

```
T1 t1;                          aload <t1>
T2 t2 = (T2) t1;      ⇔         checkcast <T2>
                                astore <t2>
```

The `checkcast` instruction is then of the utmost importance with regards to the type safety property. The specification of this instruction is as follows [Sun09g]:

Format:

checkcast
indexbyte1
indexbyte2

Operand Stack:

..., objectref ⇒ ..., objectref

Description:

The objectref must be of type reference. The unsigned indexbyte1 and indexbyte2 are used to construct an index into the runtime constant pool of the current class, where the value of the index is (indexbyte1 « 8) | indexbyte2. The runtime constant pool item at the index must be a symbolic reference to a class, array, or interface type. The named class, array, or interface type is resolved.
If objectref is null or can be cast to the resolved class, array, or interface type, the operand stack is unchanged; otherwise, the checkcast instruction throws a ClassCastException.

5.1.2 From Theory to Practice

The following exposes how the Combined Attack was mounted and performed.

5.1.2.1 A Malicious Application...

As exposed in Section 4.4.1, Govindavajhala *et al.* propose in [GA03] a way to achieve type confusion and reference forgery on a virtual machine thanks to memory errors. The approach used hereafter, although slightly different, is inspired by their attack.

In the scope of this attack, consider the classes defined in Listing 5.1[1].

The following focuses on the internal representation of instances of `B` and `C` classes (*cf.* Figure 5.1). It is important to notice that both objects have identical internal structures.

The raised question is then: What if an attacker manages to access a given object either as an instance of B or C? Treating this object as a `B` instance, it would be possible to set the value of its short field (`b.addr = 0x1234;`). And due to the internal structures

[1]The size of a reference is implementation specific. The type of field `addr` in `B` could be either `short` or `int`.

Listing 5.1: The involved classes

```
public class A {
  byte b00,...,bFF;
}

public class B {
  short addr;
}

public class C {
  A a;
}
```

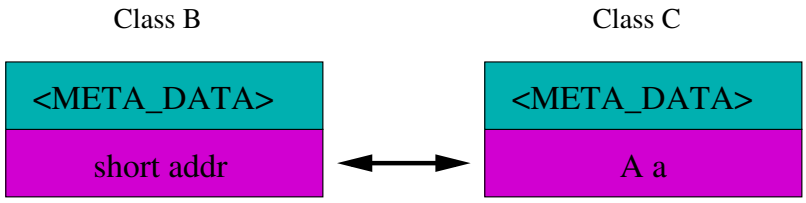


Figure 5.1: Internal representation of instance of `B` and `C`

of `B` and `C` classes, doing so consequently sets the reference of the `a` field of this very object seen as an instance of `C`, as illustrated in Fig. 5.2.

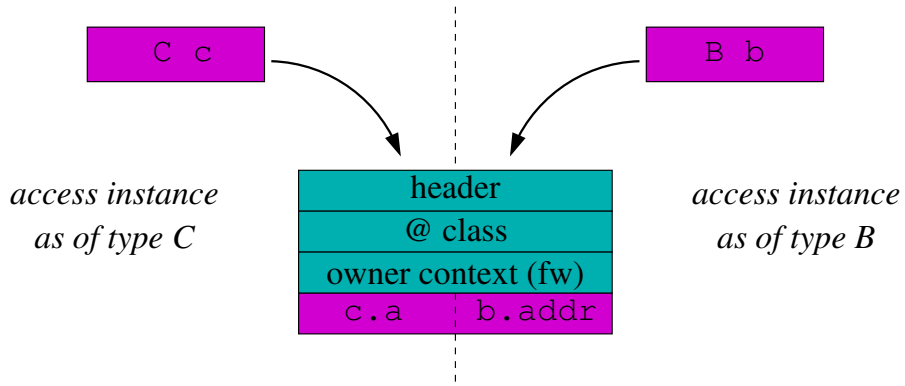


Figure 5.2: Access to the same object either as `B` or `C` instance

Now let an application module containing the extended applet described in Listing 5.2 (as well as classes `A`, `B` and `C`) be loaded, and this applet installed, on a recent chip embedding a straightforward implementation of the Java Card 3 *Connected Edition* specifications.

Obviously, this application is well-formed and the OCBV will allow its loading and installation. However, one may have noticed the incorrect cast conversion of a `B` instance

Listing 5.2: The malicious applet

```
public class AttackExtApp extends Applet {
  B b; C c; boolean classFound;
  ... // Constructor (objects initialization), install method
  public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    ...
    switch (buffer[ISO7816.OFFSET_INS]) {
      case INS_ILLEGAL_CAST:
        try {
          c = (C) ( (Object) b ); // Casting to Object prevents compilation error
          return; // Success, return SW 0x9000
        } catch (ClassCastException e) {/*Failure, return SW 0x6F00*/}
        ... // more later defined instructions
      }
    }
  }
}
```

into a `C` instance (line 10)[2]. Checking the correctness of cast conversions is not in the scope of the OCBV, it is left to the `checkcast` execution, at runtime, which should prove this one incorrect and result in a `ClassCastException`.

5.1.2.2 ... Combined with an Appropriate Fault Injection

In [VWG07], *Vermoen et al.* successfully applied the principle of Power Analysis (PA) [KJJ98] to Java Cards in order to reverse engineer an applet. In the particular case here, the attacker will only rely on PA to monitor the execution of the malicious application, which she obviously knows.

By analysing the acquired power consumption curve, she can then determine the moment when the `ClassCastException` is thrown and thus when the `checkcast` is executed, as exposed in Figure 5.3.

The experiments led consist in attempting to disturb the execution of the application at the precise moment when the `checkcast` is executed. For this purpose, a laser equipment was used. More precisely an infrared laser beam was applied on the back side surface of the chip. Figure 5.4 depicts the faulty execution of the application when the fault injection is successfully achieved. Finding the adequate fault injection parameters (location, intensity, duration of the laser shot) is highly dependent on the underlying hardware.

As exposed in Figure 5.4, the instruction's execution is a little shorter than the regular execution in Figure 5.3 (because the system does not have to treat an exception raising and handling) and that the returned status word is the one expected when no error has

[2]The `Object` conversion is only meant to fool certain compilers. This conversion will probably not even be checked as each class is a subclass of `Object`.

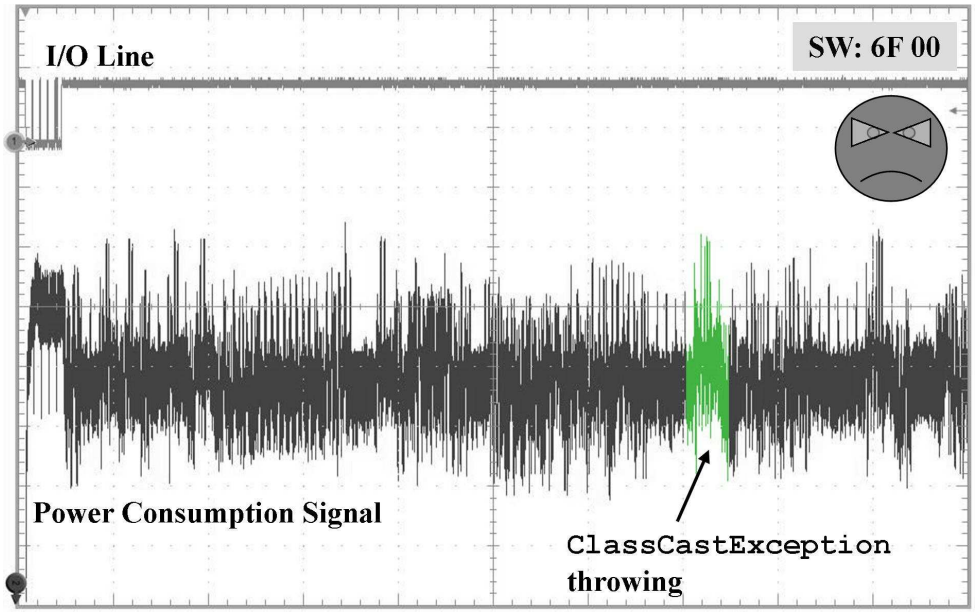


Figure 5.3: Execution of the applet's INS_ILLEGAL_CAST instruction

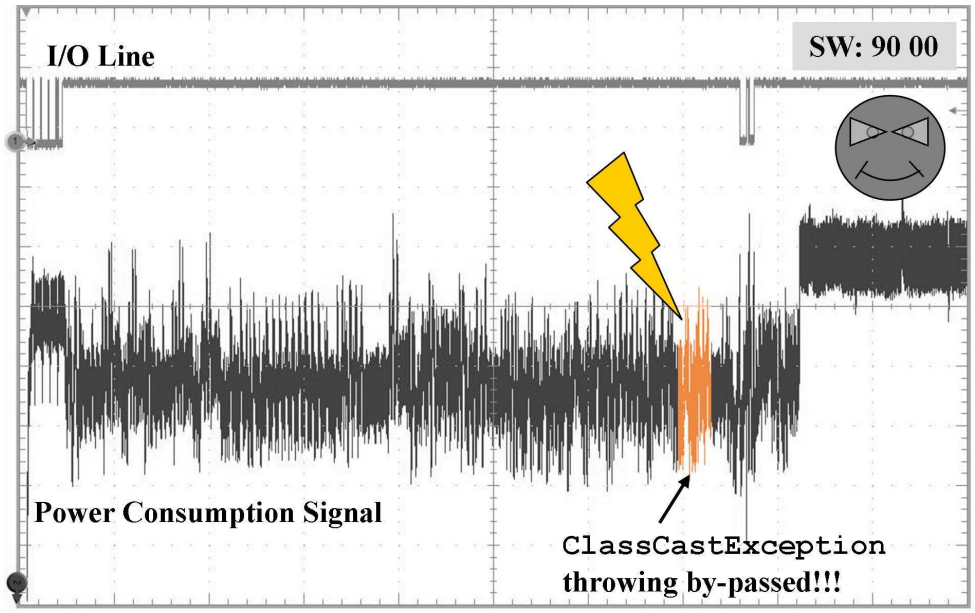


Figure 5.4: Disturbed execution of the applet's INS_ILLEGAL_CAST instruction

occurred (`90 00`). The desired type confusion has been provoked.

The attacker can then access an instance of class `B` either as an instance of class `B` or `C`. Consequently, she can forge `c.a`'s reference to any value (by setting the value of `b.addr`), which in turn may let her read and write as many bytes as declared byte fields in the definition of class `A` (*cf.* Figure 5.5).

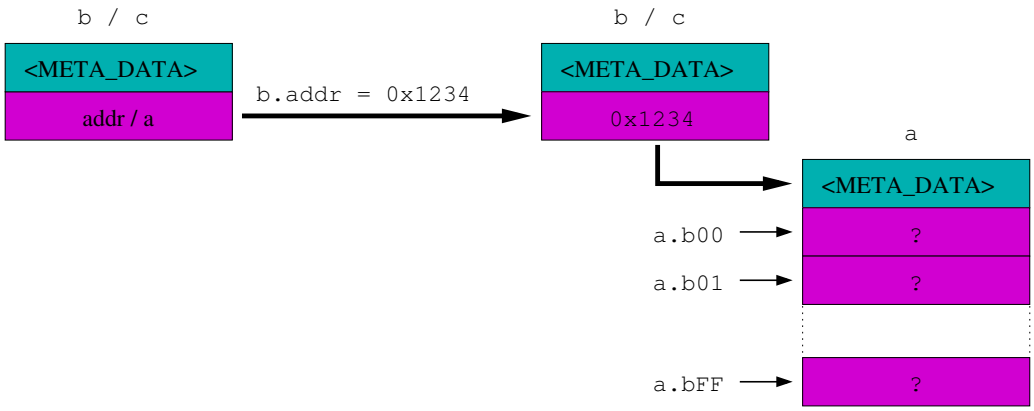


Figure 5.5: Forgery of object `a`'s reference

It is worth noticing that this can be done without requiring any additional physical disruption. The sole security check encountered has been permanently[3] neutralized by one single fault injection.

Can the whole system memory be dumped? Experimental results on the targeted platform have proven that the answer to this question is : No. And this is most probably the case on many other platforms. Accessing the memory through Java object references, the reachable memory segments should only correspond to the Java heap. Furthermore, access to an object that is not owned by the application and not shared must be forbidden by the application firewall, as specified in [Sun09e] and result in a `SecurityException` being thrown. Also, the behavior of the platform when trying to access bytes beyond the object's size, thanks to forgery, is not specified and is then typically implementation dependent.

Nevertheless, the attacker is in position to assign any reference to `c.a` and possibly read and write bytes `c.a.b`*XY* within the boundaries fixed by the application firewall and the JCRE implementation. This situation is roughly equivalent to that resulting from the type-confusion-based attacks presented in Chapter 4 on Java Card 2.x platforms. However, no ill-formed application loading is required and the attack does not rely on any particular specification/implementation flaws, the fault injection being the type confusion's cause.

5.2 Disruption of the Operand Stack

In this section the combination of a fault injection in the operand stack and a malicious application is considered. The following describes two CA taking advantage of a faulty object reference on the operand stack in slightly different ways: type confusion and instance confusion.

[3] As long as the application is not deleted from the card.

5.2.1 The Operand Stack, a Central Element of the JCVM.

The JCVM, and more generally, Java Virtual Machines (JVMs) are known as stack-based machines, in opposition to register-based machines. Actually, several stacks are described in the JVM specification [LY99]. This section focuses on one kind of these: operand stacks.

A Java *frame* is created on each Java method *invoke* to store temporary VM-specific data. The operand stack is the part of this frame in charge of holding the operands and results of the VM instruction. Most of these instructions consist in popping a certain number of operands, executing a specific process and pushing a returned value. For instance, the execution of an `iadd` (adding two integer values: *value1* and *value2*) is specified as follows [Sun09h]:

Format:

`iadd`

Operand Stack:

..., value1, value2 $\Rightarrow$ ..., result

Description:

Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 + value2. The result is pushed onto the operand stack.
The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type int. 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.
Despite the fact that overflow may occur, execution of an iadd instruction never throws a runtime exception.

The integrity of the values passing through the operand stack appears then crucial. The remainder of this section is focused on this central element of the JCRE and study its robustness with regards to fault injections.

5.2.2 The Selected Fault Model

The fault attack considered here only targets the operand stack of a JCVM. A fault model allowing the attacker to modify a value pushed onto the operand stack into a predetermined, or even a chosen value can therefore be defined. Indeed two different fault models are considered: the common *stuck-at* fault model and a model taking into account the value previously pushed onto the stack. This model is detailed below.

In the constrained context of single-threaded Java Cards, optimization may lead to use a single global operand stack. However, according to the specifications, an operand stack is allocated within a Java frame, on a method *invoke*. In both cases, this allocation should be done in RAM for performance concerns and because of the limited endurance

of NVM modules. Therefore, pushing an operand on the stack consists in writing this operand at a given address that only depends on the number of elements already on the stack and that is incremented at each push operation.

The fault model assumes that the perturbation allows to prevent (at least partially) the updating of the operand stack during a push operation. The resulting erroneous value would then result from an incomplete push operation. As a consequence, and assuming the attacker knows the values previously pushed onto the operand stack, she can predetermine the erroneous value. Furthermore, assuming she can run and attack her own application on the platform, she can choose the value previously pushed onto the stack and therefore control the resulting erroneous value. The experimental validation of this fault model is shown in Appendix A.

The following sections details possible exploitation of fault injections following this fault model through both FA and CA.

5.2.3 Yet Another Way to Type Confusion

The first attack that can come from a faulty operand stack would be to provoke a type confusion on the host platform. The strategy to break type safety is basically the same as the one proposed in [GA03] which was described in Chapter 4. That is to say, the attacker creates in her application several instances of a given class C and counts on an error to modify the Java reference of a given instance of another class $C*$ into that of one of the several instances of C. The main difference with the work presented in [GA03] is that the defined fault model allows to predetermine the error. Therefore the success rate of the attack should not depend on the number of instances of class C that have been created although a sufficient number of instances of class C can be necessary in practice.

5.2.4 Instance Confusion

The second attack resulting from the analysis of the behaviour of the operand stack with regards to fault injection is based on a so-called instance confusion. This section introduces the concept of instance confusion and presents the case study of an attack using this concept.

5.2.4.1 Instance confusion.

By analogy to the concept of type confusion, where an instance of a class C is used as if it were an instance of another class $C*$, the concept of *instance confusion* can be derived from. An instance confusion consists in using an instance i of a given class (or of a class implementing a given interface) as if it were another instance $i*$ of the same class (or of a class implementing the same interface).

Obviously, instance confusions within the bounds of the attacker's application may not represent a threat. Furthermore, to take advantage of an instance confusion outside the

bounds of her application, the attacker should have to circumvent the Java Card application firewall. However, as exposed in Chapter 3, classes implementing a *shareable interface* are likely to cross the application firewall. Section 5.3.3 will show that an appropriate instance confusion can allow the attacker to unduly gain privileges in another application on a Java Card 3.0 through the use of an authentication service.

5.2.4.2 Experimental results.

In the scope of this work, the experiments were done on a Java Card 2.2.2. The attack targets an applet holding one instance a of a class A implementing an interface I and 256 instances of a class B, also implementing I. The aim is then to prove that an attacker can provoke an instance confusion on specific objects. Following the attack scenario, the application obtains object a through a virtual method `getA()` (which implementation is given in Listing 5.3) and stores it in a local variable of type I, the interface implemented by A and B.

Listing 5.3: The getA method's bytecode

```
getfield_a_this #5              // return this.a;
areturn
```

The attack consists then in injecting a fault while pushing a onto the stack as part of the `getfield_a_this` instruction which specification is detailed hereafter as per [Sun06c].

Format:

getfield_a_this
index

Operand Stack:

..., ⇒ ..., value

Description:

The currently executing method must be an instance method. The local variable at index 0 must contain a reference objectref to the currently executing method's this parameter. The unsigned index is used as an index into the constant pool of the current package [Sun06c, §3.5]. The constant pool item at the index must be of type CONSTANT_InstanceFieldref [Sun06c, §6.7.2], a reference to a class and a field token.

The class of objectref must not be an array. If the field is protected, and it is a member of a superclass of the current class, and the field is not declared in the same package as the current class, then the class of objectref must be either the current class or a subclass of the current class. The item must resolve to a field of type reference.

The width of a field in a class instance is determined by the field type specified in the instruction. The item is resolved, determining the field offset. The value at that offset into the class instance referenced by objectref is fetched. If the value is of type byte or type boolean, it is sign-extended to a short. **The value is pushed onto the operand stack.**

The attacker can then check if the resulting operand is an instance of B. The test application is as described in Listing 5.4.

Listing 5.4: The search class instruction

```
I a = getA(); // attacked method

if (a instanceof Object) {
  if (a instanceof B)
    // Return SUCCESS
  else if (a instanceof A)
    // Return FAILURE
}
```

Out of 10,000 attacked executions of the application, the following results were obtained:[4]

[4] As expected with regards to the defined fault model, increasing the number of instances of class B up to 1024 did not really enhance the success rate of the attack (almost 10%).

- 8.74% of success: the operand popped from the stack is an instance of B.
- 25.42% of attack failure : the operand popped from the stack is an instance of A, i.e. a itself, the fault injection had no effect.
- 65.86% of unknown error : the execution of the application did not complete, *i.e.* an exception was thrown or the fault injection caused a card failure.

Reminding that the attack is designed to defeat the security of PIN- or password-based `Authenticator` a success rate of about 10% is quite outstanding, considering the theoretic security of about 2^{40} of an 8-character password.

5.3 Exploitations of the Type Confusions

This section presents different exploitations of the type confusions created by the previously presented Combined Attacks. The two first exploitations take advantage of context-less classes (namely `Class` and `String`). The third exploitation implements the instance confusion attack described in Section 5.2.4, using an authentication service.

5.3.1 Exploitations of the `Class` Forgery

5.3.1.1 Corruption of On-Card Application's Code (A Horse with no Name)

The following exposes how the reference forgery tool presented in Section 5.1.2 can jeopardize a Java Card 3 *Connected Edition* platform thanks to the new dynamic features. The working hypothesis is set before explaining how one can access and modify `Class` objects on the platform.

5.3.1.2 An Assumption about the `Class` Object

One of the features introduced by the Java Card 3 *Connected Edition* platform is on-card class loading. This can be used within an application thanks to the `java.lang.Class` class that has been added to the standard API [Sun09b] which specifies that *"Instances of the class* `Class` *represent classes and interfaces in a running Java application"*. `Class` objects are constructed by the class loading process, as defined in the standard Java VM specification [LY99], from the binary representation of these classes (the `.class` file). Working in a constrained environment, one cannot expect the `Class` object to be the exact copy of the `.class` file. Nevertheless the following hypothesis is ventured:

Hypothesis 3. *The bytecode of a class is stored within the internal structure of the corresponding* `java.lang.Class` *instance.*

This assumption appears quite natural as the `Class` object aims at representing a running or ready-to-run class. Although this assumption could be discussable, it appears quite natural to have in a single place the informations needed at run time. And this is exactly the aim of the `Class` object. Besides, if the `.class` file format can be optimized

in some ways, the bytecode array itself cannot be modified without modifying the behavior of the methods it represents.

Another interesting point concerning `Class` object is that the specification requires root classes (applet, servlet, filter and listener classes), dynamically loadable classes and shareable interface classes of an application module to be loaded and linked during this application module loading. Thus these instances of the class `Class` are constructed as soon as an application is loaded.

5.3.1.3 Searching and Accessing `Class` Objects

Section 5.1.2 shows how the attacker can forge the reference of an `A` instance and access memory using its byte fields. This access will be executed on the platform respectively by the `getfield` and `putfield` instructions whether the attacker tries to get (read) or set (write) the byte value. A particularity of Java Card 3 *Connected Edition* is that its specification [Sun09e] allows access to implicitly transferable objects *via* `getfield` and `putfield` instructions. An implicitly transferable object is an object that is not bound to a specific java context.

Therefore, when an application requests access to such an object, the application firewall will grant the access instead of checking that the java context of the application matches the requested object's java context. In other words, such objects are not protected by the application firewall. The list of specified implicitly transferable classes [Sun09e] contains an interesting element: `java.lang.Class`.

To access a `Class` object (*i.e.* to forge `b`'s reference to that of a `Class` object instance), the attacker needs to know the fully qualified name of this class and have the instruction given in Listing 5.5 in the process method of the attack application.

In this applet instruction, the attacker already takes advantage of the implicitly transferable property of `Class` objects by using type conversion, the `instanceof` instruction and the `getName()` method on the forged reference (lines 9 and 12).

Another way to achieve this could be to use the `hashCode` method of the `Object` class provided it is typically implemented as per [Sun09b] :

> "As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™programming language.)"

Under Hypothesis 3, and provided the attacker can forge `a`'s reference to a `Class` object's reference, she can modify this class's bytecode array, regardless of the application it belongs to.
The following section exposes two case studies of such an attack.

Listing 5.5: The search class instruction

```
case INS_SEARCH_CLASS:
  while (!classFound) {
    try {
      // Increment the forged reference
      b.addr++;
      // Convert the bytes given in APDU command into String
      String name = bytesToString(buffer, ISO7816.OFFSET_CDATA);
      // Is it a Class instance ?
      if (((Object) (c.a)) instanceof Class) {
        // Is it the Class instance we're looking for ?
        // Let us check its name
        if (((Class)((Object) (c.a))).getName().equals(name))
          classFound = true;
      }
    } catch (SecurityException se) {}
  }
  break;
```

5.3.1.4 Ill-Formed Code Injection

The OCBV prevents ill-formed applications from being loaded. This case study will show that the reference forgery tool enables an attacker to execute any sequence of bytecode instructions.

Assume the attacker's application module described in Section 5.1.2 contains an additional class with dummy methods filled with instructions meant to produce an easy to detect (and to modify) bytecode within the corresponding `Class` instance.

To access her dummy `Class` object, she only has to use the `INS_SEARCH_CLASS` instruction with the proper class name (she obviously knows) to forge `a`'s reference. She can then easily read the content of the `Class` object and detect the bytes corresponding to her dummy method. She can finally write the bytecode she wants, in disregard for any rule (Figure 5.6).

This proves that under Hypothesis 3, one can use the reference forgery tool to eventually have ill-formed code loaded on card despite the OCBV and without any additional fault injections.

Considering the state of the art, an application containing erroneous bytecode will not be more hazardous than the type-confusion already achieved. Actually, the attacker has the same chances to dump memory than with the attacks published in [Wit03] and [MP08]. With a good knowledge of the `Class` object's structure she could also try to modify the static field resolution, as proposed in [Hyp03] and used in [ICL10], to circumvent the application firewall. Nevertheless, this may enable future ill-formed-application-based attacks to target platform protected by OCBV.

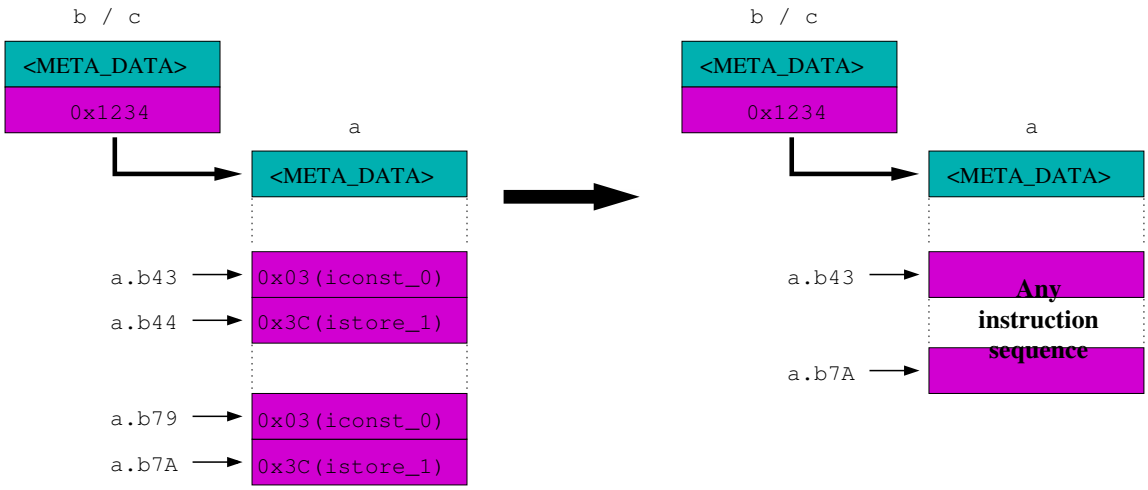


Figure 5.6: Identification of `dummyMethod` and ill-formed code injection

5.3.1.5 Modifying any Application Behavior

Unlike the previous exploitation, the following exposes how to fully take advantage of the implicitly transferable property of `Class` objects. Thanks to the reference forgery tool, the attacker can also modify any other applications regardless its context, endangering thus the whole platform integrity.

To illustrate how dangerous this can be, the case study of an application whose security relies on user/client authentication based on a signature scheme is exposed hereafter. One can note that the designers of this application can be totally confident in the embedded signature scheme, since its implementation has probably been certified resistant against various kinds of side channel and fault attacks.

Therefore, somewhere in this application's code, lines similar to those given in Listing 5.6 will appear.

Listing 5.6: Signature verification

```
if (sig.verify(inBuff, inOff, inLen, sigBuff, sigOff, sigLen)) {
  ... // Success, access granted.
} else {
  ... // Failure, access denied.
}
```

Consider now an attacker who wants to access this application's assets. Without the knowledge of the signature's private key he cannot be successful. But the forgery tool will allow her to circumvent this obstacle.

