JaCarTA : un framework d'exploit Java Card

Julien Lancia (j.lancia@serma.com) *

Abstract: Dans le domaine des cartes à puce, et plus précisément en ce qui concerne les plateformes Java Card, de nouvelles attaques logiques voient régulièrement le jour menant à des *exploits* semblables à ceux observés dans les autres domaines de la sécurité. Afin de rationaliser la conception, la mise en œuvre et la capitalisation des exploits sur les plateformes Java Card, nous présentons JaCarTA un framework d'exploit en Python dédié aux technologies Java Card.

Keywords: Java Card, Framework, Attaque, Simulation, Exploit

Introduction

Dans les premiers temps, l'exploitation de vulnérabilités informatiques nécessitait pour chaque nouvelle faille de sécurité identifiée le développement d'outils dédiés tant pour la conception que pour la mise en œuvre des attaques. Avec l'avènement des framework d'exploit [land, lanc, lanf, lana, Sko], environnements de programmation génériques pour la conception d'attaques, cette tâche fastidieuse et répétitive a été largement réduite, et le domaine de l'exploitation de vulnérabilités informatiques est passé de l'ère artisanale à l'ère industrielle. Les frameworks d'exploit offrent des abstractions de haut niveau, facilitant la conception et la mise en œuvre des attaques contre les systèmes informatiques. Ils permettent également de capitaliser les attaques existantes afin de les reproduire de manière efficace. L'utilisation des framework d'exploit est profitable en premier lieu pour les attaquants, qui peuvent se concentrer sur le concept des attaques et non plus sur les moyens techniques et gagnent ainsi en efficacité. Ils offrent également une opportunité pour les développeurs de solutions sécurisées qui peuvent mettre à l'épreuve leurs mécanismes de sécurité face à des attaques à l'état de l'art, et ce à moindre coût.

Dans le domaine des cartes à puce, les plateformes Java Card [Sun02c, Sun02b, Sun02a] sont particulièrement exposées aux attaques logiques [HM03, MP08, ICL10, Lan12a, MP08] et subissent donc les mêmes contraintes de sécurité que les autres domaines des technologies de l'information. L'évolution constante des moyens d'attaques contre ces plateformes logicielles embarquées rend donc difficile la réalisation de contre-mesures de sécurité efficaces.

L'évaluation du niveau de sécurité d'une plateforme nécessite le développement d'applets malveillantes permettant d'évaluer la robustesse des mécanismes de sécurité implémentés. Ce type d'activité nécessite d'adapter constamment les applets développées aux spécificités de la plateforme, d'intégrer des mécanismes d'attaques élémentaires existants dans de nouvelles applets, et d'implémenter de nouveaux chemins d'attaque. Il n'existe pas à notre connaissance de suite logicielle permettant de réaliser simplement ces tâches. De

*. SERMA Technologies, CESTI, 30, avenue Gustave Eiffel,
33600 Pessac, France

ce fait, l'évaluation du niveau de sécurité d'une plateforme donnée vis-à-vis des attaques à l'état de l'art est une tâche complexe pour les différents acteurs du milieu.

Afin de palier ces problèmes, nous avons développé JaCarTA (Java Card Tools for Attackers) un framework d'exploit pour plateformes Java Card. Ce framework facilite la conception des attaques grâce à une API (Application Programming Interfaces) haut niveau, la validation des chemins d'attaque par simulation, la mise en œuvre sur les cartes à puce grâce à une chaîne de compilation intégrée et la capitalisation des attaques connues afin d'évaluer simplement le niveau de sécurité d'une plateforme au regard des attaques existantes. Développé entièrement en Python, le framework JaCarTA profite de l'expressivité de ce langage interprété afin de fournir des abstractions de haut niveau pour le développement et l'exploitation des attaques, et permet d'ores et déjà de réaliser la majeure partie des attaques publiées sur les plateformes Java Card.