Thanks to the transferability of `Class` objects, under Hypothesis 3 and provided the

attacker knows the fully qualified name of the class containing the call to the `verify` method, she can then forge a reference to the corresponding `Class` instance (still using the `INS_SEARCH_CLASS_OBJECT` instruction). She will then have access to its bytecode array.

Knowing the `verify` method's descriptor, she can deduce that a call to this method will consist in pushing the `sig`'s reference and all the arguments on the stack (`inBuff`, `inOff`, ...).

She can then identify the bytes involved in the call to the `verify` method in the bytecode array (Figure 5.7).

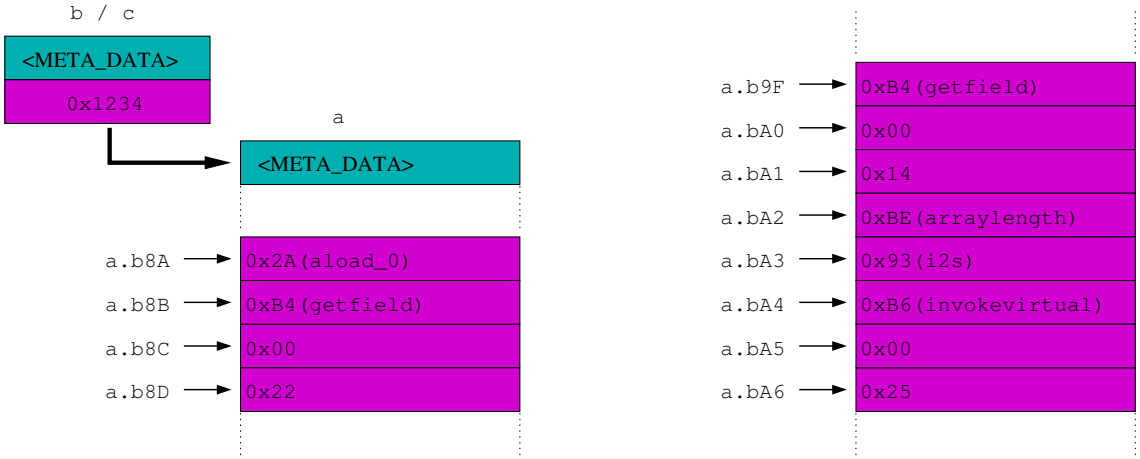


Figure 5.7: The call of the `verify` method

Finally, she just has to set all these bytes to `0x00`, which corresponds to the `nop` instruction (*i.e.* `no operation`), except the last one, to which she assigns the value corresponding to `iconst_1`, pushing the value `1` on the stack (Figure 5.8).

Operating this modification, the attacker changes the application's code as if the Java source code was that of Listing 5.7.

Listing 5.7: Signature verification

```
if (true) {
  ... // Success, access granted.
} else {
  ... // Failure, access denied.
}
```

She is then granted access to the application's assets whatever the value of the signature.

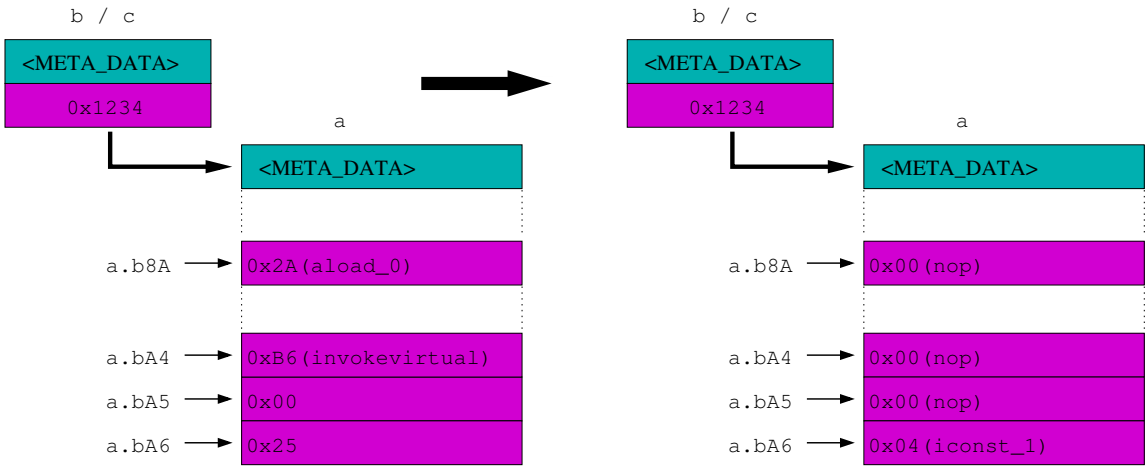


Figure 5.8: Making the signature verification always successful

This simple case study shows the potential threat such attacks can represent. The attacker's application becomes a Trojan horse capable of modifying other applications from the inside (as suggested in [ICL10]). The number of possible attack *scenarii* is only limited by the attacker's imagination. Besides, although a good knowledge of the targeted application would ease the attack, it may not be mandatory. If an attacker can read the content of all `Class` objects, identifying her target amongst those should not require huge efforts.

5.3.2 Servlet Impersonation (The `String` Theory)

The exploitation described hereafter shows how a particular type confusion can allow a given servlet to impersonate another one on the same platform. As detailed in Chapter 3, servlets are mapped to URIs on the platform. From a programmatic point of view, these URIs correspond to string of characters, *i.e.* instances of the `String` class, which is one of the novelties introduce by the Java Card 3.0 Connected Edition API [Sun09b].

5.3.2.1 Specificities of the `String` Class

As specified above, the String class is meant to represent string of characters. Therefore one can expect this class to hold fields containing for instance a pointer to an array of characters, and integer values delimiting the string of character in this array (say a start offset and the string's length). The assumed internal representation is given in Figure 5.9. However the important point is rather that the attacker can find out the internal structure than assuming a particular one.

Actually, access to these alleged fields is prohibited as the String class does not expose them. In addition, since the String class is defined as a *final* class, it cannot be extended by any subclass. Building a subclass that would try to access these fields is

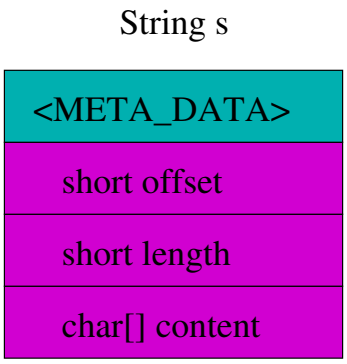


Figure 5.9: Assumed internal representation of a String object.

therefore not an option.

In order to comply with the constrained smart card environment, the JCRE specification [Sun09e] specifies that the platform maintains a pool of *interned* strings of characters that are likely to be used by various applications. The interned string pool contains:

- Strings defined by the platform (such as "true" or "false" for instance).
- Strings defined literally by applications, *e.g.* `String s = "literal";`.
- String instances on which the API method `String.intern()` has been called.

Furthermore, the platform is supposed to ensure the uniqueness of interned strings. Last but not least, due to their implicit *"shareable"* status, interned strings are specified as being contextless [Sun09e, §X], *i.e.* they are not protected by the application firewall. This is indeed the reason why these particular objects are studied here.

5.3.2.2 Accessing the Underlying Array of `char`

The aim of the attack is to manage to corrupt interned `String` instances. The first step consists then in getting access to their underlying array of characters. For that reason, the class defined in Listing 5.8 is considered, according to the assumed structure of the API's `String` class, that could be loaded on-card by an attacker.

Listing 5.8: The `MyString` class

```
public class MyString {
  short offset;
  short length;
  char[] content;
}
```

In a second step, assume that the attacker provokes a type confusion between instances of the classes `StringContainer` and `MyStringContainer` defined in Listing

5.9.

Listing 5.9: The confused container classes

```
public class StringContainer {
  String s;
}

pubic class MyStringContainer {
  MyString ms;
}
```

Therefore, by accessing the `String` instance `s` as if it were an instance of class `MyString`, the attacker would be able to access the hidden fields of `s`. Consequently, the attacker is potentially able to corrupt the actual strings of characters pointed to by references to the interned `String` pool, and even to replace the content of a given String with that of another. As shown in the following, this can have serious consequences.

5.3.2.3 Servlet Impersonation

As introduced in Chapter 3, servlet application are mapped to specific URIs that are defined in the deployment descriptor of the .WAR file (*web.xml*). When a specific request is sent to the card, the JCRE is then responsible for dispatching it to the appropriate application depending on the URI specified in this request.

As a matter of fact, a URI is nothing more then a particular string of character respecting a specific format (defined in [BLMM94]). In addition, assuming that such strings of character are part of the interned `String` pool does not appear far-fetched. Therefore, if an attacker is able of corrupting the interned `String` pool, she would be able to switch the URI of her own application with that of another one. Consequently, all incoming requests sent to the second application would indeed be dispatched to the attacker's application. The attacker's application can then exploit this impersonation in several ways, such as asking the card holder's credentials for instance.

5.3.3 Instance Confusion: Impersonating an Authentication Service

5.3.3.1 JC3.0 user authentication.

The Java Card 3.0 specifications provide an authentication facility through a dedicated set of service interfaces. These interfaces are organized as illustrated in Fig. 5.10.

To allow user authentication, these shared services are mapped to specific Unified Resource Identifiers (URI) as any other Shareable Interface Object (SIO, refer to Chapter 3).
Each of these authentication service interfaces expose methods allowing to:

- authenticate a user with provided credentials (`check`),

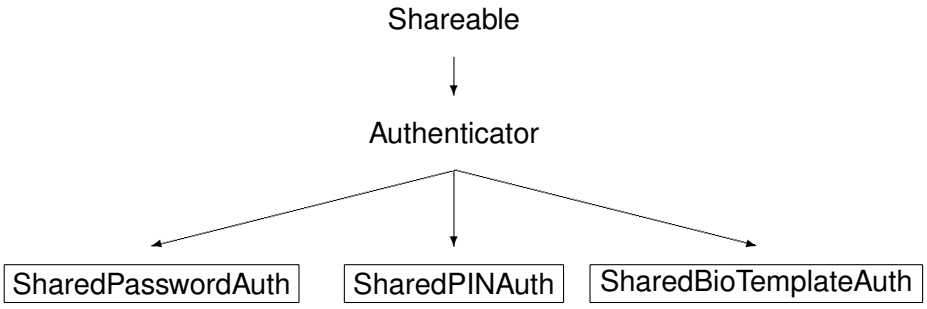


Figure 5.10: Java Card 3 authenticator classes and interfaces hierarchy.

- check wether or not a user is authenticated (`isValidated`),
- reset the authentication status of an authenticated user (`reset`).

These methods are typically called either by the application, or by the web container to restrict access to specific services or content, as detailed in the JCRE specifications (§6.4.4 and 6.4.5 of [Sun09e]). The important point to notice is that these methods are exposed in shareable interfaces. That is to say they are accessible across the application firewall. They are then likely to be abused by an attacker through an instance confusion.

5.3.3.2 Setting up and exploiting instance confusion.

Assume the attacked application uses the service offered by the `SharedPasswordAuth` interface. The first step for the attacker is then to create and load an application with several instances of a class implementing this interface. This class would typically have `check` and `isValidated` methods always returning `true` and `reset` method doing nothing, to bypass access control.

When the targeted application is about to get the authenticator instance (*i.e.* when the `lookup` method pushes its Java reference onto the operand stack), the attacker can then try to corrupt it. If she manage to provoke an instance confusion between the legitimate authenticator and one of her own, any call to the `check` or `isValidated` method would return `true` and the targeted client application would have all the reasons to consider her as an authenticated user.

The attacker may then access critical services within the attacked application. It is important to notice that even if redundant checks are performed to verify the authentication, one successful fault on the authenticator's reference is sufficient.

Chapter 6

Security Analysis of the Execution Flow Integrity

Contents

6.1 Disruption of the Operand Stack on Conditional Branching Instruction 102
6.1.1 Booleans and Conditional Branching in Java Card 102
6.1.2 Fault Attacks against Conditional Branching Instructions 103
6.1.3 Experimental Results 104
6.2 Multithreaded Corruption of the Execution Context 104
6.2.1 Involved Mechanisms 104
6.2.2 The Attack Concept 105
6.2.3 Practical Implementation on a Java Card 107
6.3 Disruption of `Exception`-Related Mechanisms 115
6.3.1 Exceptions in Java Card Platforms 115
6.3.2 Software, Fault and Combined Attacks: On the Security of Exceptions and Exception Handling 118
6.3.3 Consequences and Analysis 125

The importance of the execution flow of an application is obvious with regards to both its functionality and its security. Therefore one can easily imagine an attacker trying to tamper with it. This chapter presents the identified attack paths resulting from the analysis of the security of various mechanisms influencing the execution flow.

The first section of this chapter also results from the investigation on the consequences of errors induced in the operand stack introduced in the previous chapter. It highlights the fact that errors on boolean values manipulated in the operand stack are likely to modify the execution flow when conditional jump instructions are interpreted.

The second section relates an attack concept and its application to a Java Card 3.0 Connected Edition platform. This concept takes advantage of the multithreaded environment proposed by Java Card 3 Connected Edition platforms and provides the attacker with a total control on the executed applications. The attack concept and its

implementation on a Java Card 3.0 Connected Edition have been designed with Hugues Thiebeauld and are described in an article published in the proceedings of CARDIS 2011 [BT11].

Finally, the third section discusses the potential consequences of CA where the fault injection targets Exception-related mechanisms, namely exception throwing and handling. As exposed, such attacks are likely to corrupt the execution flow since the exception role in that matter has grown with the evolution of Java applications. The work presented in this section was led with Philippe Hoogvorst and Guillaume Duc. An article relating it has been published in the proceedings of SECRYPT 2012 [BHD12b].

6.1 Disruption of the Operand Stack on Conditional Branching Instruction

The work described in this section is related to the work already presented in Section 5.2 concerning the sensitivity of the operand stack to fault injection.

This section discusses the particular cases of the boolean type and of conditional branching instructions, describes fault attacks on such instructions, and gives experimental results proving the efficiency of the fault model defined in Section 5.2.

6.1.1 Booleans and Conditional Branching in Java Card

Amongst the basic types of the Java language different types of integral values differing by their size or sign can be found:

- the `byte` type representing signed 8-bit values,
- the `char` type representing unsigned 8-bit values,
- the `short` type representing signed 16-bit values,
- the `int` type representing signed 32-bit values,
- the `long` type representing signed 64-bit values (optional).

But a specific boolean type, which supports only two values: `true` and `false`, can also be found. Indeed, the Java language forbid the use of any other type than boolean in `if` statement, unlike C language for instance.

Nevertheless, there is no such thing as a boolean type at the bytecode level and the Java compiler produces only bytecodes manipulating values of type `int` when processing operations on boolean variables. Finally, and most importantly with regards to the remainder of this section, the conditional branching instructions produced by the compilation of a simple if statement: `ifeq` and `ifne`, only compare the top of stack value (*i.e.* the previously pushed operand) with 0 and branch or not depending on the result of this comparison (branch if the comparison succeeds in the case of an `ifeq`, branch if the comparison fails in the case of an `ifne`). That is to say, the specification imposes that

any other value than 0 will be interpreted as `true` by the JCVM.
One may note that this statement is true for any Java-based system.

6.1.2 Fault Attacks against Conditional Branching Instructions

Several choices are offered to an attacker in order to corrupt a conditional branching instruction on a Java Card. This section gives the details of a FA against an `ifeq` instruction evaluating a positive (true) condition by setting the previously pushed operand to 0. Listings 6.1 and 6.2 show respectively the test application Java source code and the corresponding bytecode.

Listing 6.1: The test application Java source code

```
boolean b = dummyTrue();
if (b) {
        Util.setShort(buffer, (short)0,(short)0x1111);
}
else {
        Util.setShort(buffer, (short)0,(short)0x2222);
}
Util.setShort(buffer, (short)2, proof);
```

Listing 6.2: The test application bytecode

```
  aload_0              // b = dummyTrue();
  invokevirtual #96
  istore 6
  iload 6              // if (b)
  ifeq 12
  aload_2              // Util.setShort(buffer, (short)0,
  iconst_0             //     (short)0x1111);
  sipush 0x1111
  invokestatic #84
  pop
  aload_2              // Util.setShort(buffer, (short)0,
  iconst_0             //     (short)0x2222);
  sipush 0x2222
  invokestatic #84
  pop
  ...
```

The `dummyTrue()` method initializes the instance field `proof` (used at line 8) proving the method has been executed and return a boolean value (`true`).

The target of the attack is this value pushing, before the `dummyTrue()` returns. The goal of the attack is then to force this value to 0. In case of success, the `ifeq` instruction will result in a jump at line 21 in the bytecode sequence and the returned value will be 0x2222 instead of 0x1111.

6.1.3 Experimental Results

This attack was put into practice on a recent smart card embedding a Java Card 2.2.2 Runtime Environment. The fault injection is achieved with a laser beam applied on the back-side of the component.

After empirically searching the fault injection parameters (timing, location, intensity, wavelength) that "maximize" the number of successful FA, the success rate of the attack reaches 78.25%, out of 10,000 disturbed executions of the application. The test application was then adapted to attack an `ifne` instruction by changing line 2 of the Java source code into `if (!b)`. Once the fault injection parameters adjusted, a success rate of 70.92% was reached, also out of 10,000 disturbed executions.

The results of similar attacks on a `false` condition evaluation are expected to be at least as good as the results obtained above. Indeed, since any value other than 0 is interpreted as `true`, any alteration of the pushed operand would lead to a successful attack.

6.2 Multithreaded Corruption of the Execution Context

This section exposes an attack concept exploiting the multithreading facility offered by Java Card 3 Connected Edition platforms as well as their communication capacities.

6.2.1 Involved Mechanisms

As stated above, a prerequisite for the attack concept is the support of multithreading and network communications over standard protocols. The following intends to outline these mechanisms as well as the working hypotheses to set the basement of this work.

6.2.1.1 Multithreading

The attack concept is grounded on the multithreading capacity of the targeted system, which allows the concurrent execution of different processes. In this work, multithreading is considered on single-core devices only. To process several threads simultaneously, the system assigns resources to a thread for a given time slice before switching to another. The entity in charge of distributing resources to the threads is the scheduler. Numerous rules can be used to decide when the scheduler will order a thread switching, *i.e.* to set the size of the so-called time slice. For instance thread switching can be triggered by a timer, an instruction counter, control flow breaks, access to certain resources, *etc*.

When a thread has consumed its allocated time slice, the scheduler orders a switch. This switch should be processed as follows :

- The current thread's execution context is saved.

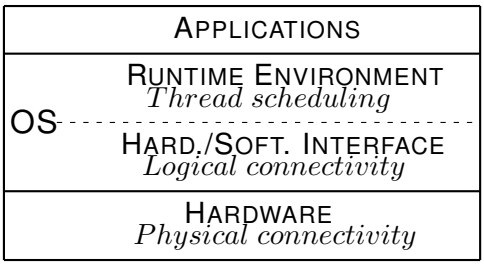


Figure 6.1: Alleged architecture of the system.

- The next thread is elected and its execution context is loaded.[1]

The different threads are then successively given access to the system resources.

The attack concept does not directly target the multithreading but rather lies on an abuse of this feature. Next section introduces the feature taken advantage of to achieve this abuse.

6.2.1.2 I/O Network Interfaces

Consider in the scope of this work a device providing a logical network interface supporting standard network protocols (TCP, HTTP(S), ...) over physical I/O interfaces. The aim of this section is to set the system architecture assumed in the remainder of this work.

As depicted in Fig. 6.1, these interfaces are not part of the so-called runtime environment (RE), but belong to lower layers. According to this statement, one can expect that some incoming requests are handled in these lower layers only and do not reach the RE. Therefore they do not enter the system's multithreading mechanism.

This last statement is a key element rendering the attack concept practicable, regardless of the targeted system. For the sake of clarity, Figure 6.2 illustrates it with different requests. As depicted on the figure, requests $\mathcal{B}$ and $\mathcal{D}$ handling leads to a thread creation within the RE, whereas requests $\mathcal{A}$ and $\mathcal{C}$ are handled by the system before entering the RE.

Now the basement of the attack concept has been set, introducing the required mechanisms and properties of the targeted system, the attack concept itself can be exposed.

6.2.2 The Attack Concept

This section introduces the attack principle in a generic way. The goal is to emphasize that this threat may potentially affect a wide range of systems. On the other hand the attack success is closely related to implementation choices in the platform, providing

[1] The next thread election may possibly take into account the concept of thread priority. This concept will not be further considered in this context since an attacker able to start new threads should also have the ability to modify their priority.

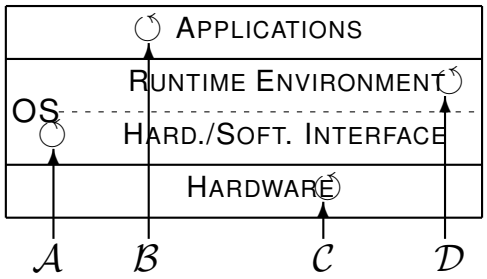


Figure 6.2: Different requests handling

some leads to find adequate protections.

The attack aims at altering a sensitive execution flow at a precise time. To achieve this, two steps must be performed as follows :

- Freezing the application execution at time T_0. It comes to setting a breakpoint on a specific operation.
- Altering the execution context available in memory to change the application behaviour when it is resumed.

Assume a preemptive thread scheduling mechanism based upon a timer. A certain amount of time is then allocated to each thread. When an execution exceeds the time slice T, the scheduler stops the process and switches to the next thread in the queue. This hypothesis is obviously not the single way to implement multithreading.

The attack relies on the corruption of the multithreading system to force the interruption of an application. The objective is to cheat the scheduler so that the thread switch occurs at T_0 rather than T. This is obtained by sending a sequence of requests to the device when the targeted thread is being processed. As the process in charge of the network requests is unlikely to be handled as a thread, the time of processing initially devoted to the current thread is lost. As a consequence the thread execution is curtailed, as depicted in Figure 6.3.

T_0 is then adjustable depending on the number of network requests sent. To appropriately determine T_0, it is better to have an idea of the thread execution flow. Nevertheless, the code knowledge may not be necessary as side channel analysis may provide sufficient information, depending on the attacked system.

Once the targeted thread is frozen, the scheduler switches to the next ones in the queue. During this time the context of the sensitive code is available in memory. An attacker in position of executing a malicious application would have then the opportunity to get to the context in memory and alter it. Depending on the attacked system, the right to load and run a application can be more or less restrictive. However in the context of a multi applicative platform these rights necessarily exist and may therefore be the target of an attack.

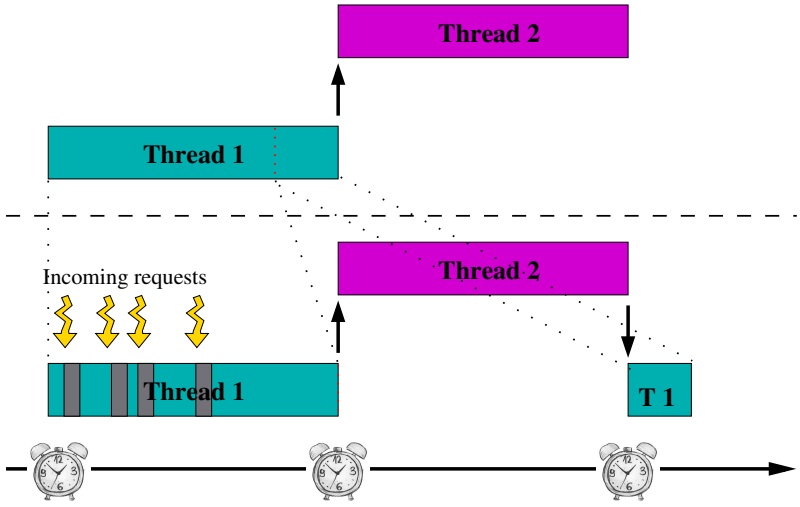


Figure 6.3: Normal (up) and curtailed (down) execution of a thread

The remaining issue to achieve this attack is memory access. In some open systems the volatile memory remains fully available. But some systems isolate the memory access to respective areas. Therefore this restriction does not allow a thread to get to the execution contexts. To successfully perform the attack, the isolation mechanism must be overcome. The next section illustrates how such a protection has been circumvented on a Java Card by the mean of a physical perturbation.

Once memory access is obtained, a full range of possibility is offered to the attacker. The control of the execution context of the thread gives access to its program counter, local variables and execution stack. Although access to these data obviously stands as a compromission of the system, an attacker has no guarantee that she will be able to take benefit of it. On the other hand, provided she has properly adjusted T_0 , well-chosen alterations would break almost any security operation. The complete attack scenario is depicted on Figure 6.4.

The attack potentially concentrates several issues which strongly depends on the kind of attacked system. But its consequences may be tragic for an application, even if the code has been proficiently secured. Furthermore this attack also underlines that the security of an application has a value only if the platform underneath is secured enough.

6.2.3 Practical Implementation on a Java Card

This section details the full attack scenario which has been put into practice on a recent device to illustrate the feasibility of this concept and outline its consequences.

6.2.3.1 Context of the Attack

The attacked platform. With regards to the different features involved in the attack, a device implementing the JC3.0 specifications appears to be a potential target. Indeed,

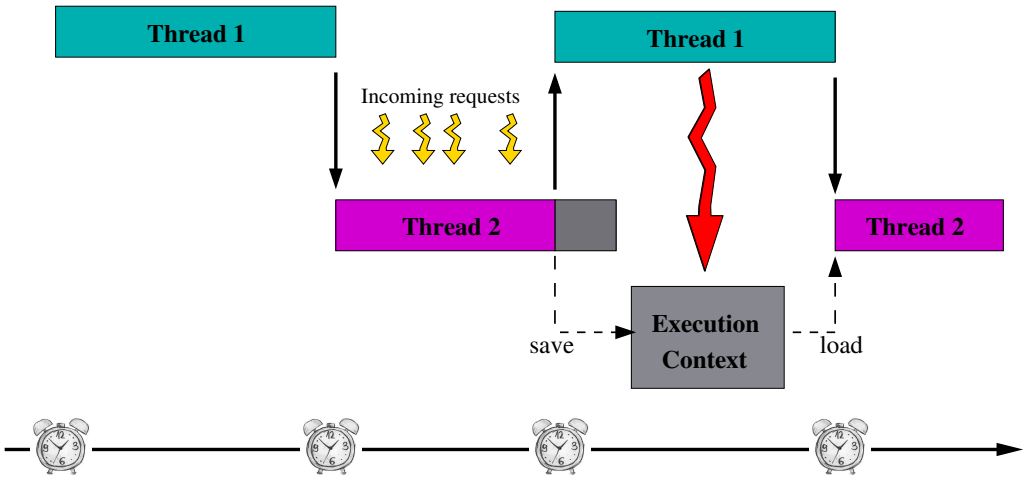


Figure 6.4: The complete attack scenario

it is a security device offering both multithreading and network communication support. Furthermore, such platform may allow post-issuance application loading, as long as the application is well-formed.

The target application. Consider in the remainder of this section an application $\mathcal{T}$ offering sensitive services. Access to those services requires an authentication, achieved through a signature. $\mathcal{T}$ then contains the lines of code given in Listing 6.3 (Java source code) and 6.4 (corresponding bytecode sequence).

Listing 6.3: Access control in $\mathcal{T}$ (source code)

```
if (sig.verify(inBuf,inOf,inLen,sigBuf,sigOf,sigLen) != true)
        accessDenied();
else
        accessGranted();
```

Listing 6.4: Access control in $\mathcal{T}$ (bytecode)

```
invokevirtual #4      // <javacard/security/Signature.verify>
iconst_1
if_icmpne 0x1C (+10)
aload_0               // <app/Target this>
invokespecial #5      // <app/Target.accessDenied>
goto 0x20 (+7)
aload_0               // <app/Target this>
invokespecial #6      // <app/Target.accessGranted>
```

Attack goal. The attack aims at gaining access to the sensitive services without producing a valid signature.