Le plan de cet article s'articule autour des fonctionnalités offertes par le framework. Dans un premier temps, nous présentons les API du framework qui facilitent la conception des attaques. Nous présentons ensuite les classes Java Card fournies par le framework et la chaîne de compilation permettant d'automatiser le rejeu des attaques sur différentes plateformes. Finalement, nous présentons le simulateur intégré dans le framework qui permet de valider le comportement des plateformes faces aux attaques logiques et par fautes (attaques combinées).

1 Conception des attaques

Les attaques logiques sur les plateformes Java Card utilisent trois techniques distinctes permettant de perturber le comportement nominal de l'environnement d'exécution. La première technique consiste à modifier le bytecode issu de la compilation des fichiers Java afin que l'exécution du programme provoque une faille de sécurité. La seconde technique consiste à modifier la structure de l'archive Java Card (fichier .cap, équivalent d'un fichier .jar ou .war sur les plateformes Java) en dehors du code des méthodes afin de leurrer l'environnement d'exécution. Finalement, la dernière technique dite par "attaque combinée" s'appuie sur une des deux techniques précédente et la combine avec une injection de faute physique sur la carte (par laser ou impulsion électromagnétique).

Nous présentons dans la partie 1.1 l'API Python qui permet la modification du bycode, puis nous détaillons dans la partie 1.2 l'API Python permettant la réalisation d'attaques sur l'archive cap. Finalement, nous présentons dans la partie 1.3 l'utilisation du framework pour faciliter la réalisation des attaques combinées.

1.1 Attaques sur le bytecode

La compilation d'une applet Java Card produit un fichier *.class* constitué d'une suite de bytecodes, équivalent de l'assembleur pour la machine virtuelle Java Card. Les attaques logiques reposant sur une modification du bytecode sont les attaques les plus répandues dans la littérature et ont montré leur efficacité pour mettre en défaut la sécurité des machines virtuelles Java Card. Notre approche pour faciliter la conception de ce type d'attaques repose sur l'utilisation de méthodes Java standard jouant le rôle de marqueurs au niveau du code Java. L'API Python fournie dans le framework JaCarTA permet ensuite d'associer à ces méthodes "marqueurs" des traitements sur le bytecode. Ceci permet, grâce à ces API, d'intégrer simplement dans les applets malveillantes les mécanismes élémen-

taires d'attaques au niveau bytecode (tels que le transtypage) tout en restant au niveau d'abstraction du langage Java.

Afin d'illustrer l'utilisation de cette API, nous présentons la réalisation d'une attaque par confusion de type à l'aide du framework JaCarTA. La confusion de type consiste à forger une référence d'un type donné vers un objet de type différent, contournant ainsi le système de typage de la machine virtuelle. Nous réalisons cette confusion de type en exploitant le fait que la pile Java Card n'est pas typée ; il est donc possible d'empiler un objet d'un type donné, et de le dépiler dans un objet d'un type différent. Considérons l'extrait de code présenté dans le listing 1. L'objectif de ce code est de produire un transtypage illégal entre la référence *objetA* de type *ObjetTypeA* et la référence *objetB* de type *ObjetTypeB*. Cette attaque repose sur la méthode *refToRefMarker*, qui possède deux arguments (src et dst) et un corps vide. En effet, cette méthode sert uniquement, au travers de ses deux argument, à indiquer au parseur de bytecode intégré dans le framework JaCarTA les référence source et destination de l'opération de transtypage. En l'occurrence, la ligne 8 du listing précise que l'on veut transtyper la référence *objetA* en *objetB*. Le champ *this.fakeB* de type *ObjetTypeB* servira a sauvegarder de manière permanente l'objet issu de ce transtypage illégal.

```
1  // Declaration du marqueur
2  public static void refToRefMarker(Object src, Object  dst){}
3  ...
4  // objetA et objetB sont les objets cible de la confusion
5  ObjetTypeA objetA = new ObjetTypeA();
6  ObjetTypeB objetB = new ObjetTypeB();
7  // Marqueur pour le framework JaCarTA
8  refToRefMarker(objetA, objetB);
9  // Sauvegarde de la confusion
10 this.fakeB = objetB;
```

Listing 1: Code Java d'une attaque par confusion de type.

La compilation du code Java présenté dans le listing 1 produit le bytecode présenté dans le listing 2. On peut voir que les deux arguments de la méthode reftoRefMarker (*objetA* et *objetB*) sont chargés en ligne 9 et 10 à partir des variables locales 1 et 2. Ce sont ces deux arguments passés à la méthode marqueur *reftoRefMarker* qui permettront, via l'API du framework, d'extraire au niveau bytecode les index de variables locales à utiliser pour réaliser la confusion de type.

```
1  new 34;            // new ObjetTypeA
2  dup;
3  invokespecial 35;
4  astore_1;          // var1 : objetA
5  new 37;            // new ObjetTypeB
6  dup;
7  invokespecial 40;  //
8  astore_2;          // var2 : objetB
9  aload_1;
10 aload_2;
11 invokestatic 109;  // refToRefMarker(objetA,objetB)
12 aload_0;
13 aload_2;
14 putfield_a 22;     // this.fakeB = objetB
15 return;
```

Listing 2: Bytecode résultant de la compilation du code Java.

Finalement, le listing 3 présente le script Python qui permet de réaliser la confusion de type grâce à l'API de manipulation de bytecode du framework JaCarTA. Les lignes 4 à 6 montrent comment utiliser le parseur de bytecode du framework pour extraire les bytecodes qui réalisent le chargement des arguments de la méthode *refToRefMarker*. Ces trois lignes permettent au parseur d'identifier les lignes 9 à 11 du listing 2 qui correspondent à l'appel de la méthode *refToRefMarker*. Les opérandes des bytecodes ainsi extraits servent en ligne 8 à produire un nouveau code, qui charge la référence de *objetA* à l'aide du bytecode aload et la sauve dans la référence *objetB* à l'aide du bytecode astore. Ce script, une fois développé et intégré dans le framework, peut être réutilisé sur toute applet Java Card afin de réaliser une confusion de type via une simple invocation Java de la méthode marqueur *refToRefMarker*, sans connaissance du bytecode sous-jacent.

L'exécution du script python présenté dans le listing 3 sur le bytecode présenté dans le listing 2 produit le bytecode réalisant un exploit par confusion de type présenté dans le listing 4.

```
1  parser = JCAParser()
2  parser.parse_file(filename)
3  # Bytecode à remplacer
4  aload1 = parser.add_bytecode('aload', nb_operand=1)
5  aload2 = parser.add_bytecode('aload', nb_operand=1)
6  parser.add_static_method('refToRefMarker')
7  # Nouveau bytecode
8  new_code = [parser.aload(aload1.get_operand()),
9              parser.astore(aload2.get_operand())]
10 # Traitement
11 nb_replacement = parser.replace_all(new_code)
12 parser.write()
```

Listing 3: Script JaCarTA qui réalise la confusion de type.

```
1  ...                    |
2  aload_1;               | aload_1;
3  aload_2;               | astore_2;    // confusion de type
4  invokestatic 109;      |
5  aload_0;               | aload_0;
6  aload_2;               | aload_2;
7  putfield_a 22;         | putfield_a 22;
8  return;                | return;
```

Listing 4: Bytecode modifié (à droite) résultant de l'exécution du script Python sur le bytecode original (à gauche).

Nous avons présenté au travers d'un exemple simple quelques fonctionnalités de l'API de manipulation de bytecode offertes par le framework JaCarTA. Cette API permet d'effectuer des manipulations plus complexes, qui ne sont pas détaillées ici par souci de concision.

Nous allons maintenant nous intéresser aux fonctionnalités de manipulation des archives *cap* fournies par le framework d'exploit.

1.2 Attaques sur le fichier CAP

Avant de pouvoir être chargées sur une carte à puce embarquant une plateforme Java Card, les classes Java qui constituent un package doivent être converties à l'aide du convertisseur fourni dans le kit de développement Java Card. La conversion produit une archive

au format *cap*, qui contient entre autre le bytecode des méthodes. Outre le code des méthodes, l'archive *cap* contient de nombreuses informations telles que les types déclarés dans le package, les identifiants de classes et de méthodes, ainsi que les informations nécessaires à l'éditeur de lien de la plateforme Java Card lors du chargement de l'applet sur la plateforme. Un certain nombre d'attaques logiques [ICL10, BICL11a] exploitent la possibilité de modifier les informations contenues dans l'archive *cap* avant le chargement sur la carte afin d'exploiter des failles de sécurité de la machine virtuelle.