6.2.3.2 The Attack Concept Key Assumptions

The previous section that the success of the attack concept relies on the validity of a couple assumptions. This section details the validation of these assumptions on the attacked platform.

Loading the attack application. Loading application on the platform is not an obvious right for Java Card users. This capacity is generally limited by the knowledge of authentication keys through GlobalPlatform, as exposed in Chapter 3. However, assume that the attacker could load an application. This ability can have various origins :

- The load keys are known (this knowledge being either legitimate or not).
- One or several fault injection(s) may have led to a breach in the GP implementation on the card.

Loading and executing a malicious application $\mathcal{A}$ have two rationales. First it should permit the modification of the targeted thread's execution context. But it is also in charge of ensuring the expected thread scheduling scenario. Details of its implementation are given along the attack path.

How to access and corrupt the Java frame. The first challenge is to access the memory and to identify the execution context. Considering a JC3.0, the following elements (refer to [LY99] for a detailed description of Java *Threads* and *Runtime Areas*) are of interest :

- the Java program counter : the address of the currently executing instruction in the current method of the current frame.
- the stack of frames : a frame is pushed onto the stack when a method is invoked and is popped out when this method completes.
- in the frames : the local variables and the operand stack.

It is intended to reach these values by forging a fake byte array. For that matter, a fault injection is considered to cause a type confusion.

The fault attack. The particular fault attack considered in the scope of this work is similar to that described in Section 5.1. This attack allows to provoke a type confusion, and subsequently to forge an object's reference and content. In the context of the work presented here, this attack turns out to have a couple of advantages :

- A single physical attack of the device is required, a perturbation during the execution of a `checkcast` instruction for instance.
- Since forged references are persistent :
  - The fault injection can be the first step of the attack scenario.

– Once one perturbation has been successful, a failure in the following steps will not require to start again from scratch.

This particular laser-induced perturbation was successfully reiterated on the targeted platform.

Accessing the Java frame. Assume that execution contexts are saved in volatile memory on thread switching. The type confusion is used to forge a byte array in memory in order to access the execution context of $\mathcal{T}$. Also assume the internal representation of a Java array contains a pointer (say a 32-bit word) to its content in memory, as exposed in [HM09] and depicted within Fig. 6.5.

To build a fake array, the attacker only has then to set the "confused object" fields to appropriate values. Then, she can expect to be able to access the memory as if it was the content of the forged array. This process is illustrated in Figure 6.5.[2]

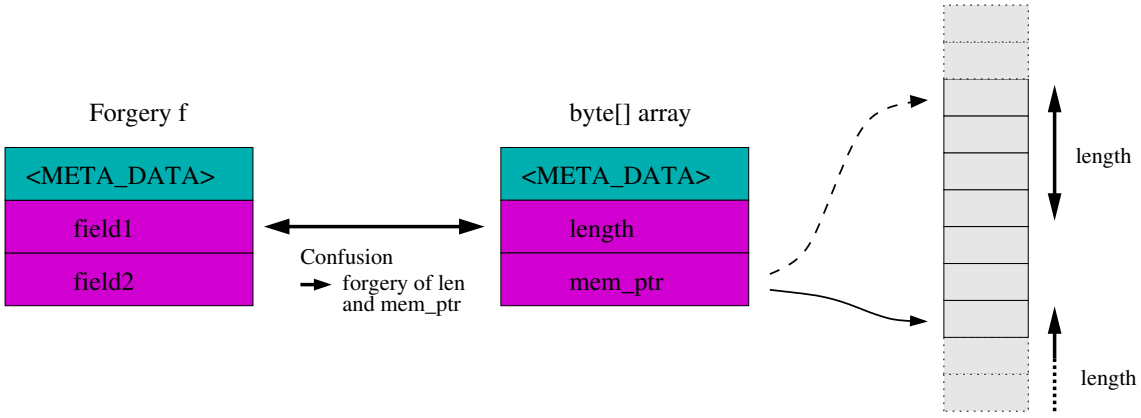


Figure 6.5: Confusion between instance of two classes in order to forge an array's address.

Once access to memory gained, the attacker needs to identify the frame within the forged array and to figure out its internal structure.

Finding and learning the structure of the Java frame. To locate the frame, she can take advantage of a straightforward linear memory allocation mechanism. According to the scheduling scenario of the attack concept, initializing a new array with obvious values when $\mathcal{A}$ is resumed permits to delimit the memory used by the targeted application. Furthermore, the attacker may have run a training session of the attack in order to learn the structure of frames on the platform. For that matter, she can build a target application that interrupts itself with easy-to-detect `short` values in local variables (`0x1903` for instance) and on the operand stack (`0x1902` for instance). The resulting dump array obtained when $\mathcal{A}$ is resumed is depicted in Figure 6.6.

[2] See Appendix B.1 for implementation details.

```
0x0000 :  55 55 55 55  55 55 XX XX  XX XX XX XX  XX XX XX XX
                              <proprietary data>
0x0030 :  XX 14 00 01  00 BA E2 01  00 E2 04 01  00 F2 45 00
0x0040 :  00 00 70 00  00 03 19 00  00 03 19 00  00 03 19 00
0x0050 :  00 03 19 00  00 03 19 00  00 03 19 00  00 03 19 00
0x0060 :  00 03 19 12  E1 53 12 00  12 E1 08 00  00 00 02 19
0x0070 :  00 00 02 19  00 00 02 19  00 00 02 19  00 00 02 19
0x0080 :  00 00 02 19  00 00 02 19  00 00 02 19  00 00 20 00
0x0090 :  00 00 49 00  XX XX XX XX  XX XX XX XX  XX XX XX XX
                              <proprietary data>
0x00D0 :  XX XX XX XX  XX XX XX XX  XX 55 55 55  55 55 55 55
0x00E0 :  55 55 55 55  55 55 55 55  55 55 55
```

Figure 6.6: Memory dump from the forged array.

She can then detect the frame and gain sufficient information on its structure :

- <number of local variables : nb_loc> <nb_loc * local variables>
- <maximum stack size : max_stk> <max_stk * operand values>
- <current top of stack>
- <jpc>

How *ping* flooding affects application execution. It was stated in Section 6.2.1.2 that some incoming requests do not require the attention of the JCRE. An Internet Control Message Protocol (ICMP) *echo* request (a ping) is a typical example of such a request. The claim is that when a ping request is incoming, the processor handles it whereas in the meantime the scheduler's timer is still running. One can then manage to shorten a thread's execution as she pleases, the number of instructions actually executed within a time slice being reduced. To validate this claim, a thread incrementing a counter was run on the attacked platform. Figure 6.7 presents the value reached by the counter after a given amount of time against the number of pings sent in the same time. This proves that the number of instructions executed within the thread is reduced when the system is flooded with pings, since the value reached by the counter is representative of the number of instructions executed[3]

This technique could be assimilated to a well-known attack in the network security field : *ping flooding* [HR06, LMS+11]. Ping flooding usually aims at consuming the bandwidth of the targeted system in order to provoke a *Denial of Service* (DoS). The approach described here is different as the aim is here to consume the time allocated to the targeted thread in order to curtail it.

6.2.3.3 The Practical Attack

The attack is divided into three steps detailed within this section.

[3]Implementation details are given in Appendix B.2.

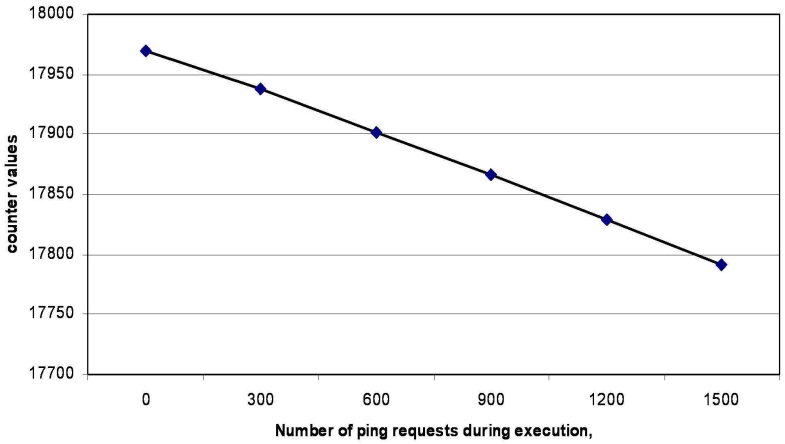


Figure 6.7: Influence of communication on instructions execution.

- In the first step, the preliminary work is done to ensure both the access to $\mathcal{T}$'s frame and the scheduling scenario;
- In the second step,the "breakpoint hitting" is forced using I/O flooding;
- In the third step, the fake array is used by the malicious application to corrupt $\mathcal{T}$'s frame.

Preliminaries. With regards to the global illustration of the attack concept, this step corresponds to the first segment of the "evil" thread's execution (T1(1) in Figure 6.4). The aim of this step is to procure a way to access the memory where the execution context of $\mathcal{T}$ will be stored. This is achieved as presented in Section 6.2.3.2.

To ensure the predicted thread scheduling scenario, the application only has to start a new thread, and force its interruption for a certain amount of time (via the `Thread.sleep()` method). Within that time, $\mathcal{T}$ is launched in a new thread. On the next thread switching, $\mathcal{A}$'s thread will then become active again. One can note that even if another thread is executed between $\mathcal{T}$'s and $\mathcal{A}$'s the attack still works. One can then focus on $\mathcal{T}$'s execution and when to force its interruption.

Setting the breakpoint. The aim of this step is to force a thread switching at a precise point during $\mathcal{T}$'s execution. It corresponds to the first segment of the targeted thread's execution of the attack concept illustration (T2(1) in Figure 6.4). The challenge at this step is to "synchronize" the pings and the thread's execution. Working on a smart card, the power consumption analysis can again reveal a strong ally at this step. Actually, one can monitor bytecode instruction execution through the power consumption of the card (as stated in [VWG07]). Therefore, the exact knowledge of the code does not appear necessary to achieve the attack.

The attacked platform comes within a USB smart card connector and communicates as an Ethernet Emulation Model (EEM) device according to the specification [USB05]. The first task is then to adapt the power consumption acquisition module to monitor power consumption behind the USB smart card connector where the Java Card is plugged (*cf.* Fig. 6.8).

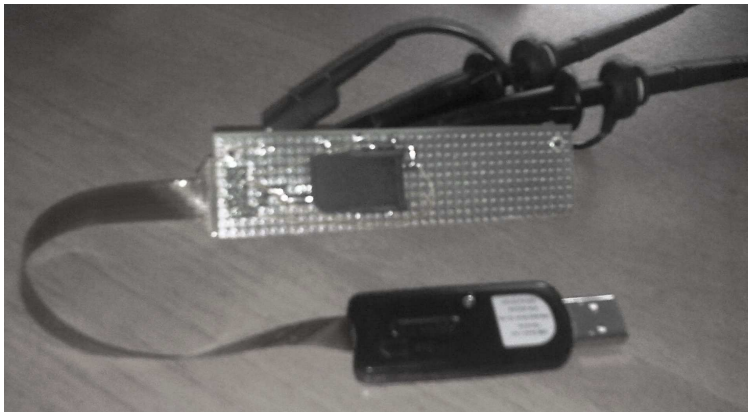


Figure 6.8: USB smart card acquisition module.

The ping flooding of $\mathcal{T}$ can then be performed and monitored.

Figure 6.9 shows the power traces of $\mathcal{T}$'s execution. On the first power trace, the signature verification is easily identified. The following traces depicts the same execution with an increasing number of ping requests (the numerous peaks on the traces). As visible, the cryptographic operation is executed more or less shifted depending on the number of pings received during the thread's execution. However, one can notice that no I/O flooding led to the interruption of this apparently atomic operation.

Based on experimentations, an average sequence of 37 ping requests during the execution of $\mathcal{T}$ causes its interruption after the `verify` method returns but before the execution of the conditional branching. Actually other "breakpoints" may also allow an attack.

The previous section has given a way to read/write the volatile memory. This section has exposed how to set the so-called breakpoint within the attacked application $\mathcal{T}$. To complete the attack, one must modify $\mathcal{T}$'s frame in order to bypass its security.

Corruption of the Java frame From application $\mathcal{A}$, one can now corrupt $\mathcal{T}$'s frame. The attacker is then literally spoilt for choice in order to bypass the application's security :

- Set the jpc to a given value in order to modify the execution flow,
- Assign given values to references or integral values in the operand stack to have a method executed on the wrong object or with wrong parameters, or return a wrong value,

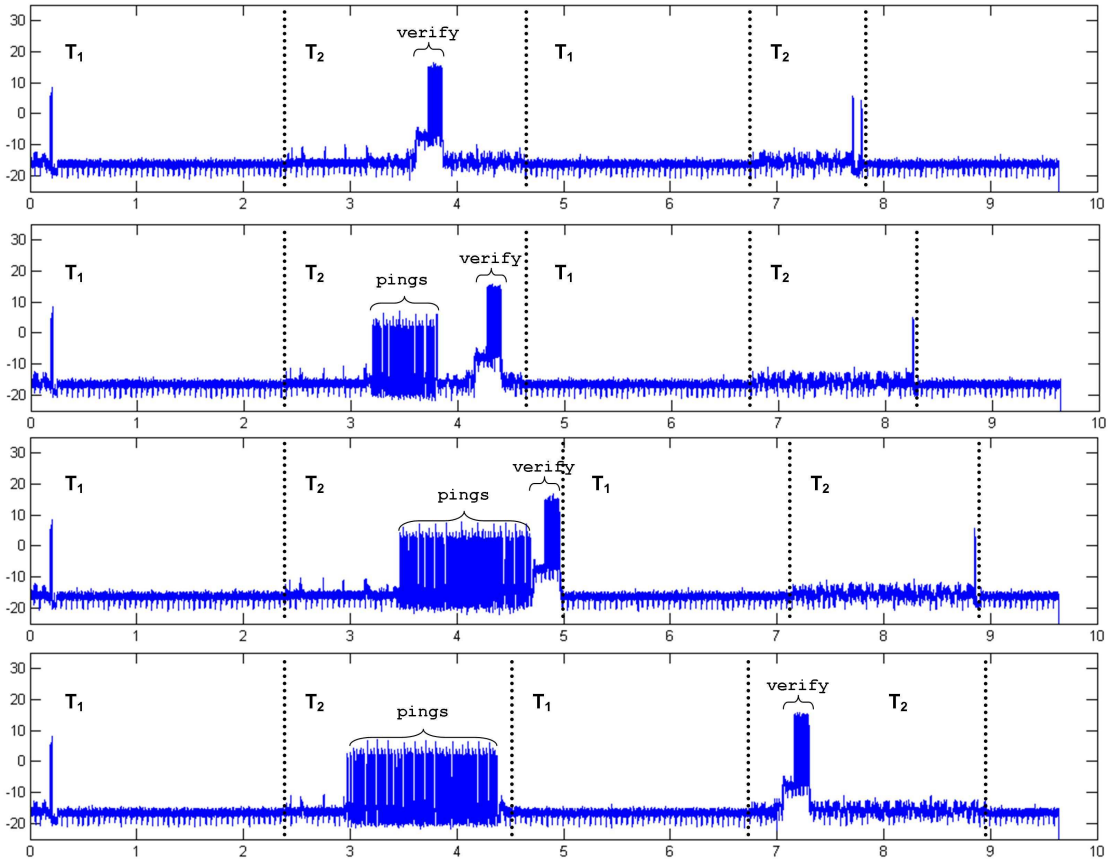


Figure 6.9: Execution of the two threads with various number of ping requests (respectively 0, 10, 30 and 40). T_1 and T_2 refer respectively to the attacker's and the target thread.

- Assign given values to references or integral values in the set of local variables, with the same consequences.

With regards to the current state of the art of fault injection, these possibilities are quite outstanding. In a manner of speaking, completing the three previous steps enhances tremendously the initial fault model. The following presents one of the numerous way to render the security check of $\mathcal{T}$ useless by modifying the value of the Java Program Counter.

Modification of the jpc. As expected, the signature verification has failed and the conditional branching at line 0x12 leads the execution flow in the "`accessDenied` branch". Because of the flooding, the so-called breakpoint is "hit" at line `0x1D` (`invokespecial #6 <accessDenied>`).

$\mathcal{A}$ is then resumed and the attacker can read $\mathcal{T}$'s frame and identify the jpc value (Figure 6.10).

An easy way to overcome the signature invalidity is then to modify the jpc in order to jump "manually" in the desired branch. That is to say to say to modify the

```
0x0040 :  XX XX XX XX  XX XX XX XX  XX XX 08 00  01 00 38 FC
0x0050 :  01 00 78 F2  01 00 D2 FE  00 00 05 00  00 00 00 02
0x0060 :  01 00 46 F2  00 00 00 00  00 00 01 00  E7 D2 66 00
0x0070 :  0B D4 07 00  01 00 38 FC  00 00 01 00  00 00 05 00
0x0080 :  00 00 00 02  01 00 46 F2  00 00 00 00  00 00 01 00
0x0090 :  00 00 18 00  00 00 1D 00  XX XX XX XX  XX XX XX XX
```

Figure 6.10: Memory dump from the forged array.

jpc value from `0x1D` to `0x16`. This is done by a mere affectation in the forged array : `ac.array[jpc_offset] = 0x16;`

When the scheduler switches back to $\mathcal{T}$, its execution continues according to the frame. That is to say at the just set offset. The next executed instruction will then be the invoke of the `accessGranted` method. As a consequence, the attacker gained access to the privileged method, although she did not have the private key to produce a valid signature.

Depending on the moment when the ping flooding force the interruption, similar results were obtained by modifying operands on the stack and local variables. What emerges from these different options is that a certain inaccuracy in the ping flooding phase is tolerable. Indeed, depending on the "breakpoint" location, an attacker with a good knowledge of the targeted application will often (not to say always) find a path to meet her objective.

This proof of concept demonstrates that such a threat should be taken in consideration when addressing the security of an embedded platform.

6.3 Disruption of `Exception`-Related Mechanisms

The Java Card 3 *Connected Edition* has brought the Java Card platform really close to standard ones, introducing notions and facilities related to standard Java applications and web services in particular. The literature on the security of Java and web services is relatively vast and several studies, security breaches and security guidelines have been published [DFW96, Lad97, Pri, Las02, CM05, HJMP10, LMS+11, Oak01, The12a, The12b, MS98, MF00, CV01, GM05, LL05]. It appears then all the more interesting to reconsider these studies in the Java Card context. This section focuses on the security of the exception-related mechanisms of the Java Card platform. This concern comes from the observation that attackers often try to circumvent an exception throwing, but rarely to take advantage of it.

6.3.1 Exceptions in Java Card Platforms

This section aims at introducing the notion of exception, the evolution of its use in a program and the related mechanisms on Java Card platforms, namely exception throwing

and exception handling. Although it may be relatively specific concerning certain properties, it does not intend to give an exhaustive description of the life and death of exceptions. Readers wishing to learn more should definitely refer to [Che00, §6], [GJSB05a, §11] and [LY99, §2.16].

6.3.1.1 The Role of Exceptions

As previously stated, exceptions are initially meant to handle errors during the execution of a program. In the following, it is shown that if the original purpose is still in use today, exceptions have been totally integrated in application's programming and are now widely used for more than "just" handling errors.

Handling abnormal conditions. On one hand, there are the traditional usages of exceptions, that is to say error handling. In this scope, an exception is created when an abnormal behaviour is detected, when the execution of a program is deemed unsuccessful, or when the system detects that pursuing the execution will lead it to halt abruptly for instance. The exception handler is then in charge of handling the abnormal conditions that have led to the exception raising, and in the worst case to properly halt the system.

Handling particular conditions. On the other hand, in certain modern programs, exceptions are used as a way to break the control flow of an application and not necessarily because an abnormal condition has occurred. For instance, an exception may be raised by a program when a given condition is satisfied or not, regardless of the potential consequences of this condition. The aim of such exception is only to force a jump and execute specific lines of code that are defined as the exception handler.

Good practices. Several references [LMS+11, Oak01, The12a, The12b] detail the importance of properly handling any exception that might occur during the execution of a program. Concerning the Java language, it is particularly important to segregate the different exception types (classes) in different *catch*-blocks in order to handle each and every exception type in an appropriate way. Furthermore, it is also generally advised not to transmit too much information within the exception, as it could be useful to an attacker[4].

6.3.1.2 The Syntax of Exception Handling

Java, as several programming languages, defines a syntax associated to exception handling. For that purpose, the Java language allows the definition of *try-catch*-blocks.

As described in Listing 6.5, the programmer has to write the code that is likely to raise an exception inside a *try*-block and the code that will be executed when a certain type of exception is raised in different *catch*-blocks. In addition, the *finally*-block allows to define a code portion that will be executed whether an exception was raised or not, caught or not.

[4] The typical example of such a situation is that of an attacker sending SQL requests to a database server and gaining information on the structure of tables thanks to the message contained in the returned exceptions.

Listing 6.5: Exception syntax in Java and Java Card

```
try {
     ...     // Code operating a sensitive process
     ...     // that will raise an exception if
     ...     // deemed unsuccessful.
} catch (ExceptionType1 et1) {
     ...     // The operation has failed in a
     ...     // specific way, handle it accordingly.
} catch (ExceptionType2 et2) {
     ...     // The operation has failed in another
     ...     // specific way, handle it accordingly.
} finally {
     ...     // Code executed whether an exception
     ...     // has been thrown or not, caught or not.
}
```

6.3.1.3 The Exception Handler Table in Java Card Binaries

In the binary format, either .CLASS files for standard Java binaries or .CAP files for Java Card binaries, exception handlers are represented for each method into a so-called *handler table*. In a .CAP file, for instance, this table is represented in the method component as specified in [Sun06c] and described below:

```
method_component {
    u1 tag
    u2 size
    u1 handler_count
    exception_handler_info exception_handlers[handler_count]
    method_info methods[]
}

exception_handler_info {
    u2 start_offset
    u2 bitfield {
        bit[1] stop_bit
        bit[15] active_length
    }
    u2 handler_offset
    u2 catch_type_index
}
```

According to the specifications, the handlers should be sorted according to their starting offset in the method's bytecode. For each handler, the exception handler table specifies the start- and end-offset of the handler in the method's bytecode array, as well as the subtype of *Throwable* it handles.

Therefore, when an exception is thrown at some point during the execution of an application, the JCVM is responsible for searching the appropriate exception handler in the table, if any. The exception handler searching is typically done as presented in Algorithm 1.

Algorithm 1: EXCEPTION HANDLER SEARCHING

```
input  : E the exception being handled
input  : H the exception handler table
output : -
1  while more handler left in H do
2      handler ← next handler;
3      if jpc is covered by handler offsets then
4          if handler match E's type then
5              jump to handler;
6              end loop;
7          end
8      end
9  end
10 forward exception to previous frame;
```

6.3.2 Software, Fault and Combined Attacks: On the Security of Exceptions and Exception Handling

Most of time, attackers only care about exceptions because they wish to avoid the particular conditions leading to an exception throwing. This section starts by describing two previous works taking advantage of an exception throwing to mount an attack. Subsequently, results on the security of exception-related mechanisms are exposed. These consist in the description of new attacks grounded on both exception handling and exception throwing.

6.3.2.1 Previous Works

The case of exceptions has not been much investigated in the literature related to Java Card security. However Barbu *et al.* presented some work at CARDIS 2010 [BTG10]. On the other hand, the standard Java security field has shown much more interest for exceptions [DFW96, Lad97, Pri, Las02, CM05, HJMP10, LMS+11, Oak01, The12a, The12b]. The work of *Dean et al.* presented as soon as 1996 at the IEEE Symposium on Security and Privacy [DFW96] is briefly described hereafter.

Using an Exception to Attack a Java Card. In [BTG10], the authors propose an attack on a Java Card 3 platform allowing to read and modify the bytecode array of an on-card application. This is achieved thanks to a type confusion involving the newly

introduced *java.lang.Class* class of the Java Card 3 Application Programming Interface [Sun09b]. The interest of this work lies in the manner of provoking the type confusion through a laser pulse fault injection [GT04, BECN+06a]. Indeed the authors explain that the *java.lang.ClassCastException* thrown by the virtual machine when an incorrect type-casting is executed can be used by an attacker. The aim is then to detect the execution of the exception throwing by monitoring the power consumption of the card in order to determine the precise moment when to apply the laser pulse to disrupt the virtual machine's execution during its type checking.

Using an Exception against a Java Client. In [DFW96], the authors describe how the exception throwing can lead to security breaches allowing a privilege escalation on the client-side through a web browser (Netscape Navigator for instance) executing a Java applet. The article is based on the fact that the incomplete execution of a class constructor may lead to an object instance partially initialized. In particular, they exhibit the example of the abstract class *ClassLoader* from Sun's Java Runtime Environment (JRE). They provide a custom class loader extending this abstract class (cf. Listing 6.6).

Listing 6.6: The incomplete class loader

```
class CL extends ClassLoader {
    CL() {
        try { super(); }
        catch (Exception e) {}
    }
}
```

The partial initialization of the class loader leads to a type confusion and eventually to a privilege elevation for the attacker's application. The attack described in this work is then completely based on the exception throwing. This breach was then closed with updates of both the Netscape Navigator and the Java Development Kit, respectively with versions 2.02 and 1.02.

6.3.2.2 Attacks on the Exception Handler Research

In the following, several ways are proposed for an attacker to take benefits from the Algorithm 1, searching the proper handler for a given exception:

- first a new attack based on a fault injection that can be either combined with a malicious application or not depending on the attack scenario;
- then a software-only attack using an incorrect handler table within a .CAP file.

Jumping in the wrong exception handler. The attack presented here consists in a fault injection targeting one of the conditional branch instructions of Algorithm 1 (*i.e.* lines 3 or 4). The physical perturbation consists in jumping a given (set of) instruction(s) by disrupting the code fetching in the card's Non-Volatile Memory, where the

application's code is stored. Such a fault model is widely accepted in the literature [SA02, GT04, BECN+06a].

Provided the attacker meets success with the fault injection, she can then force the jump in an exception handler that was meant for another type of exception, or for the same type of exception but at a different time (*i.e.* covering another part of the code). The following code sample (Listing 6.7) is definitely far-fetched but is a perfect illustration of the type of code likely to be targeted by such an attack.

Listing 6.7: Exception syntax

```
try {
    // MyOwnPIN.check method throws a TryAgainException
    // when an invalid PIN is submitted.
    if (myOwnPin.check(submitted, 0, 4)) {
        executePrivilegeMethod();
    }
} catch (NullPointerException npe) {
    // myOwnPin is not yet initialized, do it and return
    myOwnPin = new MyOwnPIN(TRY_LIMIT, PIN_SIZE);
    myOwnPin.update(PIN_ARRAY, PIN_OFFSET, PIN_LENGTH);
    return PIN_NOT_INIT;
} catch (TryAgainException tae) {
    processInvalidPINSubmittedEvent();
    return INVALID_PIN;
}
```

In this particular case, a successful perturbation allows to re-initialize the PIN object, by executing the code in the *catch(NullPointerException)*-block. An attacker reaching a good fault injection repeatability is then likely to brute-force the PIN, although it is supposed to be protected by a counter limiting the number of tries and decremented within the *MyOwnPIN.check* method. Note that in the general case, by forcing a jump in the wrong handler, the attacker always creates a type confusion between the actual exception type and the exception type that is caught. The consequences of the execution of the wrong *catch*-block are on the other hand totally dependent on the application.

Handler table corruption. As described in Section 6.3.1.3, the .CAP file contains a handler table defining where to jump when a given exception occurs at a given offset in the bytecode array.

When an exception is thrown, Algorithm 1 leads to jump at the offset defined within the handler table. It is therefore obvious that corrupting the handler table will eventually result in a jump at an incorrect offset in the bytecode array. However, before being able to execute the application on-card, the attacker would most likely have to make it pass the bytecode verification. It is then necessary to study how the bytecode verifier ensures the correctness of the handler table. The rules used by the verifier can be deduced from the rules defined in the JCVM specification. These rules are the following:

- handlers are sorted according to their *start-offset*,
- handlers do not "partially intersect", *i.e.* they are either disjoint or nested,
- handlers point to valid offsets in the current method,
- the stop-bit defines when to stop searching for other handlers.

Several modifications of the handler table were then tested versus both the Java Card converter and the bytecode verifier present in the JCDK 3.0.4. It turns out that only one manipulation was not rejected. This manipulation consists in the manipulation of the end-offset of a *try*-block so that it englobes the handler of the associated *finally*-block. This is depicted in the .JCA file obtained from the conversion of the applet (after the .CLASS file has been modified) in Listing 6.8. Note that a modification of the *bit-stop* of the finally handler was also necessary.

Listing 6.8: JCA file after modification

```
.method public testException()V 8 {
     .stack 2; .locals 2;
 L0: aconst_null;
     astore_1;
 L1: aload_1;
     aload_1;
     invokevirtual 11;  // equals(Ljava/lang/Object;)Z
     pop;
     new 12;            // java/lang/Object
     dup;
     invokespecial 3;   // java/lang/Object.<init>()V
     astore_1;
     goto L4;
 L2: astore_2;          // local variable created by the compiler
                        // to store the exception.
 L3: new 12;            // java/lang/Object
     dup;
     invokespecial 3;   // java/lang/Object.<init>()V
     astore_1;
     aload_2;
     athrow;
 L4: return;
 .exceptionTable {
     // start_block end_block handler_block catch_type_index
     L1 L4 L2 0;  // handler covers finally-block's athrow!
     L2 L3 L2 0;
 }
}
```

Consequently, when an exception is thrown within the *try*-block, it is handled by the *finally*-block which in turn re-throws the exception. This exception is subsequently handled by the same *finally*-block again, *etc*... As a matter of fact, the applet is stuck in an infinite exception *throw-and-catch* loop. Indeed a binary file manipulation can allow to break the

infinite loop by using a counter incremented in the loop and conditioning the exception throwing. Also, the *finally*-block can be manipulated in order to use the local variable created by the compiler to store the exception before throwing it at the end of the block. This last point would be very tricky to handle by the bytecode verifier, although no explicit breach was found.

6.3.2.3 Attacks on Exception Throwing

This section focuses on the exception throwing itself. Two different kind of attacks can then be isolated, whether the attacker intends to force an ill-behaviour of this mechanism or simply to bypass it. The following introduces attacks in both categories.

Throwing a Non-Throwable. In [BDH11], the authors studied the possible impacts of a perturbation of the values pushed onto the operand stack of a Java Card platform. Some of their use cases are based on the attacker's instantiating as many objects of a given class $\mathcal{C}$ as possible in order to enhance the probability that a random error affecting an object's reference transform it into one of the reference of an instance of class $\mathcal{C}$.

An issue was not treated in their article: the perturbation of an thrown exception's reference. The possible consequences are twofold depending on the nature of class $\mathcal{C}$:

1. $\mathcal{C}$ extends the API's class *Throwable*. Then the execution will continue in the wrong exception handler, provoking, for sure, a type confusion between the two exception classes and potentially bypassing some exception-dependent operations, including security-related ones.

2. $\mathcal{C}$ does not extend *Throwable*. In this case, the JCVM should not find an exception handler matching the type of the exception. The exception should therefore not be caught and cause the system to halt. However, considering a malicious application, the attacker may have created a *finally*-block taking this possibility into account, which would at least provoke a type confusion between two object instances.

However, in the latter case, even jumping in the *finally*-block is not obvious on all platforms. The *finally*-block being meant to handle any kind of exceptions, it is associated in the application's binary file to a type referred to as *ANY* in the specifications. Yet, every platform is responsible for testing that the object thrown can be casted to the type *Throwable* before jumping in the *finally* handler.

Prevent Exception Throwing. Another interesting option from an attacker's point of view is simply to prevent the throwing of an exception. In the context of Java Cards, this can be achieved on two different circonstances, by two different means.

The first one concerns the case where the exception is thrown by the JCRE. The typical example for such a situation is when an application invokes a virtual method on an object that turns out to be null. As per the specifications, the JCRE then throws a *NullPointerException*. The code responsible for the exception raising is obviously part

of the JCRE/JCVM implementation, one can therefore expect it to be written in native language (i.e. in the card's assembly language). Therefore, an attacker willing to skip the exception throwing will have to disrupt the execution of native code, which is generally done by perturbing the fetch of the code either in the ROM or the NVM.

The second one concerns the case where the exception is thrown by a Java Card application. In this case, it is the bytecode instruction *athrow* which is responsible for the exception throwing. An attacker can then work two different attack angles. The first one is the same as previously stated. Since the *athrow* is also a part of the JCVM implementation, a similar attack on the appropriate code fetching shall allow to bypass the execution throwing. The second angle consists in an attack of the bytecode instruction reading in the application's bytecode array. This bytecode is basically a byte read and processed by the JCVM's interpreter. A fault attacking sticking this byte to 0 will then force the execution of the *nop* instruction, on platforms using the standard instruction set, and therefore bypass the exception throwing. This is one of the basic examples introducing mutant application in [SICL09].

The Incomplete Object Initialization on Java Card The following explores the possibility to somehow adapt the incomplete object initialization attack described in Section 6.3.2.1 on a Java Card. The point is that according to the 2^{nd} edition of the Java Virtual Machine specification, the class file verification process should prevent such a situation from occurring, as it specifies:

> "A valid instruction sequence must not have an uninitialized object on the operand stack or in a local variable during a backwards branch, or in a local variable in code protected by an exception handler or a finally clause." [LY99, §4.9.4]

In addition, the 3^{rd} edition of the Java Language specification specifies that during an object initialization calling a superclass constructor:

> "If that constructor invocation completes abruptly, then this procedure [object initialization] completes abruptly for the same reason." [GJSB05a, §12.5]

The possibility to load an application containing the code depicted in Listing 6.6 is then subject to question on recent platforms.

Is it possible to load such an application on a Java Card?

Consider then the following constructor method, for a class extending a super-class whose constructor throws a particular exception, say *MyException* in this case (Listing 6.9).

The fact is that trying to compile this code with the Java compiler provided with the Java Development Kit version 1.6.0_18-b07 result in the following error: `'call to super must be first statement in constructor'`. Assume then that this error is due to the restrictions quoted above since the call to the superclass's constructor is indeed the first statement in the extending constructor. Getting a valid class actually representing this constructor requires then the compilation of a slightly different

Listing 6.9: Potential incomplete initialization

```
    try {
        super();          // throws MyException
        authRequired = true;
    }
    catch (MyException me) {
        // Initialization is not complete...
        // authRequired = false, the default value for boolean.
    }
```

piece of code and a little .CLASS file tweaking. The content of the two .CLASS files is given in Listings 6.10 and 6.11.

Listing 6.10: Original constructor class

```
public void <init>() throws MyException {
    aload_0
    invokespecial void MySuperClass.<init>()
try-block_start(myapp.MyException)_4:
    aload_0
    invokevirtual void MyClass.notSuper()
    aload_0
    iconst_1
    putfield boolean MyClass.authRequired
try-block_end(myapp.MyException)_13:
    goto label_17
exception_handler(myapp.MyException)_16:
    astore_1
label_17:
    return
}
```

Listing 6.11: Tweaked constructor class

```
public void <init>() {
try-block_start(myapp.MyException)_0:
    aload_0
    invokevirtual void MySuperClass.<init>()
    aload_0
    iconst_1
    putfield boolean MyClass.authRequired
try-block_end(myapp.MyException)_9:
    goto label_13
exception_handler(myapp.MyException)_12:
    astore_1
label_13:
    return
}
```

Afterward, both the Java Card converter and bytecode verifier that are provided with the most recent Oracle's development kit (Java Card Classic Edition 3.0.4 Development Kit) were successfully passed. These two operations yields a verified .CAP file, ready to be loaded and installed on-card, proving that the verifier does not reject classes with potential incompletely initialized objects. One can therefore be pretty confident on the success of the loading of an application with the same behaviour on a Java Card 3 Connected Edition endowed with an on-card bytecode verifier.

6.3.3 Consequences and Analysis

The previous section has shown that several possibilities are offered to a potential attacker to take advantage of the exception-related mechanisms of the system.

The consequences of the potential attacks described in the previous section are various indeed. Concerning the incomplete object initialization, the assets that might be targeted by such attacks are still to be precisely identified. Unlike standard Java, the Java Card 3 *Connected Edition* does not support user-defined class loaders, nor the *Object.finalize()* method evoked in [HJMP10] to extend *Dean et al.*'s work. Consequently, this seminal work cannot be transferred in the Java Card world as is. Furthermore, applications taking advantage of all the facilities offered by the Java Card 3 *Connected Edition* standard are not really widespread yet. However the example of the original attack is explicit and should encourage every developer to keep this threat in mind.

Regarding the other attacks introduced here, the consequences can be divided into two categories. On one hand, the consequences of a type confusion covers the access to private fields or to data out of the bounds of an array, the jump of access control checks, or even self-modifying applications in certain contexts. Type confusion is indeed the core of several attacks against Java and Java Card platforms [GA03, Wit03, MP08, SICL09][BTG10]. On the other hand, the execution flow disruption has been less studied but can lead to critical consequences. First, since it can be used to provoke a type confusion, the same results can be expected. Second, preventing an exception from being thrown can have even more dangerous consequences. Consider for instance the case of an access across the application firewall enforced by the JCRE that would not throw a *SecurityException*, although it has been duely detected, which is an example given in [VF10].

Chapter 7

Security Analysis of the Application Isolation Mechanism

Contents

7.1 Questioning the Code Isolation by Abusing Class Loaders 128
7.1.1 Context of the Attack 128
7.1.2 Impossible Call to Invisible Methods 129
7.1.3 Calling Invisible Methods through Type Confusion 129
7.2 Circumventing the Application Firewall with Application Replay . . . 131
7.2.1 Java Card Reference Prediction 131
7.2.2 Application Replay to Circumvent the Application Firewall . . . 134
7.3 Abuse of Global Arrays 138
7.3.1 Precisions on the APDU Buffer and Targeted Platforms 138
7.3.2 The APDU Buffer Storage Attacks 141
7.3.3 Case Study 142

For every multiapplicative platform, it is mandatory to ensure a strict isolation between the various hosted applications, both regarding their code and the data they manipulate. As exposed in Chapter 3 Java Card platforms are no exception to this rule. This chapter exposes the results of the security analysis of Java Card's application isolation mechanisms.

First an attack against the code isolation that was presented at E-SMART'09 [Bar09] and targets more particularly the code isolation mechanism through the use of SIOs (Shareable Interface Objects) is exposed in Section 7.1.

Section 7.2 explores the consequence of the introduction of a true garbage collection mechanism on the application firewall The results of this work led with Philippe Hoogvorst and Guillaume Duc were first presented at E-SMART'11 [BHD11] before an extended version was published in the proceedings of ESSOS 2012 [BHD12a].

Finally Section 7.3 depicts potential abuse of global arrays and more particularly of the Application Data Unit Protocol (APDU) buffer. This work led with Christophe Giraud and Vincent Guerin shows that the developer must always take into account that the APDU buffer can be compromised at any time and not only during communication with the card reader. It has been published in the proceedings of SEC 2012 [BGG12].

7.1 Questioning the Code Isolation by Abusing Class Loaders

This section expose a possible exploitation of the fault injection against the `checkcast` instruction described in Chapter 5. Indeed, the target of the attack is the Class Loading Delegation Hierarchy (CLDH) enforcing the code isolation required by the Java Card 3 Connected Edition specifications.

7.1.1 Context of the Attack

In the scope of the attack presented here, the code sample proposed in the Java Card Development Kit [Ora] to illustrate the use of SIOs is considered. The service offered in this sample application consists in retrieving a shared integer value. The code of the server application (i.e. the shared interface and the class implementing this interface) is given in Listing 7.1.

Listing 7.1: The server application

```
package sioservice;

public interface SIOInt extends Shareable {
  public int getValue();
}

public class SIOImpl implements SIOInt {
  private int value;

  public int getValue() {
    return value;
  }

  private void setValue(int value) {
    this.value = value;
  }
}
```

Therefore, a client application will be able to retrieve the integer value by calling the shared method `getValue`. On the other hand, only the server is able of modifying this value, either by directly assigning it or by calling the `setValue` method. These restrictions are enforced by the use of the `private` access modifier, and by the fact that only the `getValue` method is exposed in the shared interface `SIOInt`.

Figures 7.1 and 7.2 illustrate the web pages accessible on the client side.



Figure 7.1: SIO client at initialisation `http://smartcard/sioclient/index`.

7.1.2 Impossible Call to Invisible Methods

As previously described in Section 3.2.2.1, dynamic class loading on Java Card 3 Connected Edition relies on the CLDH. In this case, the hierarchy can be illustrated by Figure 7.3.

According to the CLDH specification [Sun09e, §6.7], if ever the client application calls the `SIOImpl.setValue` method, a `ClassNotFoundException` should be thrown. Anyway, this should not even be possible since such calls should be rejected by both the compiler and the bytecode verifier.

7.1.3 Calling Invisible Methods through Type Confusion

Now consider a malicious client application defining its own SIOImpl class in every respect similar to that defined by the server application, as described in Listing 7.2.

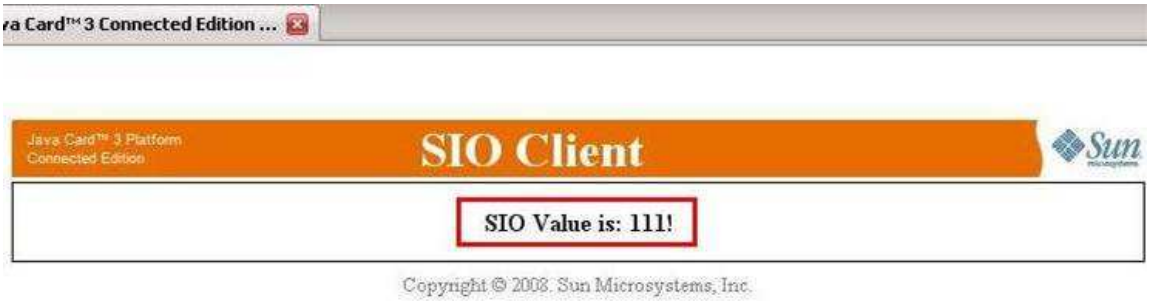


Figure 7.2: SIO client get value `http://smartcard/sioclient/getvalue`.

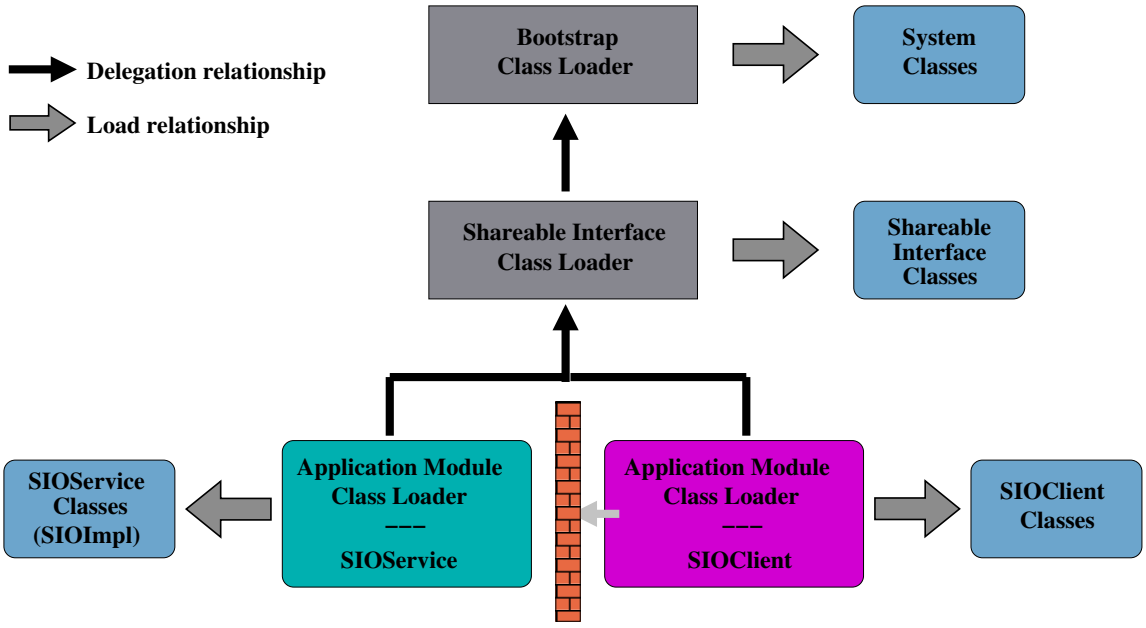


Figure 7.3: Illustration of the class loading delegation hierarchy.

Listing 7.2: The server application

```
package sioservice;

public class SIOImpl {
  private int value;

  public int getValue() {
    return value;
  }

  private void setValue(int value) {
    this.value = value;
  }
}
```

In addition, consider the malicious client has been able to provoke a type confusion between an instance of its own `SIOImpl` class and the instance of the server-defined `SIOImpl` class returned by the `ServiceRegistry`.

Then depending on the class loading mechanism a call to the client's (*i.e.* the attacker's) `setValue(int value)` on the server's `SIOImpl` instance, may result in a call to the `setValue(int value)` method implemented by the server. Yet since this method is not declared as *shared* the application firewall must forbid the call to the virtual method. To actually update the shared integral value within the SIO, the attacker must then also disrupt the application firewall, as exposed in [VF10].

7.2 Circumventing the Application Firewall with Application Replay

This section focuses on one of the new features introduced by the Java Card 3.0 specifications: a true automatic garbage collection mechanism. In the following the concept of reference prediction is introduced. In addition, this section shows how an attacker with fault injection capacity could, under certain assumptions, use this concept to circumvent the application firewall through a so-called replay attack.

7.2.1 Java Card Reference Prediction

This section introduce the notions of Java reference and garbage collection and state the assumptions under which this work is based. Finally, the process put into practice to achieve this prediction on the tested platforms is described.

7.2.1.1 Reference Assignment

On object instantiation, the JCVM is then responsible for allocating the memory to store this object and assigning it a Java reference, possibly the allocated memory address or a value abstracting this address. Regardless of its exact implementation, the remainder assumes that these Java references are assigned following a straightforward linear process. That is to say, the next reference to be assigned is the smallest reference that is not already assigned.

Formally, with $(r_i)_{1 \leq i \leq n}$ the previously allocated references, a new reference r_{n+1} is allocated such that:

$$r_{n+1} = \min\{r_i \text{ s.t. } \forall\, r_j < r_i,\, r_j \text{ is used}\} \tag{7.1}$$

This assumption is not very restrictive since it was successfully tested on different cards from different manufacturers and different versions of Java Card with the method given in Section 7.2.1.3.

For the sake of simplicity it is assumed in the remainder of this section that a reference is a value abstracting the physical address of an object. Otherwise, the method would need to be adapted, mainly taking into account the size of each object instance.

7.2.1.2 Garbage Collection

The principle of garbage collection is not a novelty, even in the Java Card context. Indeed, Java Card 2.2 proposes (optionally) a memory reclaiming process through the method `JCSystem.requestObjectDeletion()` [Sun02a]. However, this method only schedules the object deletion service prior to the next invocation of the `Applet.process()` method. That is to say, unreferenced objects are not actually deleted on the method's call.

The real novelty in the latest version of the Java Card standard is that garbage collection is automatically triggered when memory space becomes insufficient or on specific event such as card reset for instance. Furthermore, the `System.gc()` [Sun09b] method can be called at any time within an application and runs the garbage collector. Unlike the `JCSystem.requestObjectDeletion()` method, when control returns from the `System.gc()` method, the garbage collector should have actually been executed and reclaimed unused memory.

Not to be dependent on any particular garbage collection mechanism implementation the following only assumes that the garbage collector ensures that it will reclaim the memory used by objects that are not accessible anymore (unreferenced). The important point to bare in mind is rather the evolution of the Java Card specifications regarding this functionality.

7.2.1.3 Reference Prediction

The following introduce one of the contribution of this work: the reference prediction process.

How to get the reference of an object ? This section exposes a process to predict the values by which object instances to be created will be referred to. As the previous section indicates, this process involves the memory allocation and reclaiming mechanisms. But the first requirement for this process is to be able to learn the value of an object's reference.

In the context of Java Card 3.0, this question is answered within the API specification [Sun09b] (at least, one answer is suggested).

> "As much as is reasonably practical, the `hashCode` method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™programming language.)"

This suggestion can be argued by the fact that two instances of the `Object` class will only differ by their internal addresses (or Java references if these two concepts are not merged within the considered platform). It is then consistent to use it to distinguish such objects.

On Java Card 2.x.y platforms (as well as on Java Card 3.0 platforms that do not implement the `hashCode` method as suggested) the following approach including type confusion has to be considered as described in Listing 7.3.

Assuming that it is possible to learn the reference of an object instance appears then reasonable.

Listing 7.3: Getting the reference of an object

```
/**
 * Class A holds an Object field: o
 * Class B holds an integral field: ref
 * (the type of the integral field is assumed to
 * match with the size of an object address.)
 * A a and B b are public fields of the class.
 */
public void initConfusion() {
    A a = new A();
    // Need a fault injection at runtime to avoid
    // a ClassCastException throwing.
    B b = (B) (Object) a;
}
public short getReference(Object o) {
    // Type confusion has "merged" a.o and b.ref
    a.o = o;
    return b.ref;
}
```

How to predict the reference of an object ? Under the linear reference assignment assumption, the prediction process is described by Algorithm 2.

Algorithm 2: REFERENCEPREDICTION()

input : -
output: -

0 Delete current unreachable object instances: `System.gc();`

1 Create a new object instance: `Object o = new Object();`

2 Get the reference of this instance: `ref = getReference(o);`

3 Make this object unreachable: `o = null;`

4 Delete this unreferenced object: `System.gc();`

5 The next assigned reference will be `ref`

With regards to (7.1), on step 1 the allocated reference r_k is s.t.

$$r_k = \min\{r_i \text{ s.t. } \forall\, r_j < r_i,\, r_j \text{ is used}\} \tag{7.2}$$

After, step 2 and 3, the attacker knows that r_k is not used anymore.
The following allocated reference r_l will be s.t.

$$r_l = \min\{r_i \text{ s.t. } \forall\, r_j < r_i,\, r_j \text{ is used}\} \tag{7.3}$$

Consequently, putting (7.2) and (7.3) together, it comes that:

$$r_l = r_k \tag{7.4}$$

The successive object instantiation and deletion allow the attacker to discover the next available reference, since it is the one that has just been released. Actually, this behavior has been previously observed by Jip Hogenboom et al. [HM09] in the context of another mechanism leading to memory reclaiming on a Java Card 2.1.1: transaction aborting.

This can be easily tested on any platform supporting the Java Card 3 specifications by running the code in Listing 7.4.

Listing 7.4: Testing the prediction process

```
System.gc();          // First call to the garbage collector to
                      // delete current unreachable references.

o1 = new Object();    // Assign a new reference to o1 and store
h1 = o1.hashCode();   // the value of this reference in h1.

o1 = null;            // Set o1 to null and call the garbage
System.gc();          // collector to actually delete it.

o2 = new Object();    // Assign a new reference to o2 and store
h2 = o2.hashCode();   // the value of this reference in h2.

if (h1 == h2)         // Compare stored references...
    // The assumption is verified.
else
    // The assumption is not verified.
```

Under the assumptions previously stated, the attacker can now consider that she can read/write the reference of an object, but also that she can predict the reference of future object instances.

7.2.2 Application Replay to Circumvent the Application Firewall

7.2.2.1 Application Firewall Implementation.

The application firewall was introduced in Chapter 3. A possible implementation of the context isolation would be to assign each application group a context identifier. When an application creates an object, this object would then inherit the context identifier of its "maker". Then access across context can be easily checked by comparing the accessing context identifier and the accessed context identifier. If the context identifiers are matching access is granted, otherwise a `SecurityEcxeption` is thrown.

Such an implementation appears quite suitable in a constrained system. It does not consume too much memory (one 8/16/32-bit word per object to protect it), the access decision is simple (a word comparison) and it does not constrain the number of objects an application can hold. Actually, some experiments based on ill-formed applet loading on Java Cards 2.X.Y from different card manufacturers and on the C reference

implementation provided within the Java Card 2.2.2 Development Kit [1] has proven this implementation is (at least has been) used. Such an implementation is considered in the remainder of this section.

This implementation choice leads to a first question:

Question 1. *Where does the context identifier comes from ?*

Many answers could be given to that question (a random value, an internal counter, the hash of that application's name, ...). However, the important thing is to ensure that two application instances living at the same time in the card do not have the same context identifier. Hence the answer to this question has only a limited interest from an attacker's point of view.

That point is discussed again in Section 7.2.2.3.

7.2.2.2 Application Instance Deletion

Java Card platforms allow the post-issuance loading of application. With this capacity comes also that of unloading application. To sum up, an application will then go through the following cycle during its life:

1. Application module loading.
2. Application instance creation.
3. Application execution.
4. Application instance deletion.
5. Application module unloading.

In the scope of this work, the application instance deletion is the step of interest. In order not to depend on any implementation specific mechanism, the following only considers this process according to the specifications.

Java Card 2.2.2. On Java Card 2.2.2, applet instance deletion is processed by an entity referred to as the Applet Deletion Manager (ADM). The behavior of the ADM is specified in the JCRE specification. In particular, it is stated that:

> "Applet instance deletion involves the removal of the applet object instance and the objects owned by the applet instance and associated Java Card RE structures."

Consequently, all objects owned by the applet instance must be actually deleted within the deletion process.

[1] JCDK available at `http://oracle.com/`

Java Card 3.0. On Java Card 3.0, applet instance deletion is processed by the Card Management Facility (CMF). The behavior of the CMF is specified in the JCRE specification. In particular, it is stated that:

> "An application instance is successfully deleted when all objects owned by the application instance are inaccessible on the card."
>
> "Upon successful deletion, [the card management facility] fires an application instance deletion event - `event:///standard/app/deleted` - with the application instance URI as the event source to all registered listeners."

The important point to notice here is that what happens between the application instance deletion, the event firing and the actual notification of potential event listeners is not addressed in the specifications. This is indeed the starting point of the attack described in the following section.

7.2.2.3 "Application-Replay" Attack on a Java Card 3.0

The previous section has highlighted a difference between the application instance deletion processes on Java Card 2.2.2 and 3.0. Indeed the later does not mandate that object instances belonging to an application instance are deleted together with the application instance. This leads to think that the mandatory deletion of objects on Java Card 2.2.X has not been thought of as a security mechanism but rather as a functional one. Actually, since the garbage collection is not mandatory on Java Card 2.2.X platforms, one has to explicitly delete objects that are not used anymore. Else they will still be consuming memory. This explains the disappearing of this statement in the Java Card 3.0 specifications since the garbage collector ensures that those objects will be deleted eventually. The application-replay attack detailed hereafter is then limited to Java Card 3.0 platforms.