La structure logique de l'archive *cap* est relativement complexe. Tout d'abord, elle est composée d'informations interdépendantes telles que des valeurs représentant des offsets dans des structures internes ou des indexes de tableaux. De plus, la valeur de certains champs (ou *flags*) conditionnent l'interprétation du reste de la structure. On conçoit donc, au vu de la complexité du format de l'archive *cap*, que la modification manuelle de cette archive est particulièrement délicate, car la simple modification d'un octet peut avoir des implications sur toute l'archive et rendre celle-ci totalement incohérente. Pour simplifier la mise en œuvre des attaques sur l'archive *cap*, le framework JaCarTA intègre un parseur qui offre une abstraction purement objet de l'archive cap. Ce parseur maintient en permanence une structure cohérente de l'archive et permet donc de modifier toutes les valeurs internes en conservant une structure valide.

Pour illustrer l'utilisation de ce parseur, nous nous appuyons sur l'attaque EMAN [IC09] qui consiste à leurrer l'éditeur de liens lors du chargement d'une applet en modifiant la structure de l'archive *cap*. Cette attaque nécessite de supprimer une entrée dans un tableau de l'archive contenant des offsets dans le code des méthodes. Cette modification d'apparence anodine a de nombreuses répercutions, car elle nécessite de modifier la taille du tableau, la taille du composant qui contient le tableau, tous les offsets pointant vers des régions subséquentes à la modification, etc.

Le listing 5 présente le code Python permettant de supprimer une entrée du tableau d'offsets du linker grâce à l'API du parseur d'archive *cap*. Après l'instanciation du parseur, on peut observer en ligne 4 la récupération de l'élément racine. La structure de l'archive *cap* étant hiérarchique, c'est cet élément qui contient toutes les structures sous-jacentes. La ligne 7 réalise l'accès à l'élément recherché (le tableau d'offsets dans le composant *reference location*) dans un paradigme objet (la structure hiérarchique de l'archive est reproduite sous forme d'instances imbriquées). Finalement, en lignes 10 et 12, l'octet est supprimé du tableau et l'archive est reconstituée. La complexité des interdépendances de l'archive est totalement masquée à l'utilisateur qui peut se concentrer sur la réalisation de l'attaque.

```
1  capparser = CAPParser()
2  capparser.parse(capfile)
3  capparser.check()
4  # Recuperation de l'element racine
5  root = capparser.parser.get_root()
6  # Acces au tableau d'offsets du linker
7  offset_array = root.ref_loc_comp.offsets_byte2
8  # Suppression de la derniere entree
9  toRemove = offset_array[-1]
10 offset_array.remove_element(toRemove)
11 # Traitement
12 capparser.render_cap()
```

Listing 5: Script JaCarTA qui supprime une entrée du tableau de références du linker l'archive cap.

Comme pour les scripts qui manipulent le bytecode, ce script peut être intégré au framework et réutilisé pour mener l'attaque sur toute archive *cap*. Ainsi, la réalisation d'exploit par manipulation d'archive *cap* peut se faire par simple exécution d'un script Python.

Après avoir étudié les apports du framework JaCarTA pour la réalisation d'attaques sur le bytecode et sur l'archive cap, nous nous focalisons à présent sur les attaques combinées.

1.3 Attaques combinées et injection de faute

Apparu récemment, un nouveau paradigme d'attaque dites "combinées" [BTG10, VF10, BICL11b] permet de concevoir des applets en apparence légales, et dont le code malveillant est activé une fois chargé sur la carte à l'aide d'une injection de faute matérielle (impulsion laser ou électromagnétique sur le composant) [BECN$^+$04, SAKP03, GT04]. Nous avons montré préalablement comment le framework JaCarTA simplifiait la réalisation de code malveillant grâce aux API python. Nous allons maintenant détailler les apports de ce framework en ce qui concerne les injections de fautes matérielles.