The remainder of this section describe a possible attack scenario divided into two steps:

- First, the attacker needs to prevent the deletion of the targeted application's objects;
- Then, the attacker must find a way to access these objects despite the application firewall.

Illegal Memory Consumption. The aim of this first step of the attack is, for the adversary's application, to gain references to objects belonging to the targeted application, even though this application gets deleted.

Quote 7.2.2.2 states that the CMF must be considered an application instance deletion successful when all objects it owned are inaccessible. This means these objects are garbage-collectable, but not necessarily that they have been garbage-collected. In addition, *Quote 7.2.2.2* states that the CMF will fire an event on successful deletion. Finally, the attacker knows how to predict and forge object references from Section 7.2.1.

Bounding Object Instances Owned by Another Application Instance. Consider now two applications called `Forgery` and `Target`, respectively the adversary's and the targeted application. Assume that `Forgery` is loaded and instantiated. That is to say, its binary representation is on-card and has been initialized. On the other hand, assume that `Target` is only loaded. That is to say its binary representation is on-card but it has not been initialized. Furthermore, let `Forgery` register an event listener to be notified of application instance deletions (say on the URI `event:///standard/app/deleted/*`).

The `Forgery` application instance can then guess the starting and ending bounds of the `Target` application instance to be, following these steps:

1. Call the garbage collector.
2. Predict the next reference (said `start`).
3. Let `Target` be instantiated.
4. Instantiate an object to get the "current" reference and deduce the last reference instantiated by `Target` (said `end`).

The `Forgery` application then knows that the references of `Target`'s objects are s.t.

$$\forall r_i \in \texttt{Target}, \texttt{start} \leq r_i \leq \texttt{end}$$

Preventing the Deletion of Objects on Application Instance Deletion. The attacker can then request the `Target` application instance deletion. During the deletion process, the card management facility will ensure that all objects belonging to this application instance are not referenced anymore. However these objects are not necessarily deleted as long as the garbage collector is not executed.

On notification of the application instance successful deletion, the `Forgery` application can then forge object's references in an array of $\texttt{end} - \texttt{start}$ objects to values between `start` and $\texttt{end} - 1$. This is achieved through a type confusion similar to that exposed in Listing 7.3, considering the confused object is an instance field[2]. By doing so, the attacker prevents these objects from being actually deleted by the garbage collector.

At this point, the attacker's application instance hold references to objects that do not belong to it. Trying to access these objects in that application would then lead to a `SecurityException` throwing. The following section adapts the principle of the replay attack to overcome this.

Application Firewall Circumvention. The adversary's application holds references belonging to a deleted application instance. The only way to access these references would then be to collaborate with a new application instance impersonating the deleted one.

[2]Consequently a single fault injection is necessary for all reference forgeries.

It becomes obvious now that the answer to *Question* 7.2.2.1 would then only be useful to help answering the real critical question:

Question 2. *Can a new application instance be given the same context identifier as a former (deleted) application instance ?*

If *Question* 7.2.2.1 cannot be accurately answered without knowing the exact implementation of the platform, *Question* 7.2.2.3 could be answered by experimentation. Given that no Java Card 3.0 platforms have been publicly released so far this particular behavior on various Java Card 3.0 platforms has not been tested. The experiment was nevertheless run on different cards implementing different versions of the Java Card 2 specifications with mostly positive results. Assume now that the answer to the *Question* 7.2.2.3 was "Yes".

The attacker would then only have to instantiate a new application, "send" the forged objects from `Forgery` to that new application and try to access them. This operation can be repeated until no `SecurityException` is thrown, which means that the new application has been assigned the same context identifier as the original `Target` application instance. That is to say, the new application impersonates the previous `Target`'s application instance.

The last difficulty resides in the "sending" of the forged objects from `Forgery` to the new application, since `Forgery` is not authorized to use these objects by the application firewall. This is why an array of forged objects was considered in 7.2.2.3 (the array itself is then still legally usable by `Forgery`). A mere library permits then to store this array from `Forgery` and access its content from the new application without having to pass through the application firewall.

Eventually, the new application instance has then full access to the objects created by the `Target`'s application instance. The application firewall has been circumvented.

7.3 Abuse of Global Arrays

The *APDU buffer* is another vital point of a Java Card which can be compromised by a combined attack. This buffer is used to exchange all data that passes between the smart card and the terminal. It is therefore a central element of any smart card. The following parts show how an attacker can spy on the content of the APDU buffer or modify it through concrete examples on Java Cards.

7.3.1 Precisions on the APDU Buffer and Targeted Platforms

The attack exposed is about the attacker's ability to illegally access the Application Protocol Data Unit buffer within an applet of her own. This section:

- discusses the possible usage of the APDU buffer and their potential security issues,

- outlines a couple of statements from the JCRE specification relative to the security of the APDU buffer in a Java Card platform,
- presents the two platforms considered in the scope of this work.

7.3.1.1 The APDU Buffer Usage

At first glance, having hand over the APDU buffer is only a Man-In-The-Middle attack [ANN05] between the card and the card reader. However when looking more carefully, one can observe that the APDU buffer not only contains received or emitted data but is also a temporary buffer that the application uses, due to the inherent resource constraints in the smart card field. Two examples are exposed hereafter which illustrate how powerful an adversary is compared to a Man-In-The-Middle attack when she can spy on and/or modify the content of the APDU buffer.

The first example is based on *Secure Channel* which is the most common countermeasure to counteract Man-In-The-Middle. The principle of a Secure Channel is to share a key between the card and the reader and then to ensure confidentiality and integrity of the communication by using this key with cryptographic functions such as encryption or MAC verification for instance. For simplicity reasons and memory consumption optimisation, such operations are often performed in the APDU buffer. In such a case a Man-In-The-Middle cannot alter the exchanged data without being detected nor recover the sensitive information which are exchanged. However, an attacker having hand over the APDU buffer can not only spy on the communication, bypassing the confidentiality insurance, but she can also modify the command after the MAC verification leading to very powerful potential attacks.

A second example concerns the management of the APDU buffer during commands requiring a very large amount of memory space, such as asymmetric cryptographic computations for instance. In such a case, the developer can use the APDU buffer as a temporary buffer during the application execution to extend the memory space capability of the device. However, an attacker spying on the APDU buffer during the cryptographic computation can compromise the security of the system if sensitive values such as cryptographic keys or temporary results are manipulated in this buffer. Moreover, if the attacker can modify the values manipulated in the APDU buffer, a logical fault can be injected on a temporary value leading to an erroneous cryptographic output. Such a faulty output can then be used to recover the corresponding cryptographic secret key by using Differential Fault Analysis [GT04].

The examples described above emphasize the strength of an attacker if she succeeds in spying on and modifying the APDU buffer during the execution of an application. In the following, the specificities of the APDU buffer in the context of a Java Card are presented with more details.

7.3.1.2 Specificities of the APDU Buffer in a Java Card

On a Java Card platform, a global array is a particular type of array object that is owned by the JCRE but accessible by different applications. The APDU buffer object contains such an array which is accessible through the `javacard.framework.APDU.getBuffer()`

virtual method. This arrays holds the incoming APDU command and the outgoing APDU response.

The sensitive nature of such arrays is quite obvious since they are potentially shared amongst all applications. Therefore the JCRE specifications mandate several restrictions and verifications concerning its use.

Firstly, to prevent an application from accessing a global array when it should not, the following restriction applies:

> *"Accessing Class Instance Object Fields (...). Otherwise, if the bytecode is* `putfield` *and the field being stored is a reference type and the reference being stored is a reference to a temporary JCRE entry point object or a global array, access is denied."*[Sun09e, §2.4.2.8]

Secondly, to avoid any data leakage from one application to another, the following statement is specified:

> *"Because of the global status of the APDU buffer, it MUST be cleared to zeroes whenever an applet is selected, before the applet container accepts a new APDU command."*[Sun09e, §2.4.2.2]

7.3.1.3 Targeted Platforms

The different versions of the Java Card specification have already been introduced in this dissertation. The platforms targeted by the attack described in the following are

- the Java Card 3.0 Classic Edition,
- the Java Card 3.0 Connected Edition.

7.3.1.4 Java Card 3.0 Classic Edition

Concerning the Java Card 3.0 *Classic* Edition, the interfaces with the GlobalPlatform environment and the Card/(U)SIM Application Toolkits are particularly of interest in the scope of this work.

7.3.1.5 Java Card 3.0 Connected Edition.

Concerning the Java Card 3.0 *Connected* Edition the attack is based on the multithreading support and in particular on the notion of *Restartable Tasks*. The notion of *Restartable Tasks* is based on a task registry in which an application can register/unregister tasks it wishes to launch automatically whenever the system is powered on. In particular, an application can register an object instance of a class implementing the interface `Runnable` (by extending the class `Thread` typically) into the registry by a call to the static method `TaskRegistry.register(Runnable t)`. Subsequently, the `run()` method of this object instance will be automatically executed in a new thread of execution every time the system starts up.

7.3.2 The APDU Buffer Storage Attacks

This section exposes:

- a fault attack allowing an attacker to store the APDU buffer despite the JCRE restrictions,
- how to exploit such a capability to mount a full attack path on platforms implementing either the *Classic* or the *Connected* Editions of the Java Card 3.0 specifications.

7.3.2.1 A Fault Attack to Store the APDU Buffer

Attacks against Java Card platforms often take advantage of ill-formed applications loaded on-card without going through the bytecode verification process. The combination of a malicious, but yet well-formed, application with a single physical fault injection can allow an attacker to gain a permanent access to the APDU buffer array whatever Edition the Java Card implements.

As stated in Section 7.3.1.2, the JCRE is responsible for preventing an application from storing references to global arrays, and in particular to the APDU buffer array. Therefore, the JCRE must perform runtime checks to enforce this rule. Without loss of generality, assume that the JCRE operates the check described in Listing 7.5 when executing a `putfield` instruction.

Listing 7.5: Detection of APDU buffer storage attempt in `putfield`

```
// ref points to the object to store
if (isGlobalArray(ref)) {
  // Handle storage attempt
  throw SecurityException
}
```

Obviously, the aim of the attacker is to force the jump in the *else-branch*. Since such a disturbance can be achieved thanks to a fault injection [GT04], the attacker can run within her own applet a method trying to store the reference of the APDU buffer array into a global array and disturb the conditional branching execution to avoid the `SecurityException`.

Assume that the attacker has been successful with the fault injection. As a result, she has been able to store the reference of the APDU buffer into a non-volatile field. Using this field, she can access the APDU buffer at any time. The following shows how such a capability can be exploited by an attacker on the Java Card 3.0 *Classic* and *Connected* Editions.

7.3.2.2 Attacking a Java Card 3.0 Classic Edition

Assume that the attacker can access the APDU buffer at any time by using the attack presented in Section 7.3.2.1. However, without any interaction with other entities on-card, she cannot take advantage of this privilege in other ways than accessing the command and response of her own application.

To be able to exploit her new facility, the attacker's application must be given a chance to run when the APDU buffer is meant for another application. It is therefore necessary that it exposes one or several method(s) that might be called by another entity on the platform through shareable interfaces. The point is that when called, these shared method would allow the attacker to read or corrupt the APDU buffer "belonging" to the entity calling the method.

One can think that such a situation where an entity on-card calls the shared method of another applet would only appear in an attack proof of concept. However, Section 7.3.3 demonstrates that different contexts can lead to this situation in practice.

7.3.2.3 Attacking a Java Card 3.0 Connected Edition

The following shows how an attacker can take advantage of the Java Card 3.0 *Connected* Edition multithreading facility to exploit the privilege of accessing the APDU buffer whenever she wants.

Considering the attacker has been able to store the APDU buffer, one can imagine a restartable task whose `run()` method infinitely loops and spies upon the APDU buffer, such as detailed in Listing 7.6.

Listing 7.6: The eavesdropping restartable task

```
/**
 * - fieldBuf is the stored APDU buffer reference.
 * - BUF_LEN is the assumed APDU buffer length.
 * - tmpBuf is a byte array initialized with a size of BUF_LEN.
 * - os is an OutputStream used by the attacker
 */
public void run() {
  while (true) {
    // APDU buffer is different, copy its content.
    if (arrayCompare(fieldBuf, 0, tmpBuf, 0, BUF_LEN) != 0) {
      System.arraycopy(fieldBuf, 0, tmpBuf, 0, BUF_LEN);
      os.write(tmpBuf);
    }
  }
}
```

The attacker is then potentially able to dump every byte written in the APDU buffer inside her `run` method. Moreover, she can also modify the content of the APDU buffer by writing into instead of copying it.

This section has shown how a Combined Attack on a Java Card can allow an attacker to spy the content of the APDU buffer during the execution of an application. Section 7.3.3 exposes different case studies based on this capability.

7.3.3 Case Study

This section exhibits two case studies from two important specifications of the Java Card ecosystem, namely the GlobalPlatform (GP) environment [Glo11a, Glo12a] and the Card

and (U)SIM Application ToolKit (CAT/(U)SAT) [Eur11a, Eur11c] that were presented in Chapter 3. In addition, it details a possible exploitation of the restartable task described in Section 7.3.2.3.

7.3.3.1 Attacking through the GlobalPlatform Environment: OPEN

GP is an entity developing and publishing specifications relative to the deployment and management of embedded applications on secure chip technologies. As part of the GP specifications [Glo11a], we find the description of the GP environment: the OPEN.

The GP Environment: OPEN As per [Glo11a], the OPEN is the on-card entity responsible for command dispatch, card content management operations, security management operations and secure inter-application communication. According to this last responsibility, the OPEN including its contactless extension [Glo12a], is the GP entity that sends notifications to other on-card entities when certain events occur. In order to keep track of the different on-card entities, the OPEN owns and uses an internal GP registry as an information resource. This registry contains information for managing the card, executable load files, applications, *Security Domain* associations, and privileges.

Shareable Interface Method Call from the OPEN. The event notifications are operated through calls to a shareable interface method. The only limitation is then for the attacking application to register for such notifications. In the scope of GP's contactless services, applications that implement the `CLApplet` interface of the GP API [Glo11b, Glo12b] shall be notified of changes occurring to their GP registry entry. These changes can have various origins, depending on the Contactless Registry Service (CRS). For instance, the application is notified of its installation on the platform, of the modification of the contactless communication protocol, etc... (the complete list of events can be found in [Glo11b, Glo12b]). These notifications are implemented by calls to the `notifyCLEvent` method of this interface.

Therefore, the applet detailed in Listing 7.7, which has gained a permanent access to the APDU buffer array as described in Section 7.3.2, is likely to analyse its content each time the registry entry of the applet is updated. As such updates occur quite often, the attacker can then access frequently to the APDU buffer.

Listing 7.7: APDU analysis on event notification when the attacking applet implements `CLApplet`

```
public class MyApplet extends Applet, implements CLApplet {
    ...
    public void notifyCLEvent(short event) {
       analyseAPDU(); // using the stored reference
    }
}
```

One could note that these updates of the GP registry are typically privileged operations performed by the `CRSApplication`. Therefore, the data potentially contained in the

APDU buffer is likely to be particularly sensitive. This could be for instance data having led to a successful authentification or granted authorization.

In this section we have shown how the access to the APDU buffer could lead to gain sensitive information relative to the security of both the platform and the hosted applications. The following section describes how the privacy of the card holder can also be threatened.

7.3.3.2 Attacking through the CAT/(U)SAT

Mobile communication is in constant evolution since the early 90s. With the well known (U)SIM card and UICC (resp. for (Universal) Subscriber Identity Module and Universal Integrated Circuit Card), it is today the most important market in the smart card industry. The CAT [Eur11a] and (U)SAT [Eur11c] are standards from the mobile communication. Their main goal is to define how the smart card should interact with the outside world and initiate commands independently of both the handset and the network. This section shows how these can be misused by an attacker with the APDU buffer access privilege.

The CAT Runtime Environment and the (U)SAT Framework. As part of these toolkits, the CAT and (U)SAT Application Programming Interfaces (APIs) for Java Card are respectively specified in [Eur11b] and [Eur11d]. Java Card toolkit applets are then likely to control access to the network, displaying menus on the handset, etc...

These features are achieved thanks to the *Toolkit Registry*, which similarly to the GP registry defined in the previous section, allows a *Toolkit Applet* to register to events fired by the runtime environment.

Eavesdropping and Corrupting the Short Message Service. As for the attack described in the previous section, event notifications are operating through calls to a shareable interface method. In order to register to some events, an applet must implement the interface `uicc.toolkit.ToolkitInterface` and call the `setEvent(short event)` method of its ToolkitRegistry instance (available by a call to `uicc.toolkit.ToolkitRegistrySystem.getEntry()`). Subsequently, the implemented `processToolkit(short event)` will be triggered each time an event to which the applet is registered occurs.

In the context of the attack we describe, the attacker has then all the reasons to register to all possible events, in order to have her eavesdropping method called as often as possible, in particular when events associated to the reception of a short message through the Short Message Service (SMS) occur. The attacker's toolkit applet is described in Listing 7.8.

Provided, short messages are located in the APDU buffer, the attacker can intercept them and either redirect them to the outside world or modify their content as she pleases. This is one of the many ways the attacker can use the APDU buffer in this context. Other potential applications can also take advantage of the pro-active capability of the CAT environment to redirect outgoing messages or calls to taxed services for instance.

The two previous case studies were targeting the Java Card 3.0 *Classic* Edition and earlier. The next one described how the so-called eavesdropping restartable task can be

Listing 7.8: Eavesdropping and corrupting the SMS

```
public class MyApplet    extends Applet,
                         implements ToolkitInterface {
  ToolkitRegistry r;
  public MyApplet() {
    r = ToolkitRegistrySystem.getEntry();
    ...
    r.setEvent(ToolkitConstants.EVENT_UNFORMATTED_SMS_PP_UPD);
  }

  public void processToolkit(short ev) {
    if(ev == ToolkitConstants.EVENT_UNFORMATTED_SMS_PP_UPD){
      analyseAPDUSMS(); // using the stored reference
    }
    ...
  }
}
```

exploited on a Java Card 3.0 *Connected* Edition.

7.3.3.3 Attacking through the Eavesdropping Restartable Task

This section depicts a scenario where the attacker can eavesdrop or tamper with the communication, even if secured by a secure channel (see Chapter 3) and temporary data. In both cases we can use the result presented in Section , exploiting I/O flooding to force a thread scheduling at a specific time. As a consequence, the attacker finds herself in the situation described in Section 7.3.1.1 where she can access the APDU buffer almost whenever she pleases. In the following, we detail a case study proving the potential threat of such a situation.

Breaking the Secure Channel. As stated in Section 7.3.1.1, a Secure Channel is a mechanism provided by GP to ensure both the confidentiality and integrity of the terminal-card communication through cryptographic mechanisms. It is shown here that the restartable task we have introduced in Section 7.3.2.3 can be use to break a Secure Channel.

The initialisation of a Secure Channel is made thanks to two APDU commands, namely INIT_UPDATE and EXT_AUTHENTICATE, with specific `CLA` and `INS` bytes (respectively `80 50` and `84 82`). Therefore, the eavesdropping task can detect the beginning of a Secure Channel session by detecting these commands. Subsequently, the deciphering (resp. ciphering) and MAC checking (resp. computing) of an incoming (resp. outgoing) APDU is operated thanks to a call to the method `unwrap(byte[] baBuffer, short sOffset, short sLength)` (resp. `wrap(byte[] baBuffer, short sOffset, short sLength)`) of the `SecureChannel` interface. The point is that if this method is called with the APDU buffer array as parameter, the attacker owning the restartable task will be able to both eavesdrop and corrupt the communication. The attack scenario is depicted in Figure 7.4.

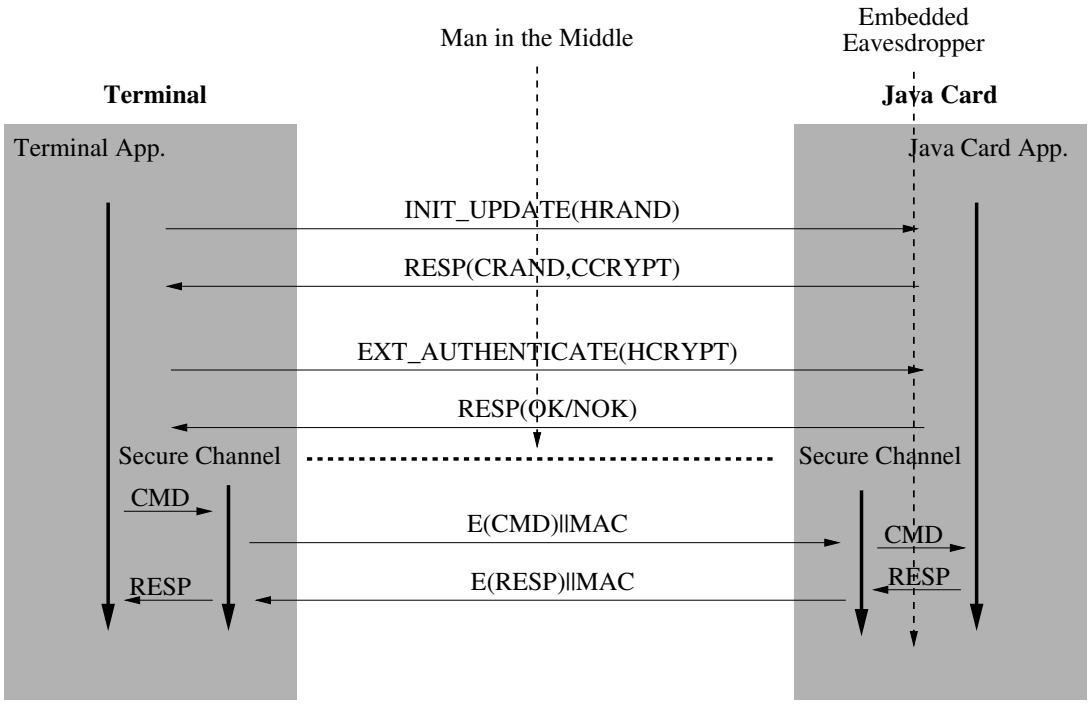


Figure 7.4: Breaking the Secure Channel with the Eavesdropping Restartable Task

The Secure Channel is indeed used in numerous application in all smart card fields of application, from finance to health-care. If deemed successful, the described attack would have serious consequences regarding security and privacy.

Conclusion to Part II

As presumed in the literature [Ver06b, MP08], adding the fault injection capacity to the attacker model results in several potential breaches. The security analysis performed has allowed to show potential weaknesses concerning three particularly important notions of the Java Card security:

- the type safety,
- the execution flow integrity,
- the application isolation.

Looking back on the state-of-the-art attacks, it is obvious that the type safety property is the cornerstone of the security of Java Card platforms. Breaking this property has indeed allowed attackers to either dump the memory content or corrupt internal data, including on-card application code [Wit03, Hyp03, CHS03, Ver06b, MP08, HM09, ICL10]. Chapter 5 has shown that the type safety property enforced within the Java Card Runtime Environment can be broken, even on platforms where an ill-formed application cannot be loaded. Indeed, the two exposed Combined Attacks exploit fault injections targeting different elements of the system. In the first one, the code implementing a basic instruction of the Java Card platform is disrupted, whereas the second one illustrates a fault injected on the data manipulated by the Java Card Runtime Environment. In addition, this chapter has also introduced the notion of instance confusion which despite its similarity with type confusion can offer different attack vectors. Finally, several exploitations of these various confusions have been detailed. All these different exploitations represent dangerous threats to both the platform and the applications it host. However, the possible modification of any application embedded on-card allowed through the abuse of the `Class` class is definitely the most threatening one.

The second tackled security notion concerns the execution flow of an embedded Java Card application. Chapter 6 has proven that the modification of the execution flow of an application can compromise the application itself but also the platform hosting it. First it has been shown that the lack of a real boolean type at the Java bytecode level definitely raises security issues. As the exposed experimental results prove, the obvious perturbation of a conditional branching instruction might very well be practical on various platforms if the application or the platform do not enforce redundant checks on the condition evaluation. Then a completely different issue regarding multithreaded platforms has been exposed. Regarding the usual resource constraints of smart cards, it is difficult to imagine the implementation of complex thread scheduling mechanisms. In

addition, the introduced I/O flooding technique which has permitted in this context to force the interruption of an application at a given time might be used for other purposes. The attack described here practically proves the fact that adding new features to a platform inherently enlarge the attack surface and multiply attack vectors. It is worth noting that the introduction of the features exploited here are resulting from the integration of the Java Card in the web ecosystem, which is a topic of interest from a security point of view [BBKL11, Cal12]. Finally an overview of the potential attacks that can be built upon, and around, the exception-related mechanisms of the Java Card technology has been given. In particular it has exposed that attacks on these mechanisms are likely to bypass certain operations contained in *catch*-blocks and to break the type safety property of the Java Card Virtual Machine. In addition, this study has allowed to exhibit a residual weakness regarding the incomplete initialization of objects which had already been pointed out in the standard Java web applet context with serious consequences on the client side executing the applet. Again this raises issues to consider as per the integration of Java Cards in the web ecosystem.

The last tackled notion is the application isolation which is a crucial security feature for any multiapplicative system. Chapter 7 has exposed different security issues regarding the isolation of both application code and data. Since the JCRE specification only mandates the implementation of these features the exposed analysis relies on several assumptions which have been experimentally validated on different platforms. First it has been exposed that the class loading mechanism might be abused by a type confusion, thus circumventing the code isolation mechanism. This attack echoes the `Shareable` abuse presented in Chapter 4 and results in the execution of a non-expected shared method, potentially behaving very differently than the expected one. Yet it also requires a physical perturbation of the application firewall, like the attack described in [VF10]. Afterward a different way to circumvent the application firewall has been introduced This attack exploits a novelty of the Java Card 3 specification : automatic garbage collection, and what is considered as a residual weakness in the specification. It is based on the introduced notion of reference prediction and on an assumption on the implementation of the application firewall allowing a so-called *application-replay*. Finally the consequences of an attacker gaining a permanent access to a global array such as the APDU buffer have been explored. As exposed such a capability compromises both the security of the whole platform and the privacy of the card holder himself, provided he is not himself the attacker.

Part III

Contributions to Enhance the Security of Java Cards against Hardware Attacks

Introduction to Part III

Part II has exposed various attack paths resulting from the security analysis of Java Card platforms. These attack paths have shown the importance of providing a strong security level, even against attackers with fault injection capacity. Furthermore, they have shown that Fault and Combined Attacks are really hard to handle by static analysis tools applied at the Java Card application level such as the bytecode verifier. This difficulty is due on the one hand to the so-to-speak dynamic character of these attacks and on the other hand to the fact that the security of the application relies, at least partially, on the security of the Java Card Runtime Environment, which relies itself, at least partially, on the security enforced by the underlying hardware. It results that securing a whole system always implies to find an appropriate trade-off at each layer of this system between security and performance. This is the main reason for the importance of a thorough vulnerability analysis, so that one knows the particular weaknesses of a system and secures them accordingly.

The type safety property has proven to be a crucial point in the security of Java Card platforms. It appears then important for a Java Card system to provide strong mechanisms to ensure this property. However it was shown that several paths can be taken by attackers to break it. Since all these paths do not meet at some specific point, it is therefore necessary to deal with them one by one. Another particularity of type confusion attacks is that even when considering fault attacks with a transient effect, the generated type confusion can be persistent. This represents a real asset for attackers. If a platform decides not to perform the numerous checks that could prevent a type confusion to occur, it should on the other hand implements other controls that would eventually detect the type confusion and react in consequence.

Another important element regarding the security of Java Card applications is that the integrity of its execution flow should be protected so that it does not behave in an unwanted or unchecked way. Fault attacks have proven to be very efficient to disrupt branching instructions. Different countermeasures can enforce the integrity of the execution flow of trusted code and prevent the execution of intrusted code installed on card by malicious ways.

Finally the application isolation is obviously an important property for any multiapplicative system. Ensuring this property should also be subject to a particular care with consideration for an attacker with fault injection capability. This particular care should be taken at both the specification and implementation level.

Chapter 8

Offensive, Defensive and Defending Virtual Machines

Contents

8.1 The Offensive and Defensive Virtual Machines 153
8.1.1 Definitions 153
8.2 The Defending Virtual Machine 154
8.2.1 Definition 154
8.2.2 When to Check for Faults? 154
8.3 Sealing the Exposed Weaknesses in a Defending Virtual Machine . . . 155
8.3.1 Protection of the `checkcast` Instruction 155
8.3.2 Preserving the Execution Context in Multithreaded Environments 156
8.3.3 Building a Fault-Resilient Firewall 158
8.3.4 Preventing Application-Replay 158
8.3.5 Ensuring Global Arrays Access Restriction 159

This chapter defines the various approaches that can be used by JCVM implementors to ensure the security of both the platform itself and the applications it may host.

8.1 The Offensive and Defensive Virtual Machines

The offensive and defensive virtual machines actually follow a similar philosophy. Definitions 2 and 3 state their respective characteristics as defined in the Java Card protection profiles introduced in Chapter 3.

8.1.1 Definitions

Definition 2. An **offensive JCVM** is a virtual machine that ***does not enforce any specific countermeasure at runtime*** and that relies on the assumption that the interpreted applications have been successfully verified by an ***off-card bytecode verifier*** before being loaded.

Definition 3. A **defensive JCVM** is a virtual machine that ***does not enforce any specific countermeasure at runtime*** and that relies on the execution of an ***on-card bytecode verifier*** ensuring that the interpreted applications are well-formed at load time.

As defined, both approaches strongly rely on the bytecode verification process to ensure the security of the whole system. However, as the previous part of this dissertation has exposed, numerous attacks can be mounted without requiring any ill-formed application loading at all. Considering Fault Attacks and Combined Attacks, the security brought by the bytecode verifier can be circumvented and does not appear sufficient.

Virtual machines sticking to a strict offensive or defensive approach may nevertheless require that applications pass additional static analysis tools detecting potentially hostile applications, such as the one described in Section 4.4.3 for instance. However this may represent an additional obstacle with regard to an open platform philosophy.

8.2 The Defending Virtual Machine

The third approach that may be used by JCVM implementors is the defending approach.

8.2.1 Definition

Definition 4. A **defending JCVM** is a virtual machine that ***does enforce some specific countermeasures at runtime*** and that additionally relies on the execution of a ***bytecode verifier either on-card or off-card*** ensuring that the interpreted applications are well-formed at load time.

Depending on the mechanisms that are actually implemented on a defending JCVM, such platforms are then more likely to offer a good security level against Fault and Combined Attacks. To reach this good security level a thorough security analysis is necessary. Part II has exposed several potential weaknesses resulting from such an analysis. The following chapters give leads and techniques to seal such breaches.

However one must always find an appropriate trade-off in order not to penalize too much the performance of defending JCVMs while preserving the security level.

8.2.2 When to Check for Faults?

A foundation of countermeasure designing lies in the definition of the assets to protect. A good comprehension of the threats is necessary to achieve this work. This section aims at analysing the attacks previously described in order to determine the operations that are sensitive and thus worth protecting. As the attacks target the JCRE, the following considers operations at the Java bytecode level.

In the context of Java Cards, defenses are meant to prevent:

- Data from being unduly sent out of the card,

- Applications from ill-behaving.

Indeed, these identified assets summarize the assets defined in the Java Card Protection Profile (§3.2 of [Tru06]).

The fault detection can then be restricted to the bytecode instructions related to:

- Field manipulation (get/putfield, get/putstatic).
- Control-flow breaks (invokes, returns, conditions, exceptions).

Indeed, the other instructions are arithmetic or logic operations, or operate on local variables.

Amongst the identified instructions to check, only the operations on static fields and control-flow-break instructions are not protected by the Java Card application firewall. Therefore, it appears suitable to add fault detection checks within the application firewall mechanism, as well as on these other instructions.

8.3 Sealing the Exposed Weaknesses in a Defending Virtual Machine

8.3.1 Protection of the `checkcast` Instruction

Section 5.1 has exposed how a fault injection can induce an erroneous execution of the `checkcast` instruction. With regards to the specification of this instruction (given in Section 5.1), one can state that its implementation should follow Algorithm 3.

The described attack basically consists in the disruption (or the skip) of the native conditional branching instruction at line 11 in Algorithm 3.

This attack is then based on a weakness of the underlying hardware and on the lack of redundant checks of the condition evaluation within the implementation of the `checkcast` instruction. With regards to the definitions given above, this implementation can be found on both *offensive* and *defensive* JCVM. On the other hand, *defending* JCVM may implement this instruction in a secured way by making this last assumption invalid. This can simply be done by adding redundant checks on the different conditional branching instructions.

In addition, when implementing a secure conditional branching, one must take into account the potential ability of an attacker to perform a double fault injection, that is to say to inject the same fault twice in a row. The redundant check must then be performed accordingly. Algorithm 4 depicts a secure algorithm for the `checkcast` instruction against both single and double fault injection.

Algorithm 3: The `checkcast` instruction

input : Popped object instance : o
input : Parameter class : c
output: o or `ClassCastException`

```
1  if o == null then
2  |  push(o);
3  |  return;
4  end
5  if c == java.lang.Object then
6  |  push(o);
7  |  return;
8  end
9  c' ← getClass(o);
10 while c' != java.lang.Object do
11 |  if c' == c then
12 |  |  push(o);
13 |  |  return;
14 |  end
15 |  c' ← getSuperClass(c');
16 end
17 throw(ClassCastException);
```

8.3.2 Preserving the Execution Context in Multithreaded Environments

Section 7.3.3.3 has exposed an attack scenario based on an abuse of the multithreading feature and a type confusion allowing the attacker to tamper with the execution context of a running application.

This attack is undoubtedly not easy to set up. However its apparent complexity should not conceal the potential consequences for an application. This statement is particularly true for the following reasons:

- The fault model turns out to be extremely powerful. Therefore most of the sensitive functions of an application may be defeated, even if they have been secured with care.
- A weak system may lead to an alteration of any hosted applications.
- The adequate protection is unlikely to be found in the application. As a result, an application with a thorough concern of security may be broken. It is then of the utmost importance that a system shows the evidence it is reliable and trustworthy.

Many ways can be explored to find effective protections against this threat. Firstly it is worth strengthening the scheduler to make sure it cannot be abused. The protection must be adapted to the rule enforced by the multithreaded system. Based on a time slice the scheduler of the tested Java Card makes use of a timer. By stopping this timer in the

Algorithm 4: The `checkcast` instruction

```
   input  : Popped object instance : o
   input  : Parameter class : c
   output: o or ClassCastException or error
 1 o ← pop();
 2 c ← getInsParameter();
 3 if o == null then
 4 |   if o != null then
 5 |   |   handleAttackDetection();
 6 |   end
 7 |   push(o);
 8 |   return;
 9 end
10 if c == java.lang.Object then
11 |   if c != java.lang.Object then
12 |   |   handleAttackDetection();
13 |   end
14 |   push(o);
15 |   return;
16 end
17 c' ← getClass(o);
18 while c' != java.lang.Object do
19 |   if c' == c then
20 |   |   if c' != c then
21 |   |   |   handleAttackDetection();
22 |   |   end
23 |   |   push(o);
24 |   |   return;
25 |   end
26 |   c' ← getSuperClass(c');
27 end
28 throw(ClassCastException);
```

handler in charge of the network requests, the tested Java Card withstands the attack.

The identification of the targeted instruction on the power consumption trace has also been an elementary step of our attack. Therefore, the difficulty to set up the attack increases with the difficulty to locate the instruction to attack. Techniques to harden the power consumption analysis such as described in [Sha00, MG08] would then stand as an additional barrier to circumvent for the attacker.

Another way consists of a strong isolation between the memory areas of different contexts. This includes the runtime environment area where the thread contexts are stored. Such an isolation may prevent the attacker from having access to the sensitive

context. It is more or less difficult to achieve according to the systems. On a smartcard it may be interesting to take advantage of specific hardware features, such as a memory protection unit (MPU). This kind a protections enforces a strong isolation by the mean of hardware controls, which remain very difficult to overcome.

Lastly it may be worth of implementing some integrity controls on the contexts during the thread switch. As the control value must be prevented from being modified by an adversary, this may be achieved through a MAC verification using an internal symmetric key for instance. Before restoring a context, the scheduler would be in charge of checking that nothing has been tampered with and could send an alarm if an inconsistency is found.

8.3.3 Building a Fault-Resilient Firewall

As described in Section 4.4 one of the first Fault Attacks against a Java Card platform was targeting an implementation of the application firewall. Similarly to the attack described in Section 5.1, the fault injection used here was meant to disrupt a native conditional branching instruction corresponding to the access control decision.

Therefore, and as exposed in the original article of Éric Vétillard et al. [VF10], an efficient countermeasure against this attack would be to make this decision fault-resilient by adding a redundant check of the condition evaluation, as described in Section 8.3.1 for the implementation of the `checkcast` instruction.

8.3.4 Preventing Application-Replay

The attack describe in Section 7.2 relies on two key elements:

- a “lazy” application instance deletion process.
- the attacker’s ability to provoke a type flaw and to forge an object’s reference.

Object Deletion. The basement of this attack lies in *Quote* 7.2.2.2, *i.e.* the card manager only ensures that objects owned by the application instance to be deleted are not accessible anymore. In a way, the specifications encourage implementors to give in to the temptation to rely on automatic garbage collection for the effective deletion of these objects.

Thus, it is assumed that the garbage collection will be executed later and that it will delete all inaccessible objects. But between the successful deletion event is fired and the next garbage collection is requested, many things can happen, as exposed in Section 7.2.

It appears then necessary that the application instance deletion process ensures not only that the objects previously owned by the application to be deleted are inaccessible but also that they are actually deleted when the application instance is deleted, or at least that their content be reset to a default value (*null* or 0 depending on the field type). This

is indeed what prevents the attack from succeeding on Java Card 2.X platforms, although it seems that this has not been specified to enforce security according to the analysis led in Section 7.2.

This possible breach may easily be taken care of within the implementation of the CMF (Card Management Facility) by explicitly calling the garbage collector at the end of the deletion process. Nevertheless, it would be better if the Java Card 3 specifications include the mandatory deletion of objects belonging to an application instance as part of its deletion process.

8.3.5 Ensuring Global Arrays Access Restriction

The attack exposed in Section 7.3 has highlighted the need to protect the access to the APDU buffer, and more generally to global arrays, against attackers with fault injection capability. As exposed, an attacker with hands on the APDU buffer can mount various attacks allowing her to endanger both the security of the system and the privacy of the card holder.

This threat may come from the non-secured implementation of the global array storage prevention in the platform, as described in Listing 7.5 of Section 7.3. Again to prevent such attacks from succeeding, the JCVM implementors should identify these controls and add redundant checks where needed.

One should also not forget to consider the possibility to forge an object reference to one of these global arrays thanks to an adequate type confusion, as exposed in Section 5.1.

Chapter 9

Ensuring Type Safety in the Presence of Faults

Contents

9.1 Protection of the Operand Stack 161
9.1.1 Software Fault Detection 161
9.1.2 Costs Comparison 164
9.2 Towards Type-Confusion-Immune and Type-Safe Platforms 167
9.2.1 Type-Confusion-Immune Platforms 167
9.2.2 Type-Safe Platforms 167

Chapter 5 has shown that several fault-injection-based attacks can allow to break the type safety property. Until now, no auto-immune type system has been proposed in the literature. It is therefore necessary to build fault-resilient mechanisms to enforce this property.

9.1 Protection of the Operand Stack

The aim of these countermeasures is then limited to protecting the system against a faulty value on the operand stack. Also, the focus is given here to dynamic checks in the context of a defensive VM and static defenses such as detecting potential dangerous mutation within an application [SICL09, SLIC10] are not considered.

9.1.1 Software Fault Detection

This section details different approaches to detect faults within the scope of the JCVM.

9.1.1.1 The basic approach: redundant checks.

The most straightforward implementation of a fault detection mechanism on the stack would be to check the coherency between the value pushed onto the stack and the top of stack value after the push operation. Likewise, with regards to the pop operation, the

coherency between the value that has been popped and the former top of stack value can be checked. This is achieved as exposed in Listing 9.1 and 9.2, respectively for the push and pop operations.

Listing 9.1: Redundant check on push operation

```
// push value onto the operand stack.
push(expected);
// check the top-of-stack value is correct.
if (get_tos() != expected)
   // handle potential error.
   handle_fault();
...
```

Listing 9.2: Redundant check on pop operation

```
// pop the top-of-stack value.
expected = pop();
// check the popped value is correct.
if (get_prev_tos() != expected)
   // handle potential error
   handle_fault();
...
```

This countermeasure has been implemented on a Java Card Virtual Machine. The additional costs on various bytecode instructions are presented in Table 9.1 of Section 9.1.2.

9.1.1.2 First refined approach: propagating errors to ensure fault detection

A possible approach to reduce the cost of redundant check is to propagate a potential error to another component of the JCVM. This is then only valid if the standard JCVM behaviour is to check this other component.

The Java Card application firewall aims at ensuring a strict isolation between the different applications and the JCRE. A typical implementation of this mechanism exhibited on numerous cards and simulation tools is to assign a context identifier to each application. This identifier is also assigned to each object instance created within the scope of an application. The context isolation is then enforced by comparing an object context identifier and the current application identifier, according to the JCRE specification [Sun09e, §2.4].

This context identifier can then be used to propagate the potential operand stack errors. The implementation of the countermeasure is then as exposed in Listing 9.3 and 9.4, respectively for the push and pop operations (with "^" the XOR operation).

Listing 9.3: Error propagation on push operation

```
// push value onto the operand stack.
push(expected);
// propagate potential error on the current application context identifier.
fw_context_id |= (get_tos() ^ expected);
...
```

Listing 9.4: Error propagation on pop operation

```
// pop the top-of-stack value.
expected = pop();
// propagate potential error on the current application context identifier.
fw_context_id |= (get_prev_tos() ^ expected);
...
```

Consequently if an error occurs on the pushed value, the current context of ownership is modified.

Therefore an attacker can no longer retrieve data from the attacked application since she would have to either call virtual or interface methods to send data out of the card or eventually use instance class fields. In both ways, she would have to pass through the application firewall and the firewall will not allow it since the verification of the context identifier would fail. Similarly, if the fault aims at corrupting a conditional branch, the subsequent execution will be interrupted as soon as a firewall check occurs. An additional check of the validity of the context identifier is only necessary on access to static fields that are not protected by the application firewall.

The fact that few additional checks need to be inserted (only for access to static fields) is clearly an advantage regarding the computational cost of this method.

The major drawback of this method is that corrupting the context identifier value, it is possible (yet not very likely) to fix it to the value identifying another application installed on the card. In such a case, our countermeasure would eventually open a breach in the application firewall. Table 9.1 of Section 9.1.2 presents the experimental cost of this refined countermeasure.

9.1.1.3 Second refined approach: introduction of a stack invariant.

A second approach to detect faults in the operand stack consists in adding in the Java Card frame structure a variable σ that allows to exhibit an invariant property.

Definition 5. σ is the sum, considering the XOR operation, of all the values pushed on and popped from the operand stack.

The following invariant property can then be exhibited:

Property 1. *Let $\mathcal{S}_T$ be the set of all the values contained by the operand stack at a given time T. Then at any given time T,*

$$\sigma \oplus \Sigma \mathcal{S}_{\mathcal{T}} = 0$$

Proving this property and its invariance is straightforward. Indeed since σ is by definition the sum of the values pushed onto and popped of the stack, all the values that have been popped have been eliminated from the XOR sum. Therefore, only the values that are still on the stack at a given time T are components of σ.

The implementation of the countermeasure is then as described in Listings 9.5 and 9.6, respectively for the push and pop operations.

Listing 9.5: Updating σ on push operation

```
// push value onto the operand stack.
push(expected);
// update sigma.
sigma ^= expected;
...
```

Listing 9.6: Updating σ on pop operation

```
// pop the top-of-stack value.
expected = pop();
// update sigma.
sigma ^= expected;
...
```

As previously stated, we can check the invariant property during application firewall checks and access to static fields and methods by XORing all the values on the stack to σ.

This approach requires then to add a routine in charge of checking the invariant property. Also it requires to add one word in each Java frame created. Table 9.1 in Section 9.1.2 presents the experimental cost of this countermeasure.

9.1.2 Costs Comparison

This section presents (in Table 9.1) and compares the cost (in time) of the different countermeasures introduced previously and discusses the result of this comparison as well as the different benefits and drawbacks of the different approaches. The countermeasures have been experimented on different short bytecode sequences representing various occasions on which the stack integrity is checked.

The measures have been computed by using the same approach as the MESURE project on a single software implementation of the JCVM executed on a single device. For instance, measuring the `aload+astore` sequence has been done by measuring

the execution time of the two processes described in Listing 9.7, namely $\tau_{reference}$ and $\tau_{measured}$, and using equation 9.1 defined hereafter.

Listing 9.7: The $reference$ and $measured$ applet process

```
public void process(APDU apdu) {
  // The used local variable
      Object local = null;

  if (selectingApplet())
    return;

  byte[] buf = apdu.getBuffer();
  if (buf[ISO7816.OFFSET_CLA] == CLA_TEST) {
    switch (buf[ISO7816.OFFSET_INS]) {
      case INS_REFERENCE:
        // Nothing
        break;
      case INS_MEASURED:
        // aload+astore
        local = local;
        break;
      default:
        ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  } else {
    ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
  }
}
```

$$\tau_{\texttt{aload+astore}} = \tau_{measured} - \tau_{reference} \tag{9.1}$$

Instructions	Basic	Propagation	Invariant
`aload+astore`	39.09 %	21.98 %	12.29 %
`aload+getfield+astore`	19.83 %	12.39 %	11.75 %
`aload+aload+putfield`	27.93 %	18.77 %	17.59 %
`aload+invokevirtual+return`	7.53 %	1.69 %	1.77 %
`aload+invokevirtual+areturn+astore`	8.82 %	3.26 %	2.38 %
`aload+putstatic`	18.60 %	11.58 %	8.89 %
`getstatic+astore`	19.18 %	10.76 %	10.21 %

Table 9.1: Countermeasures impact on bytecode instructions execution time. (increase % referring to an initial implementation with no countermeasures.)

As the costs for the different countermeasures are given in percentage, it is important to bare in mind that the different instructions have very different complexity (which explains the large difference between the results for the sequences `aload+astore` and `aload+invokevirtual+return` for instance).

9.1.2.1 The redundant approach

As expected, this straightforward countermeasure causes a great performance degradation, with a time overhead reaching almost 40 % in the case of the `aload+astore` sequence.

9.1.2.2 The propagation approach.

This countermeasure is definitely more efficient than the basic one. The cost reduction is mainly due to the fact that no integrity check is actually performed when executing simple push/pop operations, as attested by the results observed for the `aload+astore` sequence.

To fix the potential issue of context identifier manipulation, an option could be to force legitimate identifiers to even values and propagated errors to odd values. The detection of an invalid context identifier would then be straightforward.

9.1.2.3 The invariant approach.

The invariant method is also more efficient than the basic one. Its performance is even a little better than that of the propagation countermeasure. This increased performance is explained by the two following statements:

- Like the propagation approach, it does not explicitly perform integrity check on simple push/pop operations.
- Unlike the propagation method this approach does not require to read back the operand stack to detect errors.

Another advantage of this approach is that it does not present the drawback of a potential breach opening in the application firewall.

As expected, the invariant and propagation approaches turn out to be more efficient than the basic one and are relatively close in terms of performance. Nevertheless, the propagation method requires no additional data and only few additional checks on access to static fields. However, as exposed, the propagation method potentially opens a security breach in the application firewall. The use of another variable that would be frequently checked may be recommended. Such variables are typically implementation-dependent and no other quasi-standard one (like the context identifier) can be exhibited.

The invariant approach has proven to be slightly better than the propagation one in terms of execution time. Its implementation is easy and costs one data-word per Java frame.

9.2 Towards Type-Confusion-Immune and Type-Safe Platforms

Until here, this chapter has provided countermeasures against state-of-the-art attacks. But other attacks may render such countermeasures insufficient. For instance an attack consisting in a perturbation of the fetched local variable on an `aload` instruction can easily be envisaged. On the other hand, novel attacks against the type safety may be discovered in the future. Two different approaches can then be envisaged to ensure the security of the platform. They are detailed hereafter.

9.2.1 Type-Confusion-Immune Platforms

The first approach consists in analysing the potential consequences of a type confusion in order to protect the system accordingly even if a type confusion occurs.

With regards to the attacks described in Chapters 4 and 5, this particularly concerns:

- the Java heap organization, and in particular with regards to access to instance fields;
- the implementation of the JCVM regarding the link to virtual methods.

Indeed the observations made by attackers in the literature tends to prove that such mechanisms are already present on certain platforms. The failure of the attack based on the type confusion between a byte array and a short array described in Chapter 4 shows for instance that some platforms take into consideration this situation, probably by checking the bounds of the array.

9.2.2 Type-Safe Platforms

Another approach could consist in checking the type safety property more often.

This could be done for instance as part of the garbage collection mechanism. Since the garbage collector is supposed to determine the objects that are not used anymore, it can be assume that it needs at some points to explore the object instances and their fields. This can be an occasion to detect type confusion, or at least confusion between object instance and integral fields.

On platforms supporting the multithreading feature, a JCRE-owned thread running in background can also be considered to detect type confusions.

Such mechanisms would at least make the attacks more difficult by removing the persistent character of the fault induced. However the cost of such mechanisms has not been evaluated and may not be affordable for Java Card platforms.

Chapter 10

Securing the Execution Flow

Contents

10.1 Enhancing the Security of Conditional Branching Instructions 169
10.1.1 Context 170
10.1.2 Existing Solutions 171
10.1.3 The Proposed Method 174
10.2 Preventing the Interpretation of Injected Code 177
10.2.1 The Issue Raised by Code Injection 178
10.2.2 Scrambling the Instruction Set 178
10.2.3 Efficiency of the Method 181
10.2.4 Ensuring the Code Isolation 182

Chapter 6 exposes different ways an attacker could manage to tamper with the execution flow of an application running on a Java Card platform.

This chapter describes how a JCVM can counteract these attacks and more generally how it can increase its security level.

Section 10.1 provides a method allowing to protect the execution of conditional jumps against fault attacks at the application level.
The countermeasure described here has been designed with Christophe Giraud and has led to a patent filing [BG11].

Finally Section 10.2 describes a modification of the JCVM which allows to prevent the interpretation of injected code as exposed by Guillaume Bouffard *et al.* (Section 4.4.2). This countermeasure was designed with Philippe Andouard and has been accepted for presentation at the CHIP-TO-CLOUD'12 security forum [BA12].

10.1 Enhancing the Security of Conditional Branching Instructions

In particular, this countermeasure acts on the conditional branching instructions of the Java Card Virtual Machine operating on boolean variables:

- ifeq;
- ifne.

These instructions are defined in the VM specifications [LY99] and have been introduced in Section 5.2.

10.1.1 Context

Conditional branching instructions are generally critical from a security point of view. For instance on a Java Card platform, an application requiring the authentication of the card holder with his secret PIN is typically operated by checking the return of the `OwnerPIN.check` API method verifying the submitted PIN against the stored one. For the sake of simplicity, consider that the result of the `OwnerPIN.check` method has been stored in a boolean local variable. The verification of the user authentication is then performed as exposed in Listing 10.1

Listing 10.1: A typical (insecure) PIN verification

```
// boolean isPINValid contains the result of the PIN verification
if (isPINValid) {
    // The card holder is authenticated
    ...
} else {
    // The card holder is not authenticated
    ...
}
```

Listing 10.2 exposed the bytecode sequence generated by the compilation of the code in Listing 10.1.

Listing 10.2: A typical (insecure) PIN verification at the bytecode level

```
    bload <n>          // push isPINValid onto the operand stack
    ifeq L1            // branch at L1 if the returned value equals 0 (false)
    ...                // authenticated case
    ...
    goto L2
L1: ...                // not authenticated case
    ...
L2: ...                // end of the application
```

In theory, such an implementation is correct since it is functional. But since the Java Card specifications do not provide built-in protections against Fault Attacks, this implementation can be tampered with by an attacker.

In the context of conditional branching instructions, two categories of fault attacks must be particularly considered:

- a disruption of the execution flow jumping the conditional branching instruction resulting in a forced jump in the first defined branch;
- a disruption of the manipulated data involved in the condition evaluation resulting in a jump in one of the defined branches. This is the case exposed in Section 5.2.

The first category is usually dealt with by always setting the worst case for the attacker in the first branch so that she will not take advantage of the induced fault. With regards to the second category, one may note that programming languages usually map the boolean values `false` and `true` respectively to the integral value `0` and any non-null value. Conditional branching instructions consequently only compare the boolean value to 0 to evaluate the corresponding condition. An attacker capable of randomly modifying the value of a boolean variable has then great chances to set this variable to `true` (exactly $2^n - 1/2^n$, if a boolean value is coded with a n-bit word).

The remaining problematic is then to ensure that if the execution of a bytecode sequence is conditioned by a given variable, it will be actually executed only if this condition is true, even in the presence of multiple fault attacks.

10.1.2 Existing Solutions

The problematic of fault attacks targeting conditional instructions is not new. Consequently several countermeasures have already been developed and are actually used in the smart card industry. The following briefly describes the generic approach and points out its drawbacks.

10.1.2.1 Description

The developer of an application cannot always be aware of the level of security enforced by the various platforms on which his code is meant to run. Therefore, the generic approach to secure conditional branching at the application level is to add redundant verification of the condition.

These redundant checks can be implemented by checking twice the same condition of by checking the condition and its negation. By using this technique, the code given in Listings 10.1 and 10.2 (respectively at the Java source and bytecode level) would become as described in Listings 10.3, 10.4, 10.5, 10.6.

Regardless of the technique used, a single fault injection on the conditional branching instruction or on the condition itself is always detected by the the redundant check.

10.1.2.2 Drawbacks

The main drawback of such countermeasures is that it is still vulnerable to double fault attacks. It is generally considered as difficult to perform a double attack with different parameters for the physical perturbation. However reproducing the same perturbation twice, even in a very short time slot is not so difficult for experienced attackers. Such a

Listing 10.3: A typical (secure) PIN verification

```
if (isPINValid) {
    if (isPINValid) {
        // The card holder is authenticated
        ...
    } else {
        // An attack is detected!!
        ...
    }
} else {
    if (!isPINValid) {
        // The card holder is not authenticated
        ...
    } else {
        // An attack is detected!!
        ...
    }
}
```

Listing 10.4: A typical (secure) PIN verification at the bytecode level

```
    iload <n>          // push isPINValid onto the operand stack
    ifeq L1            // branch at L1 if the returned value equals 0 (false)
    iload <n>          // push isPINValid onto the operand stack
    ifeq L2            // branch at L2 if the returned value equals 0 (false)
    ...                // authenticated case
    ...
    goto L3
L2: ...                // means an attack occurred
    ...
    goto L3
L1: iload <n>          // push isPINValid onto the operand stack
    ifne L4            // branch at L2 if the returned value equals 1 (true)
    ...                // not authenticated case
    ...
    goto L3
L4: ...                // means an attack occurred
    ...
L3: ...                // end of the application
```

double attack can target either the manipulated data or the executed conditional branching instruction. Table 10.1 sums up the security brought by the simple and mirrored doubling countermeasures with regards to single and double attacks aiming at modifying a value or jumping an instruction.