Les attaques physiques sur les cartes à puce sont soumises à de fortes contraintes de localisation et temporelles. En effet, le mécanisme ciblé est réalisé dans une zone physique déterminée de la puce, et se produit à un instant précis de l'exécution du code. Afin de réussir une attaque combinée, il est donc nécessaire d'identifier précisément la zone et l'instant de l'attaque. L'identification de la zone temporelle peut être réalisée grâce à l'analyse des canaux de fuites [MDS02, AARR03, KJJ99] (consommation en courant, émanations électromagnétiques) résultant de l'exécution du code à attaquer sur la carte à puce. Toutefois, les contre-mesures physiques embarquées sur la carte et la nature du code à attaquer peuvent complexifier l'analyse de ces canaux de fuite.

Certaines attaques combinées [BICL11a] ciblent un bytecode précis, ce qui réduit considérablement la fenêtre temporelle d'attaque. Afin de localiser précisément la zone temporelle d'exécution du bytecode ciblé, une approche consiste à placer, autour du bytecode à attaquer, du code provoquant des fuites facilement identifiables. Or, la compilation d'une instruction élémentaire en Java produit généralement un ensemble de bytecodes, ce qui interdit l'ajout de motifs de code avec une précision au niveau du bytecode. Il est donc nécessaire de manipuler le code de l'applet au niveau du bytecode. Grâce aux techniques de méthodes marqueur présentées dans la partie 1.1, il est possible d'ajouter des marqueurs au niveau Java qui, une fois traitées par le framework JaCarTA, ajouteront des motifs de code déterminés autour du bytecode à attaquer et faciliteront ainsi l'identification de la zone temporelle de l'attaque.

Le listing 6 présente le code Java servant de base à une attaque combinée sur le bytecode goto. La compilation de ce code produit le bytecode présentés dans le listing 7. Le but de l'attaque est d'injecter une faute précisément sur l'opérande du bytecode goto en ligne 9 afin de dérouter le flot d'exécution vers une autre partie du code. Il est donc nécessaire de localiser précisément le bytecode goto pour augmenter les chances de succès de l'attaque.

```
1 for(short i=0 ; i<n ;(short)i++){
2 foo=(byte)0xBA;
3 }
```

Listing 6: Code Java de l'attaque sur le bytecode goto.

```
    sconst_0;
    sstore_1;
L0: sload_1;
    sload_2;
    if_scmpge L1;
    bspush BA;
    putfield_b_this 0;
    sinc 1 1;
    goto L0;
L1: ...
```

Listing 7: Bytecode correspondant à l'attaque sur le bytecode goto.

Pour insérer des motifs de code autour du bytecode goto et ainsi augmenter les fuites autour de ce bytecode, nous ajoutons un appel à la méthode marqueur *bytecodeMarker* tel que présenté dans le listing 8. Cette méthode marqueur sera exploitée par le framework d'exploit pour déterminer le bytecode cible de l'attaque et ajouter le motif de bytecode. L'argument de la méthode (0x70) correspond à la valeur du bytecode goto.

```
bytecodeMarker(0x70);
for(short i=0 ; i<n ;(short)i++){
foo=(byte)0xBA;
}
```

Listing 8: Code Java de l'attaque sur le bytecode goto avec méthode marqueur.