The second worth noticing drawback is that such methods induce a costly overhead in terms of development cost since the redundant checks must be added by the developers in the high-level source code. Consequently it also needs to be thoroughly tested to ensure that no particular case has been forgotten by the developers. This raises another issue since the test cases are by nature not functional and are consequently difficult to

Listing 10.5: A typical (secure) PIN verification

```
if (isPINValid) {
    if (!isPINValid)
    {
        // An attack is detected!!
        ...
    } else {
        // The card holder is authenticated
        ...
    }
} else {
    if (isPINValid) {
        // An attack is detected!!
        ...
    } else {
        // The card holder is not authenticated
        ...
    }
}
```

Listing 10.6: A typical (secure) PIN verification at the bytecode level

```
    iload <n>          // push isPINValid onto the operand stack
    ifeq L1            // branch at L1 if the returned value equals 0 (false)
    iload <n>          // push isPINValid onto the operand stack
    ifne L2            // branch at L2 if the returned value equals !0 (true)
    ...                // means an attack occurred
    ...
    goto L3
L2: ...                // authenticated case
    ...
    goto L3
L1: iload <n>          // push ownerPIN onto the operand stack
    ifeq L4            // branch at L2 if the returned value equals 0 (false)
    ...                // means an attack occurred
    ...
    goto L3
L4: ...                // not authenticated case
    ...
L3: ...                // end of the application
```

	Data modification		Instruction jump	
	Single fault	Double fault	Single fault	Double fault
Doubling	Yes	No	Yes	No
Mirrored doubling	Yes	No	No	No

Table 10.1: Security brought by the simple and mirrored doubling countermeasures.

test.

Finally applying these countermeasures increase the size of the application, as any redundant check would do. This increase is in both case due to :

- twice the size of the attack reaction code ($S_{reaction}$),
- twice the size of the code loading the boolean value (the `bload_<n>` instruction here) ($S_{loadBoolean}$),
- two additional conditional branching instructions (3 bytes each),
- two additional branching instructions (3 bytes each).

The memory footprint increase is then of the order of $2.S_{reaction} + 2.S_{loadBoolean} + 12$ bytes.

To fewer extents, one can also note that such methods do not comply with the seminal idea behind of the Java Card language which was to allow non-specialized developers to write applications for smart cards.

10.1.3 The Proposed Method

As exposed in the previous section, dealing with single and double fault attacks against conditional branching instructions presents certain issues. The following detail a method allowing to solve most of these issues and to partially solve the remaining one.

10.1.3.1 Description

The exposed method is based on an automatic processing of conditional branching instructions operating on boolean variables at the bytecode level in the compiled Java .CLASS files. The processing of the instructions is operated in three steps:

1. Detection of the conditional branching instructions to protect;
2. Addition of instructions operating the redundant checks;
3. Addition of instructions for error handling.

Detection of the instructions to protect This first step consists in finding the conditional branching instructions operating on boolean variables in the bytecode array of the different methods in the .CLASS files. Unlike .CAP files, the .CLASS files contain information on the type of the different variables used in each method. It is then possible to parse the .CLASS files to determine the types of the different variables used in the different methods. It is then possible to parse the bytecode arrays of the different methods to identify the the conditional instructions operating on boolean variables.

The implementation of this process is made relatively trivial with the use of bytecode manipulation libraries such as CAPMAP or BCEL [Sma, BCE] for instance.

Addition of redundant checks and error handling instructions The second and third steps of the process can be operated in the same time by adding instructions in the bytecode arrays of the different methods.

The second step consists in the addition of instructions enforcing a redundant verification of each conditional instruction. The added instructions are part of the instruction set proposed by the JCVM and allow to redundantly check for each conditional branching instruction, *i.e.* for each conditional branching instruction comparing a boolean variable to 0, a bytecode sequence allowing to compare the same variable with the integral value 1, which corresponds to a boolean value `true` (as generated by the compiler).

The third step consists in adding a bytecode sequence implementing a particular behaviour to observe when an error is detected such as resetting the hardware, entering an infinite loop or erasing the NVM.

Again, the use of libraries dedicated to the manipulation of Java classes allows to ease the addition of instructions by automatically recomputing the numerous indices and offsets used in the bytecode arrays and other components of these binary files.

In a generic way, steps 2 and 3 implements Algorithm 5.

Algorithm 5: Addition of redundant checks

output: -

```
1  Build list L of boolean variables;
2  Build list L' of conditional branching instruction operating on boolean variables;
3  while more instructions left in L' do
4      ins ← next instruction;
5      duplicate pushed boolean variable;
6      if ins == ifeq then
7          insert iconst_1 instruction;
8          insert if_icmpeq instruction;
9          insert ifne instruction;
10     end
11     if ins == ifne then
12         insert iconst_0 instruction;
13         insert if_icmpeq instruction;
14         insert ifeq instruction;
15     end
16     update branch offsets;
17 end
```

Applied to the sample code given in Listing 10.2, the process generates the bytecode sequence exposed in Listing 10.7.

Listing 10.7: A typical (unsecured) PIN verification at the bytecode level

```
    iload <n>          // push isPINValid onto the operand stack
    iload <n>          // push isPINValid onto the operand stack
    ifeq L1            // branch at L1 if the returned value equals 0 (false)
    iconst_1           // push 1 onto the operand stack
    if_icmpeq L2       // branch at L2 if the boolean value equals 1 (true)
L4: ...                                  // handle error
    ...
    goto L3
L2: ...                // authenticated case
    ...
    goto L3
L1: ifne L4            // branch at L4 to handle error
    ...                // not authenticated case
    ...
L3: ...                // end of the application
```

One can note that the modifications applied in the .CLASS files would be passed on the .CAP file after conversion.

Finally the application verifies at runtime in two steps if the conditional branching instruction operand is either 0 or 1. If the results of these two steps are not coherent a specific treatment is executed.

10.1.3.2 Handling the `OwnerPIN.check` Method

Although the example of the `OwnerPIN.check` is the most meaningful when considering methods returning boolean values, it stands as an exception to carefully handle when adding redundant checks. Indeed in the proposed method the sequence of instructions resulting in the loading of the boolean value is doubled. The problem raised by the `OwnerPIN.check` method is that executing it twice would decrease twice the counter checked against the maximum number of incorrect PIN presentations.

To solve this issue, it is then necessary to launch a particular process when this method is encountered in order to double the boolean value loading by calling the `OwnerPIN.isValidated` method.

10.1.3.3 Advantages of this method

The major advantage of the proposed method is that it enhances the security level of the application with regards to the previously introduced methods.

The code generated by the process described above is highly resistant to both single and double fault attacks disrupting either the data or the instruction. The only remaining vulnerability would be that an attacker manages to set the value pushed onto the operand stack to 1 two times in a row. Considering the generally admitted fault model allowing the attacker to randomly modify a value such a situation is very unlikely to happen. The

probability of such an event is indeed of 2^{-2*n} considering n-bit values (with n typically equals to 16 or 32 for integral values).

For comparison purpose, Table 10.2 presents the security brought by the simple and mirrored doubling countermeasures and by the proposed method with regards to single and double attacks aiming at modifying a value or jumping an instruction.

	Data modification		Instruction jump	
	Single fault	Double fault	Single fault	Double fault
Doubling	Yes	No	Yes	No
Mirrored doubling	Yes	No	No	No
Proposed method	Yes	Yes	Yes	Yes

Table 10.2: Security brought by the exposed countermeasures.

The second obvious advantage of this method is that it is totally automatic and transparent for the application's developer. It consequently drastically reduces the development and code review costs. Furthermore it ensures that no code portion remains insecure.

In addition, operating at the bytecode level this method turns out to be a little more efficient in terms of memory footprint. Indeed, the error detection instructions can be factorized and the cost of redundant checks can be slightly reduced. The increase of the application size is due to :

- the size of the attack reaction code ($S_{reaction}$),
- the size of the code loading the boolean value (the `bload_<n>` instruction here) ($S_{loadBoolean}$),
- the size of the constant value loading instruction (1 byte),
- two additional conditional branching instructions (3 bytes each),
- one additional branching instruction (3 bytes).

The memory footprint increase is then of the order of $S_{reaction} + S_{loadBoolean} + 10$ bytes, which is almost half the memory footprint increase of the previous methods ($2.S_{reaction} + 2.S_{loadBoolean} + 12$ bytes).

Finally, it is worth noticing that the proposed method does not require any modification of the JCVM and is fully compliant with all released JCVMs. It is therefore particularly interesting when *offensive* or *defensive* VMs are considered.

10.2 Preventing the Interpretation of Injected Code

Section 4.4.2 has detailed an attack proposed by Guillaume Bouffard *et al.* in which the attacker manages to redirect the control flow to the content of a static byte array

corresponding to a certain bytecode instructions sequence. This attack therefore relies on the public knowledge of the instruction set supported by the JCVM.

This section exposes a method allowing to prevent such attacks, or at least to restrain it to much more powerful attackers, by scrambling the instruction set.

10.2.1 The Issue Raised by Code Injection

The instruction set supported by the JCVM is publicly released as part of the JCVM specification. Therefore an attacker has the knowledge of the mapping between bytecode instructions (`aload_0`, `invokevirtual`, `ifeq`, `istore_2` for instance) and byte values (respectively `0x2A`, `0xB6`, `0x99`, `0x3D`).

Considering an attacker capable of redirecting the execution flow to a specific byte array, as exposed in Section 4.4.2 by disrupting the parameter of a `goto_w` instruction, this knowledge is of great value. Indeed, the attacker can set up any bytecode instructions sequence in her byte array by assigning the adequate byte values, thus circumventing the security brought by the bytecode verifier (should this process be executed on- or off-card).

10.2.2 Scrambling the Instruction Set

Based on this state of fact, the method proposed here consists in assigning a particular instruction set to each application loaded on the platform.

10.2.2.1 The Scrambling Process

To do so, the first requirement is to generate a permutation of the instruction set. Different approaches can be used in that purpose depending on the security requirement:

- XORing a given random byte to the values defined in the specifications, which is a lightweight process allowing to obtain 2^8 different instruction sets;
- generating a random permutation of the specified instruction set, which is a much more costly process [Knu97] allowing to obtain $!256$ different instruction sets.

Regarding the cost of the instruction set generation, one can nevertheless note that such a process is only executed at application load-time, that is to say once in the entire application life cycle.

Once the new instruction set is generated, the platform only has to reflect the modification of the instruction set to the bytecode arrays defined within the application being loaded. This consequently requires to parse these bytecode arrays as part of the loading process, which also represent an additional overhead cost. Nevertheless this process can be executed at almost no cost if the platform executes the verification of the bytecode at load-time.

Listings 10.8 and 10.9 illustrates the modification of an application [1] after the new instruction set has been applied. For sake of simplicity, the scrambled instruction set has been generated here by applying a XOR with the byte 0x7D to the standard instruction set.

Listing 10.8: The original bytecode instructions sequence

```
2B        aload_1
B6 0006   invokevirtual #6 <javacard/framework/APDU.getBuffer>
4D        astore_2
2C        aload_2
03        iconst_0
33        baload
9A 0014   ifne 20 (+12)
2C        aload_2
04        iconst_1
33        baload
10 A4     bipush 164
A0 0014   if_icmpne 20 (+4)
B1        return
2B        aload_1
B6 0007   invokevirtual #7 <javacard/framework/APDU.setIncomingAndReceive>
57        pop
2C        aload_2
07        iconst_4
33        baload
3E        istore_3
10 07     bipush 7
BC 08     newarray 8 (byte)
59        dup
...
```

In addition, the scrambling process can be enhanced in order to associate the different occurences of the same instruction to different values. This can be easily achieved by XORing the offset of the instruction within the bytecode array to its associated value, hence the permutation is diversified with these offsets. Listing 10.10 illustrates the result of this operation on the scrambled bytecode array for the same method.

Note that a simple way to deal with the case where the offset of an instruction is greater than 255 (*i.e.* `0xFF`) consists in taking into account the XOR of the different bytes of this value. Furthermore, considering that the offset of an instruction within the bytecode array might be known by an attacker, other diversificating values can be considered such as the physical address of the instruction within the card's memory for instance.

10.2.2.2 Instruction Set Switch at Runtime

As each on-card application is assigned a particular instruction set, it is necessary to operate an instruction set switch when the system operates an application switch, such

[1] A part of the `process` method defined in the HelloWorld classic applet sample of the JCDK 3.0.1.

Listing 10.9: The modified bytecode instructions sequence

```
56       aload_1
CB 0006  invokevirtual #6 <javacard/framework/APDU.getBuffer>
30       astore_2
51       aload_2
7E       iconst_0
4E       baload
E7 0014  ifne 20 (+12)
51       aload_2
79       iconst_1
4E       baload
6D A4    bipush 164
DD 0014  if_icmpne 20 (+4)
CC       return
56       aload_1
CB 0007  invokevirtual #7 <javacard/framework/APDU.setIncomingAndReceive>
2A       pop
51       aload_2
7A       iconst_4
4E       baload
43       istore_3
6D 07    bipush 7
C1 08    newarray 8 (byte)
24       dup
...
```

as when calling shared methods. If this instruction set switch is not operated, any shared method would be considered as malicious injected code.

These occasions correspond to what is referred in the JCRE specifications as *context switches* [Sun09e, §6.9.1]. Indeed, because of the application isolation principle, the platform is required to keep tracks of the potential switches from an application to another.

From an optimization point of view, the instruction set switches can then be operated as part of the application context switch. However, Section 10.2.4 will show that in order to benefit from the instruction set scrambling to enforce the application code isolation, the application context switch and instruction set switch should be performed in two different steps.

10.2.2.3 The Library Exception

As the Java Card specifications allows the definition of shared libraries, these should be avoided within the bytecode arrays modification process.

Listing 10.10: The scrambled and diversified bytecode instructions sequence

```
56        aload_1
CA 0006   invokevirtual #6 <javacard/framework/APDU.getBuffer>
34        astore_2
54        aload_2
78        iconst_0
49        baload
EF 0014   ifne 20 (+12)
5A        aload_2
75        iconst_1
43            baload
63 A4     bipAush 164
CD 0014   if_icmpne 20 (+4)
DF        return
42        aload_1
DE 0007   invokevirtual #7 <javacard/framework/APDU.setIncomingAndReceive>
32        pop
48        aload_2
60        iconst_4
55        baload
5F        istore_3
70 07     bipush 7
DE 08     newarray 8 (byte)
05        dup
...
```

10.2.3 Efficiency of the Method

10.2.3.1 From the security point of view

Since the attacker does not know the instruction set used by her application, she cannot predict the behaviour of the code she injects, thus the harder it is to mount a specific attack. In addition, the interpretation of injected code is likely to lead to the following errors:

- an undefined instruction has been encountered;
- stack over-/underflow;
- an unknown local variable index has been encountered;
- virtual method link error;
- ...

If the platform does perform checks to detect such errors it would then halt during the interpretation of injected code. Furthermore, to increase the probability that such an error would occur, the permutation can enforce the principle of making the various `aload` instructions of the standard instruction set undefined in the permuted one.

However, the countermeasure may be broken if the attacker is capable of dumping the memory of the system, since knowing the original code and the modified code of her own

application she would then easily deduce the instruction set permutation. She would only have then to update the byte array containing the injected code to take the permutation into account.

10.2.3.2 From the time and memory consumption points of view

This method only requires a supplementary indirection at runtime, as part of the instruction decoding step in a standard fetch-decode-execute process, and either 1 or 256 bytes to store, whether the instruction set is shuffled by a single byte XORing or a random permutation.

10.2.4 Ensuring the Code Isolation

The instruction set scrambling method also allows to enforce the code isolation principle to a certain extent.

By assigning a random instruction set to each on-card application, this method implicitly make every applications consider the other as if they were injected code. Therefore the same consequences would appear if ever the code isolation mechanism was to be circumvented by an attacker, which would lead the JCVM to halt.

Considering the attack described in Section 7.1, even if the attacker was successful in the final application firewall disruption, she would not be able to have the server-side method `setValue` executed. This holds only if we consider that the application context switch and the instruction set switch are performed in two different steps, else operating one implicitly operates the other and both mechanisms are disrupted.

Conclusion to Part III

As a matter of fact, it appears of the utmost importance to consider fault attacks as well as combined attacks when designing secure Java Card platforms. The mechanisms required to face this threat vary depending on the platform's philosophy, that is to say whether the embedded virtual machine can be qualified as offensive, defensive or defending with regards to Definitions 2, 3 and 4. If the offensive and defensive virtual machines can only rely on a thorough verification of the applications loaded on card, possibly with various static analysis tools, Chapter 8 has shown that a defending virtual machine can be relatively easily protected against identified threats.

Chapter 9 has exposed different countermeasures to enforce the type safety property. First a protection against the attack described in Section 5.2 have been detailed. This countermeasure aims at ensuring the integrity of the values pushed onto and popped from the operand stack. Since this countermeasure must be implemented within the Java Card Virtual Machine, it is only compatible with the defending virtual machine approach. However, the countermeasures described so far only protect the type safety property against known attacks. In a second time we have discussed possible directions allowing to build an intrinsically type-safe platforms, or at least platforms that would eventually detect type confusions and react in consequences.

Then we have presented mechanisms allowing to prevent the modification of the execution flow of an application in Chapter 10. The first mechanism described is meant to protect conditional branching instructions. This mechanism operates at the application level and is therefore suitable for offensive and defensive virtual machines as well as for defending virtual machines that would not provide a particular security mechanism on conditional branching occasions. As exposed, this mechanism has a certain cost in terms of both time and memory footprint. However the proposed method is more efficient than the usual redundant checks added in applications code by their developers. The second mechanism proposed consists in assigning a random instruction set to each different application. This mechanisms is meant to prevent the correct interpretation of code injected for instance by disrupting branching instruction to jump in a particular byte array. We can note that this mechanism can also participate in the application code isolation mechanism.

Conclusions & Perspectives

Since its introduction in 1996, Java Card has grown to become nowadays the world leading platform in the smart card application field. Although one should not see any relation between these two advances, it is the very same year Java Card was introduced that Paul Kocher published the article that initiated the concern of the embedded systems security community for side-channel attacks and subsequently for fault attacks, mainly against embedded cryptographic implementations. Smart cards being devoted to ensure the security of the protocols they support, actors of the smart card industry have to keep up with the advances made by the scientific community in these two fields in order to step up with new attacks and to integrate new countermeasures as soon as possible.

The applicative facilities offered by Java Card platforms have drawn the attention of security analysts for almost a decade. However, as every embedded systems, Java Cards are not intrinsically secured against side-channel and fault attacks. Contrariwise attackers may take advantage of both logical and physical attacks to set up more compromising attack *scenarii*. We call such attacks Combined Attacks.

This PhD thesis has been dedicated to the evaluation and eventually the enhancement of the security of Java Cards against Fault and Combined Attacks. The security analysis we have undertaken in a first step has allowed us to reach the conclusion that several mechanisms ensuring the security of the Java Card platforms and of the hosted applications are likely to be circumvented in multiple ways by appropriate combinations of software and hardware attacks. In particular, we have been able to publish the first practically achieved Combined Attack on a sample Java Card 3 Connected Edition platform.

With regards to the results of our security analysis, we have then proposed different countermeasures against the attacks highlighted in the second part of this dissertation. Meant to resist to fault attacks, these countermeasures are all somehow based on redundant checks. However efforts have been made in order not to penalize too much the performance of the platform.

When designing countermeasures against Fault and Combined Attacks for Java Card platforms, two approaches can be envisaged depending on the characteristic of the targeted platform. On the one hand, offensive and defensive platforms only accept applications passing various static analysis tools and somehow require that the application defends itself against attacks. On the other hand, defending platforms *a priori* accept any application passing the bytecode verification process and defends the hosted applications against attacks, to a certain extent. The first approach is definitely more efficient when considering the execution of a trusted application in a safe environment. However, if redundant checks were to be added to this application at the Java language level so that it would pass static analysis tools validating the resistance of the application against fault attacks, the consequences on both the memory footprint and the performance of this application would be disastrous. We can then only argue for the defending virtual machine approach when considering potentially intrusted applications and fault injections. Yet this defending virtual machine needs to be aware of the numerous threats and step up with the evolution of the Java Card security field.

The introduction of fault attacks in the Java Card field represents an important evolution. The enhancement of present fault injection techniques such as the development of

electromagnetic fault induction and the generalization of multiple fault injection is likely to endow attackers with even greater capacities that would outdate simple countermeasures such as basic redundant checks.

In addition, the eventual arrival of truly open platforms, even if certain restrictions may limit the possibilities offered to *intrusted* applications, also let us presage of new developments in the Java Card security field. Furthermore the engaged introduction of Java Card in the web ecosystem with the Java Card 3 Connected Edition stands as another challenge. Indeed, once achieved this process will not only increase the attack surface of the platform but it will also most likely draw the attention of a new class of experienced attackers. Securing the platforms will then only be achieved at the price of a thorough vulnerability analysis with regards to state-of-the-art logical and physical attacks as well as a complete integration of the legacy of web security.

Finally, we can only expect that CAs similar to those described in this thesis are to be implemented on other virtual machines. This statement is obvious indeed regarding the standard Java VM since the seminal work presented in [GA03] was set in this context. But the exploitation of similar concepts on Java ME VMs embedded on mobile phones and even on the Dalvik VM embedded on Android smartphones are surely promising topics.

Appendix A

Practical Validation of the Fault Model

The experimental results we present here have been obtained on a recent ARM-based smartcard device. A Java Card 2.2.2 Virtual Machine is running on the device. The fault injection was achieved with a laser equipment.

A.1 The test application.

We intend to provide an experimental validation of the previously introduced fault model. We then develop a Java Card applet, the `process` method of which is exposed below.

```
1. public void process(APDU apdu) {
2.   [...]
3.   ref = Util.getShort(buffer, OFFSET_CDATA);
4.   target = Util.getShort(buffer, (short) (OFFSET_CDATA+2));
5.   ref = ref;     // push and pop ref : sload #ref
                                        sstore #ref
6.   res = target;  // push and pop target : sload #target
                                              sstore #res
7.   Util.setShort(buffer, OFFSET_CDATA, res);
8.   [...]
9. }
```

The first step of our applet consists then in pushing a reference value onto the operand stack and popping it. Therefore we know which value was written in the operand stack before we proceed. Then we push a second value onto the stack. This second push is the target of our fault injection. The subsequent popped value which is stored in variable `res` is then expected to be an erroneous value. The last step of the applet process is then to send this value out of the card.
Fig. A.1 illustrates the evolution of the operand stack along the execution of lines 5 and 6.

A.2 Experimental results.

To evaluate the validity of our fault model, we perform several fault attacks with different parameters (namely, the time and space parameters of the attack as well as the width and intensity of the laser

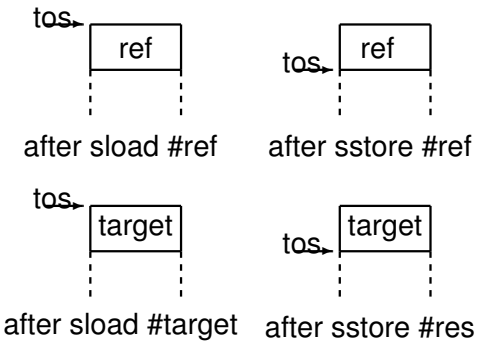


Figure A.1: Evolution of the operand stack content and of the top-of-stack (tos) along execution of lines 5 and 6 of the test applet.

0x00F1	0x0000	0x00F2	0x00CC	0x149C	0x0121	0x0D88	0xFF19
0x0006	0x0129	0x149E	0xEA00	0x00BB	0x27FF	0x168F	0x000D
0x1490	0x4778	0x0011	0xD203	0xC14A	0x00D9	0xAABC	0x1200
0x2B2B	0x0012	0x5576	0xBB00	0x6000	0x7600	0xFFFF	0xAA0E
0xAA8B	0x2AAF	0xAAEC	0xAAEB	0xABB0	0x2AAE	0xAB6B	0xEEA0

Table A.1: Results (`res` values) of the fault attacks with various fault injection parameters and fixed `ref` and `target` values.

beam) and different input parameters.
Table A.1 sums up the different results obtained. In this table we only present the results regarding a given couple of input : (`ref`, `target`) = (`0xAABB`, `0xCCEE`). The cells highlighted in grey within Table A.1 denote the results that were dependent on the inputs. We also highlighted the results 0x0000 and 0xFFFF which correspond to the two *stuck-at* fault models. Note that we only present in this table the results that were reproducible.

A.3 Conclusions.

It is difficult to deduce the exact perturbation caused by the laser beam in all cases, especially when the results are not correlated with the inputs. Such results may be correlated with internal values contained in the registers of the processor at the time the laser is activated.
On the other hand, some results can be easily interpreted since they are compound of different chunks of either the reference or the target value and 0s (for instance, `0x0000`, `0x00CC`, `0xBB00`).

To conclude, the results we obtained only partially validate our fault model since `res` is either all-0, all-1, truncated `ref`, truncated `target` or other unknown values.
However this proves that an adversary can manage to disturb the push operation and may have a certain control on the erroneous operand eventually pushed.

Appendix B

Implementation of the Multithreaded Attack on Java Card 3 Connected Edition

B.1 Array Forgery

The classes loaded wit the attacker's application :

```
public class ArrayContainer {
    byte[] array;
}

public class ForgeryContainer {
    Forgery f;
}

public class Forgery {
    int field_1, field_2;
}
```

The code within the attacker's application to forge the array in the volatile memory :

```
ArrayContainer ac = new ArrayContainer();
ac.array = new byte[1];
ForgeryContainer fc = (ForgeryContainer) (Object) ac;
fc.f.field_1 = 0x100; // set the length of ac.array to 256
fc.f.field_2++;       // increment the memory pointer of ac.array
// Access to memory through ac.array[i]
```

B.2 Request Flooding Validation Thread

The run method of the thread used to validate the influence of communication :

```
public void run() {
    i = 0;
    startTime = System.currentTimeMillis();
    while ((System.currentTimeMillis() - startTime) < TIME_BOUND) {
        i++;
    }
}
```

The Python *ping flooder* :

```
def flood(host, url, delay, socket, ID, count, ping_delay)
    # Send request to target application
    conn = httplib.HTTPConnection(host)
    conn.request("GET", url)

    # Wait
    time.sleep(delay)

    # Send ping flood
    for i in xrange(count):
        send_ping(ID, socket, host)
        time.sleep(ping_delay)
```

Publications

[BA12] Guillaume Barbu and Philippe Andouard. Instruction Randomization, or When the Attacker Gets Lost in Translation. Accepted Talk at Chip-to-Cloud'12 Security Forum, September 2012.

[Bar09] Guillaume Barbu. Fault Attacks on Java Card 3 Virtual Machine. Accepted Talk at e-Smart'09, September 2009.

[Bar10] Guillaume Barbu. Combined Attacks on Java Card 3 - Type Confusion Issues. Accepted Talk at e-Smart'10, September 2010.

[BDH11] Guillaume Barbu, Guillaume Duc, and Philippe Hoogvorst. Java Card Operand Stack: Fault Attacks, Combined Attacks and Countermeasures. In *Smart Card Research and Advanced Applications, 10th International Conference – CARDIS 2011*, volume 7079 of *LNCS*, pages 297–313. Springer Verlag, 2011.

[BG11] Guillaume Barbu and Christophe Giraud. Procédé et Système de Sécurisation d'une Application Logicielle comprenant une Instruction Conditionnelle basée sur une Variable Booléenne. French patent FR1158517, September 2011.

[BGG12] Guillaume Barbu, Christophe Giraud, and Vincent Guerin. Embedded Eavesdropping on Java Card. In *Proceedings of the IFIP International Information Security and Privacy Conference 2012 – SEC 2012*, LNCS. Springer Verlag, 2012.

[BHD11] Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc. Application-Replay on Java Card 3, When the Garbage Collector Gets confused. Accepted Talk at e-Smart'11, September 2011.

[BHD12a] Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc. Application-Replay Attack on Java Cards: When the Garbage Collector Gets Confused. In G. Barthe and B. Livshits, editors, *Proceedings of the International Symposium on Engineering Secure Software and Systems – ESSoS 2012*, volume 7159 of *LNCS*, pages 1–13. Springer Verlag, 2012.

[BHD12b] Guillaume Barbu, Philippe Hoogvorst, and Guillaume Duc. Tampering with Java Card Exception - The `Exception` Proves the Rule. In *Proceedings of the International Conference on Security and Cryptography – SECRYPT'12*. SciTePress Digital Library, 2012.

[BT11] Guillaume Barbu and Hugues Thiebeauld. Synchronized Attack on Multithreaded Systems - Application to Java Card 3.0 -. In *Smart Card Research and Advanced Applications, 10th International Conference – CARDIS 2011*, volume 7079 of *LNCS*, pages 18–33. Springer Verlag, 2011.

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on Java Card Combining Fault and Logical Attacks. In *Smart Card Research and Advanced Application Conference – CARDIS 2010*, volume 6035 of *LNCS*, pages 148–163. Springer Verlag, 2010.

Bibliography

[AFV07] F. Amiel, B. Feix, and K. Villegas. Power Analysis for Secret Recovering and Reverse Engineering of Public Key Algorithms. volume 4876 of *LNCS*, pages 110–125. Springer-Verlag, 2007.

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES Secure against some Attacks. In Koç et al. [KNP01], pages 309–318.

[ANN05] N. Asokan, V. Niemi, and K. Nyberg. Man-in-the-middle in tunnelled authentication protocols. In Bruce Christianson, Bruno Crispo, James Malcolm, and Michael Roe, editors, *Security Protocols*, volume 3364 of *LNCS*, pages 28–41. Springer Berlin / Heidelberg, 2005.

[ASM] The ASM Project. `http://asm.ow2.org/`.

[BBKL11] M. Barreaud, G. Bouffard, N. Kamel, and J.-L. Lanet. Fuzzing on the http protocol implementation in mobile embedded web server. pages 14–27, 2011.

[BCE] The Byte Code Engineering Library, Appache™Commons. `http://commons.apache.org/bcel/`.

[BCO04] Éric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. volume 3156 of *LNCS*, pages 16–29. Springer-Verlag, 2004.

[BD04] Gilles Barthe and Guillaume Dufay. A tool-assisted framework for certified bytecode verification. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2004.

[BDL97] Dan Boneh, Rene DeMillo, and Robert Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology - EUROCRYPT'97*, volume 1233 of *LNCS*, pages 37–51. Springer-Verlag, 1997.

[BECN+06a] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *IEEE*, 94(2):370–382, 2006.

[BECN+06b] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstal, and Claire Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE, 94(2),*, pages 370–382, 2006.

[Bel96] Bellcore. New Threat Model Breaks Crypto Codes. Press Release, September 1996.

[BICL11] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined Software and Hardware Attacks on the Java Card Control Flow. In *Smart Card Research and Advanced Applications, 10th International Conference – CARDIS 2011*, volume 7079 of *LNCS*. Springer Verlag, 2011.

[BL12] Guillaume Bouffard and Jean-Louis Lanet. *The Next Smart Card Nightmare, Logical Attacks, Combined Attacks, Mutant Applications and Other Funny Things*. 2012.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. *RFC 1945: HyperText Transfer Protocol – HTTP/1.0*. The Internet Engineering Task Force (IETF), 1996.

[BLMM94] T. Berners-Lee, L. Masinter, and M. McCahill. *RFC 1738: Uniform Resource Locators (URL)*. The Internet Engineering Task Force (IETF), 1994.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. volume 1294 of *LNCS*, pages 513–525. Springer-Verlag, 1997.

[Cal12] Andrew Calafato. An analysis of the vulnerabilities introduced with the java card 3 connected edition. Msc thesis, Royal Holloway, University of London, March 2012.

[CBR02] L. Casset, L. Burdy, and A. Requet. Formal development of an embedded verifier for java card byte code. 2002.

[Che00] Zhiqun Chen. *Java Card Technology for Smart Cards, Architecture and Programmer's Guide*. Addison-Wesley, 2000.

[CHS03] Serge Chaumette, Iban Hatchondo, and Damien Sauveron. JCAT: An Environment for Attack and Test on Java Card. pages 270–275, August 2003.

[CM05] Nikolaj Cholakov and Dimo Milev. The Evolution of the Java Security Model. In *Proceedings of the International Conference on Computer Systems and Technologies (CompSysTech'2005)*, 2005.

[CNA06] CNAM/CEDRIC, POPS CNRS/INRIA/USTL and Trusted Logic S.A. The MESURE Project. Available at `http://mesure.gforge.inria.fr/`, 2006.

[CPR07] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side Channel Cryptanalysis of a High Order Masking Scheme. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007.

[CS05] Serge Chaumette and Damien Sauveron. An Efficient and Simple Way to Test the Security of Java Card. In *Proceedings of WOSIS'05*, pages 331–341, May 2005.

[CV01] Denis Caromel and Julien Vayssière. Reflection on MOPs, Components, and Java Security. In *Proceedings of the Engineering C of Object-Oriented Programs (ECOOP)*, volume 2072 of *LNCS*. Springer-Verlag, 2001.

[DC66] R.L. Dennis and System Development Corporation. *Security in the Computer Environment*. Clearinghouse for Federal Scientific & Technical Information, 1966.

[dC12] Bertrand du Castel. Personal History of the Java Card, 2012. French version originally published in MISC magazine, HS-2, Nov. 2008.

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[DLV03] Pierre Dusart, Gilles Letourneux, and Olivier Vivolo. Differential Fault Analysis on AES. volume 2846 of *LNCS*, pages 293–306. Springer-Verlag, 2003.

[DPRS11] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. Univariate Side Channel Attacks and Leakage Modeling. *Journal of Cryptographic Engineering*, 1:123–144, 2011.

[DR06] T. Dierks and E. Rescorla. *RFC 4346: The Transport Layer Security (TLS) Protocol Version 1.1*. The Internet Engineering Task Force (IETF), 2006.

[DR08] T. Dierks and E. Rescorla. *RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2*. The Internet Engineering Task Force (IETF), 2008.

[EMV11] EMVCo. *Integrated Circuit Card Specifications for Payment Systems*. November 2011.

[EPW10] Thomas Eisenbarth, Christof Paar, and Björn Weghenkel. Building a Side Channel Based Disassembler. In *Transactions on Computational Science X*, volume 6340 of *LNCS*, pages 78–99. Springer-Verlag, 2010.

[Eur10] European Telecommunications Standards Institute. ETSI TS 102 613 V9.1.0 (2010-04); Smart Cards; UICC - Contactless Front-end (CLF) Interface; Part 1: Physical and data link layer characteristics (Release 9), 2010.

[Eur11a] European Telecommunications Standards Institute. ETSI TS 102 223 V10.5.0 (2011-09); Card Application Toolkit (CAT) (Release 10), 2011.

[Eur11b] European Telecommunications Standards Institute. ETSI TS 102 241 V9.1.0 (2011-05); UICC Application Programming Interface (UICC API) for Java Card (Release 9), 2011.

[Eur11c] European Telecommunications Standards Institute. ETSI TS 131 111 V10.4.0 (2011-11); Universal Subscriber Identity Module (USIM) Application Toolkit (USAT) (Release 10), 2011.

[Eur11d] European Telecommunications Standards Institute. ETSI TS 131 130 V10.1.0 (2011-07); (U)SIM Application Programming Interface ((U)SIM API) for Java Card (Release 10), 2011.

[Fed99] Federal Information Processing Standards. *FIPS 46-3, Data Encryption Standard (DES)*. 1999.

[Fed01] Federal Information Processing Standard #197. *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology, 2001.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616: HyperText Transfer Protocol – HTTP/1.1*. The Internet Engineering Task Force (IETF), 1999.

[FKK11] A. Freier, P. Karlton, and P. Kocher. *RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0*. The Internet Engineering Task Force (IETF), 2011.

[FV10] Émilie Faugeron and Sébastien Valette. How to hoax an off-card verifier. Accepted Talk at e-Smart'10, September 2010.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using Memory Errors to Attack a Virtual Machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy – SP'03*, page 154, Washington, DC, 2003.

[Gad05] K.O. Gadellaa. Fault Attacks on Java Card : An Overview of the Vulnerabilities of Java Card Enabled Smartcards against Fault Attacks. Msc thesis, Technische Universiteit Eindhoven, Department of Mathematics and Computer Science, Aug. 2005.

[GD82] Helmut Gröttrup and Jürgen Dethloff. Einrichtung zur Durchführung von Bearbeitungsvorgängen mit wenigstens einem Identifikanden und einer Vorrichtung. German patent DE2760486C2, 1982.

[Gia04] Lucille A. Gianuzzi. *Introduction to Focused Ion Beam - Instrumentation, Theory, Techniques and Practice*. Springer, 2004.

[Gir04] Christophe Giraud. DFA on AES. volume 3373 of *LNCS*, pages 27–41. Springer-Verlag, 2004.

[GJSB05a] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addiosn-Wesley, 3rd edition, 2005.

[GJSB05b] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™Language Specification*. Addison-Wesley, third edition, 2005.

[Glo09] GlobalPlatform Inc. *GlobalPlatform Card Technology Secure Channel Protocol 03 Card Specification v2.2 - Amendment D, Version 1.1*. September 2009.

[Glo11a] GlobalPlatform Inc. *Global Platform Card Specification Version 2.2.1*. January 2011.

[Glo11b] GlobalPlatform Inc. *GlobalPlatform Card Technology Java Card API and Export File for Card Specification v2.2.1 (org.globalplatform) v1.5*. January 2011.

[Glo12a] GlobalPlatform Inc. *GlobalPlatform Card Technology Contactless Services Card Specification v 2.2 - Amendment C, Version 1.01*. February 2012.

[Glo12b] GlobalPlatform Inc. *GlobalPlatform Card Technology Java Card Contactless API and Export File for Card Specification v2.2.1 (org.globalplatform.contactless) v1.1*. February 2012.

[GM05] Zvi Gutterman and Dahlia Malkhi. Hold Your Sessions: An Attack on Java Session-Id Generation. In *Proceedings of the Cryptographer's Track at the RSA Conference (CT-RSA)*, LNCS. Springer, 2005.

[GMO01] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic Analysis: Concrete Results. In Koç et al. [KNP01], pages 251–261.

[GP99] Louis Gobin and Jacques Patarin. DES and Differential Power Analysis (the Duplication Method). In Ç.K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES '99*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[GPV06] Gilles Grimaud, Pierre Paradinas, and Eric Vétillard. Measuring the performance of the Java Card Platform. In *Java One*, May 2006.

[GSM+10] Sylvain Guilley, Laurent Sauvage, Julien Micolod, Denis Réal, and Frédéric Valette. Defeating any Secret Cryptography with SCARE Attacks. volume 6212 of *LNCS*, pages 273–293. Springer-Verlag, 2010.

[GT04] Christophe Giraud and Hugues Thiebeauld. A Survey on Fault Attacks. In *Smart Card Research and Advanced Application Conference (CARDIS04)*, LNCS, pages 159–176. Springer Verlag, 2004.

[Hab65] D.H. Habling. The Use of Lasers to Simulate Radiation-Induced Transients in Semicondictor Devices and Circuits. *IEEE Transactions on Nuclear Science*, 12:91–100, 1965.

[HJMP10] Laurent Hubert, Thomas Jensen, Vincent Monfort, and David Pichardie. Enforcing Secure Object Initialization in Java. In *Proceedings of the European Symposium on Research in Computer Securiy*, ESORICS'10, pages 101–115. Springer-Verlag, 2010.

[HM09] Jip Hogenboom and Wojciech Mostowski. Full memory read attack on a java card. In *4th Benelux Workshop on Information and System Security Proceedings (WISSEC'09)*, 2009.

[HR06] M. Handley and E. Rescorla. *RFC 4732 : Internet Denial-of-Service Considerations*. The Internet Engineering Task Force (IETF), 2006.

[Hyp03] Konstantin Hyppönen. Use of Cryptographic Codes for Bytecode Verification in Smartcard Environment. Msc thesis, University of Kuopio, Jun. 2003.

[ICL10] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developping a Trojan Applet in a Smart Card. *Journal in Computer Virology*, 2010.

[ISO98] ISO/IEC 7816. *Identification Cards - Integrated Circuit(s) Cards with Contacts*. 1998.

[ISO00] ISO/IEC 14443. *Information Technology - Identification Cards - Contactless Integrated Circuit(s) Cards - Proximity Cards*. 2000.

[ISO03] ISO/IEC 7810. *Identification Cards - Physical Characteristics*. 3^{rd} edition, November 2003.

[ISO04] ISO/IEC 18000. *Information Technology - Radio Frequency Identification for Item Management*. 2004.

[JLQ99] Marc Joye, Arjen K. Lenstra, and Jean-Jacques Quisquater. Chinese Remaindering Based Cryptosystems in the Presence of Faults. *Journal of Cryptology*, 12 (4):241–245, 1999.

[JQYY02] Marc Joye, Jean-Jacques Quisquater, Sung-Ming Yen, and Moti Yung. Observability Analysis - Detecting when Improved Cryptosystems Fail. volume 2271 of *LNCS*, pages 17–29. Springer-Verlag, 2002.

[Kah96] D. Kahn. *The Codebreakers: The Story of Secret Writing*. Scribner, 1996.

[KJJ98] Paul Kocher, James Jaffe, and Ben Jun. Introduction to differential power analysis and related attacks, 1998. `http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf`.

[KJJR11] Paul Kocher, James Jaffe, Ben Jun, and Pankaj Rohtagi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1 (1):5–27, 2011.

[KNP01] Ç.K. Koç, D. Naccache, and C. Paar, editors. *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*. Springer, 2001.

[Knu97] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113. Springer-Verlag, 1996.

[Lad97] Mark D. Ladue. When Java was One: Threats from Hostile Bytecode. In *Proceedings of the 20th National Information Systems Security Conference*, pages 104–115, 1997.

[Las02] Last Stage of Delirium Research Group. Java and Java Virtual Machine Security Vulnerabilities and their Exploitation Techniques. In *BlackHat Conference*, 2002.

[Ler01] Xavier Leroy. Java byte-code verification: an overview. 2102:265–285, 2001.

[Ler02] Xavier Leroy. Bytecode verification on java smart cards. *Software Practive & Experience*, 32:319–340, 2002.

[Lib12] Joint Interpretation Library. Application of Attack Potential to Smartcards, Jan. 2012.

[LL05] Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. Technical report, USENIX, 2005.

[LMS+11] Fred Long, Dhruv Mohlndra, Robert C. Seacord, Dean F. Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Carnegie Mellon Software Engineering Institue (SEI) series. Addison-Wesley, 2011.

[LY99] Tim Lindholm and Frank Yellin. *The Java™Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[MF00] Gary McGraw and Edward W. Felten. *Getting Down to Business with Mobile Code*. John Wiley & Sons, 2000.

[MG08] Radu Muresan and Stefano Gregori. Protection Circuit against Differential Power Analysis Attacks for Smart Cards. *IEEE Transactions on Computers*, 57:1540–1549, 2008.

[MNTT02] N. Maltesson, D. Naccache, E. Trichina, and C. Tymen. Applet verification strategies for ram-constrained devices. 2587:118–137, 2002.

[Mor74] Roland Moreno. Procédé et dispositif de commande électronique. French patent FR2266222, March 1974.

[MP07] Wojciech Mostowski and Erik Poll. Testing the Java Card Applet Firewall. Technical report, December 2007. Available at `https://pms.cs.ru.nl/iris-diglib/src/icistechreports.php`.

[MP08] Wojciech Mostowski and Erik Poll. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In *Smart Card Research and Advanced Application Conference – CARDIS 2008*, LNCS, pages 1–16. Springer Verlag, 2008.

[MS98] Nimisha V. Mehta and Karen R. Sollins. Expanding and Extending the Security Features of Java. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[Nec97] G. Necula. Proof-carrying code. pages 106–119, 1997.

[Oak01] Scott Oaks. *Java Security*. O'Reilly, second edition, 2001.

[Ora] Oracle Inc. Java Card Development Kit. Available at `http://java.sun.com/javacard/devkit/`.

[PQ03] G. Piret and J.-J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In C.D. Walter, Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

[Pri] Princeton University, Department of Computer Science, Secure Internet Programming Group. Reports on Security Flaws in Commercial Available Softwares.

[QS01] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *Smart Card Programming and Security – E-Smart 2001*, volume 1240 of *lncs*, pages 200–210. sv, 2001.

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy Current for Magnetic Analysis with Active Sensor. In *e-Smart 2002*, 2002.

[RE03] Wolfgang Rankl and Wolfgang Effing. *Smart Card Handbook*. John Wiley & Sons, Ltd, third edition, 2003.

[reJ] The REJAVA Project. `http://rejava.sourceforge.net/`.

[Res00] E. Rescorla. *RFC 2818: HTTP over TLS*. The Internet Engineering Task Force (IETF), 2000.

[RR98] E. Rose and K.H. Rose. Lightweight bytecode verification. 1998.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Sí0] Ahmadou Al Khary Séré. *Tissage de Contremesures pour Machines Virtuelles Embarquées*. Phd thesis (french), Université de Limoges, September 2010. Numéro 29-2010.

[SA02] Sergei Skorobogatov and Ross Anderson. Optical Fault Induction Attack. In B.S. Kaliski Jr., Ç.K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer, 2002.

[Sau01] Damien Sauveron. Sécurité et Vérification d'Applications Embarquées en Environnement Java Card. Msc thesis (french), Université de Bordeaux I, June 2001.

[Sau04] Damien Sauveron. *Étude et Réalisation d'un Environnement d'Expérimentation et de Modélisation pour la Technologie Java Card. Applications à la Sécurité*. Phd thesis (french), Université de Bordeaux I, Décembre 2004. Numéro 2930.

[Sav11] Aymerick Savary. Automatic Generation of Vulnerability Tests for the Java Card Byte Code Verifier. Accepted Talk at e-Smart'11, September 2011.

[SER] The SERP Project. `http://serp.sourceforge.net/`.

[SGM09] Laurent Sauvage, Sylvain Guilley, and Yves Mathieu. Electromagnetic radiations of fpgas: High spatial resolution cartography and attack on a cryptographic module. *ACM Trans. Reconfigurable Technol. Syst.*, 2(1):4:1–4:24, March 2009.

[Sha00] Adi Shamir. Protecting Smart Cards from Passive Power Analysis with Detached Power Supplies. In Ç.K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES2000)*, volume 1965 of *LNCS*, pages 71–77. Springer-Verlag, 2000.

[SICL09] Ahmadou Al Khary Séré, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Automatic Detection of Fault Attack and Countermeasures. In *Proceedings of the 4th Workshop on Embedded Systems Security – WESS'09*, pages 1–7, 2009.

[SLIC10] Ahmadou Al Khary Séré, Jean-Louis Lanet, and Julien Iguchi-Cartigny. Checking the Paths to Identify Mutant Application on Embedded Systems. In *Proceedings of the International Conference on Security Technology – SecTech'10*, volume 6485 of *LNCS*, pages 459–468. Springer Verlag, 2010.

[SLIC11] Ahmadou Al Khary Séré, Jean-Louis Lanet, and Julien Iguchi-Cartigny. Evaluation of Countermeasures Against Fault Attacks on Smart Cards. *International Journal of Security and Its Applications*, 5(2):49–61, April 2011.

[Sma] Smart Secure Devices (SSD) Team – XLIM, Université de Limoges. The CAP File Manipulator (CAPMAP). `http://secinfo.msi.unilim.fr/`.

[SMF97] J.R. Samson, W. Moreno, and F. Falquez. Validating Fault Tolerant Designs using Laser Fault Injection (LFI). In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 175–183, 1997.

[Sor87] Soref, R.A. and Bennett, B.R. Electrooptical Effects in Silicon. *IEEE Journal of Quantum Electron*, 23:123–129, 1987.

[SP06] Kai Schramm and Christof Paar. High Order Masking of the AES. In David Pointcheval, editor, *CT-RSA'06*, volume 3860 of *LNCS*, pages 208–225. Springer-Verlag, 2006.

[ST04] Adi Shamir and Eran Tromer. Acoustic cryptanalysis: On nosy people and noisy machines. EuroCrypt 2004 Rump Session, 2004. Available at `http://tau.ac.il/~tromer/acoustic/`.

[Sun97] Sun Microsystems Inc. Java Card Platform Specification Version 2.0, November 1997.

[Sun99a] Sun Microsystems Inc. Java Card™2.1 Application Programming Interface Specification, March 1999.

[Sun99b] Sun Microsystems Inc. Java Card™2.1 Runtime Environment Specification, March 1999.

[Sun99c] Sun Microsystems Inc. Java Card™2.1 Virtual Machine Specification, March 1999.

[Sun00a] Sun Microsystems Inc. Java Card™2.1.1 Application Programming Interface Specification, May 2000.

[Sun00b] Sun Microsystems Inc. Java Card™2.1.1 Runtime Environment Specification, May 2000.

[Sun00c] Sun Microsystems Inc. Java Card™2.1.1 Virtual Machine Specification, May 2000.

[Sun02a] Sun Microsystems Inc. Java Card™2.2 Application Programming Interface Specification, June 2002.

[Sun02b] Sun Microsystems Inc. Java Card™2.2 Runtime Environment Specification, June 2002.

[Sun02c] Sun Microsystems Inc. Java Card™2.2 Virtual Machine Specification, June 2002.

[Sun03a] Sun Microsystems Inc. Application Programming Interface Specification, Java Card™Platform, Version 2.2.1, October 2003.

[Sun03b] Sun Microsystems Inc. Runtime Environment Specification, Java Card™Platform, Version 2.2.1, October 2003.

[Sun03c] Sun Microsystems Inc. Virtual Machine Specification, Java Card™Platform, Version 2.2.1, October 2003.

[Sun06a] Sun Microsystems Inc. Application Programming Interface Specification, Java Card™Platform, Version 2.2.2, March 2006.

[Sun06b] Sun Microsystems Inc. Runtime Environment Specification, Java Card™Platform, Version 2.2.2, March 2006.

[Sun06c] Sun Microsystems Inc. Virtual Machine Specification, Java Card™Platform, Version 2.2.2, March 2006.

[Sun09a] Sun Microsystems Inc. Application Programming Interface Specification, Java Card™Platform, Version 3.0.1, Classic Edition, May 2009.

[Sun09b] Sun Microsystems Inc. Application Programming Interface Specification, Java Card™Platform, Version 3.0.1, Connected Edition, May 2009.

[Sun09c] Sun Microsystems Inc. Java™Servlet Specification, Java Card™Platform, Version 3.0.1, Connected Edition, May 2009.

[Sun09d] Sun Microsystems Inc. Runtime Environment Specification, Java Card™Platform, Version 3.0.1, Classic Edition, May 2009.

[Sun09e] Sun Microsystems Inc. Runtime Environment Specification, Java Card™Platform, Version 3.0.1, Connected Edition, May 2009.

[Sun09f] Sun Microsystems Inc. Virtual Machine Specification, Java Card Platfokrm Version 3.0.1 Connected Edition. May 2009.

[Sun09g] Sun Microsystems Inc. Virtual Machine Specification, Java Card™Platform, Version 3.0.1, Classic Edition, May 2009.

[Sun09h] Sun Microsystems Inc. Virtual Machine Specification, Java Card™Platform, Version 3.0.1, Connected Edition, May 2009.

[The81] The Information Sciences Institute of the University of South California. *RFC 791: Internet Protocol, DARPA Internet Program Protocol Specification*. The Internet Engineering Task Force (IETF), 1981.

[The11a] The European Telecommunications Standards Institute. *TS 102223, Card Application Toolkit, v10.5.0*. Sep 2011.

[The11b] The European Telecommunications Standards Institute. *TS 102241, UICC Application Programming Interface for Java Card, v9.1.0*. May 2011.

[The11c] The European Telecommunications Standards Institute. *TS 131111, Universal Subscriber Identity Module (USIM) Application Toolkit (USAT)*. Nov 2011.

[The11d] The European Telecommunications Standards Institute. *TS 131130, (U)SIM Application Programming Interface for Java Card, v10.1.0*. Jul 2011.

[The11e] The European Telecommunications Standards Institute. *TS 151014*. Sep 2011.

[The12a] The Open Web Application Security Project (OWASP). Information Leakage, 2012.

[The12b] The Open Web Application Security Project (OWASP). Uncaught Exceptions, 2012.

[TN11] Christopher Tarnovsky and Karsten Nohl. Reviving Smart Card Analysis. Chaos Communication Camp, 2011.

[Tru06] Trusted Logic S.A. Java Card™Protection Profile Collection, Version 1.1, May 2006.

[Ugo77] Michel Ugon. Support d'information portatif muni d'un microprocesseur et d'une mémoire morte programmable. French patent FR2401459, August 1977.

[USB05] USB Implementers Forum, Inc. Universal Serial Bus Communication Class Subclass Specification for Ethernet Emulation Model Devices, Rev. 1.0, February 2005.

[Ver06a] Dennis Vermoen. *Reverse Engineering of Java Card Applets Using Power Analysis*. Msc thesis, Faculty of Electrical Engineering, Mathematics and Computer Science – Delft University of Technology, May 2006.

[Ver06b] Olli Vertanen. Java Type Confusion and Fault Attacks. In L. Breveglieri, I. Koren, D. Naccache, and J.-P. Seifert, editors, *Proceedings of the Workshop on Fault Diagnosis and Tolerance in Cryptography – FDTC'06*, volume 4236 of *LNCS*, pages 237–251. Springer, 2006.

[Ver11] Ingrid Verbauwhede. The fault attack jungle - yet another concern for the designer (invited). IEEE International Workshop on Fault Diagnosis and Tolerance in Cryptography, 2011.

[VF10] Eric Vétillard and Anthony Ferrari. Combined Attacks and Coutermeasures. In *Smart Card Research and Advanced Application Conference – CARDIS 2010*, volume 6035 of *LNCS*, pages 133–147. Springer Verlag, 2010.

[VWG07] Dennis Vermoen, Marc Witteman, and Georgi N. Gaydadjiev. Reverse engineering java card applet using power analysis. In *Proceedings of the* 1*st Workshop on Information Security Theory and Practice (WISTP'07)*, volume 4462 of *LNCS*, pages 138–149. Springer, 2007.

[Wit03] Marc Witteman. Java Card Security. *Information Security Bulletin*, 8:291–298, October 2003.

[Wri87] Peter Wright. *Spy Catcher: The Candid Autobiography of a Senior Intelligence Officer*. William Heinemann Australia, 1987.

[YJ00] Sung-Ming Yen and Marc Joye. Checking before Output may not be enough against Fault-based Cryptanalysis. volume 49 (9), pages 967–970, 2000.