Le listing 9 présente le script Python qui effectue l'ajout des motifs de code (20 bytecodes *nop* en l'occurrence) autour du bytecode goto grâce à l'API de manipulation de bytecode du framework JaCarTA. Comme précédemment, les lignes 1 à 5 initialisent le parseur et extraient la méthode marqueur *bytecodeMarker* et son argument. Les lignes 7 et 8 convertissent l'argument hexadécimal de la méthode en identifiant de bytecode. En ligne 11, l'appel à la méthode *get_next_bytecode* de l'API du framework JaCarTA permet de rechercher la première occurrence d'un bytecode successive à la méthode marqueur. Cette fonction retourne le bytecode trouvé (variable *bytecode*) et tous les bytecodes compris entre le marqueur et le bytecode trouvé (variable *other_bytecode*). Finalement, les lignes 13 à 19 reconstruisent le code en ajoutant les motifs (20 bytecodes *nop*).

```
parser = JCAParser()
parser.parse_file(filename)
# le premier argument identifie le bytecode à trouver
bc_arg = parser.add_bytecode('bspush', nb_operand=1)
parser.add_static_method('bytecodeMarker')
# Convertit l'argument en bytecode
bc_int = parser.get_operand(bc_arg, 0)
bc_val = parser.to_bytecode(bc_int)
# Recupere le premier bytecode recherche
bytecode = parser.add_next_bytecode(bc_val)
(other_code, bytecode) = parser.get_next_bytecode(bytecode)
# Motif à ajouter
new_code = other_code +
           [parser.nop()]*20 +
           [bytecode] +
           [parser.nop()]*20
# Traitement
nb_replacement = parser.replace_all(new_code)
parser.write()
```

Listing 9: Script JaCarTA qui réalise l'ajout des motifs de code.

Ce script Python permet, en utilisant directement des méthodes marqueur au niveau Java, d'ajouter des motifs de code avec une précision au niveau bytecode. En permettant ainsi l'ajout de motifs indentifiables via les canaux de fuite, le framework jaCarTA facilite les attaques par fautes et les attaques combinées nécessitant une identification précise des zones temporelles.

Nous avons présenté dans les sections précédentes les API de manipulation de bytecode et d'archives *cap* offertes par le framework JaCarTA. Ces API permettent d'écrire simplement des scripts Python qui produisent des applets malveillantes pouvant mettre en défaut la sécurité des plateformes Java Card. Nous allons maintenant détailler les fonctionnalités du framework permettant d'automatiser la mise en œuvre de ces attaques, afin de tester facilement le niveau de sécurité des plateformes Java Card.

2 Automatiser le rejeu des attaques

Comme vu dans les sections précédentes, le framework JaCarTA offre des abstractions de haut niveau permettant de manipuler le bytecode des applets au travers de scripts Python. Les attaquants peuvent ainsi facilement développer des applets Java Card malveillantes pour évaluer la résistance des plateformes embarquées.

Dans un deuxième temps, une fois les scripts de manipulation de bytecode développés, le framework JaCarTA offre une chaîne d'outillage permettant d'appliquer simplement les scripts python aux applets Java Card, de charger les applets malveillantes ainsi produites sur une carte à puce et de valider le comportement de la plateforme Java Card ciblée face au code hostile.

Cette chaîne d'outillage s'appuie sur deux éléments : une applet d'attaque générique servant de base de développement pour les applets, et une chaîne de compilation Java Card intégrant la gestion des scripts Python de mutation d'applets.

2.1 Applet d'attaque générique

Le framework d'exploit JaCarTA propose une approche modulaire au développement d'attaques Java Card. Cette approche repose sur une classe abstraite *Module*, qui sert de base à l'implémentation des attaques élémentaires, et la classe abstraite *ModularApplet* qui fournit un ensemble de services génériques aux modules d'attaques et permet de grouper les différents modules dans une applet.

L'utilisation des applets modulaires permet d'agréger les fonctionnalités d'attaques par similitudes dans des applets d'attaques génériques et évolutives. L'applet modulaire se comporte comme un *dispatcher* de haut niveau lorsque le terminal envoie une instruction APDU à la carte, qui active le code hostile du module correspondant.

La classe *ModularApplet* déclare également un certain nombre de méthodes marqueur génériques pouvant être utilisées par tous les modules. Les méthodes marqueur spécifiques à une attaque donnée sont quant-à-elles déclarées dans les modules d'attaques.

Lorsque les applets modulaires et les modules d'attaques sont prêts à être déployés, le framework d'exploit JaCarTA fournit une chaîne de compilation qui facilite la mise en œuvre des attaques sur les cartes à puce.

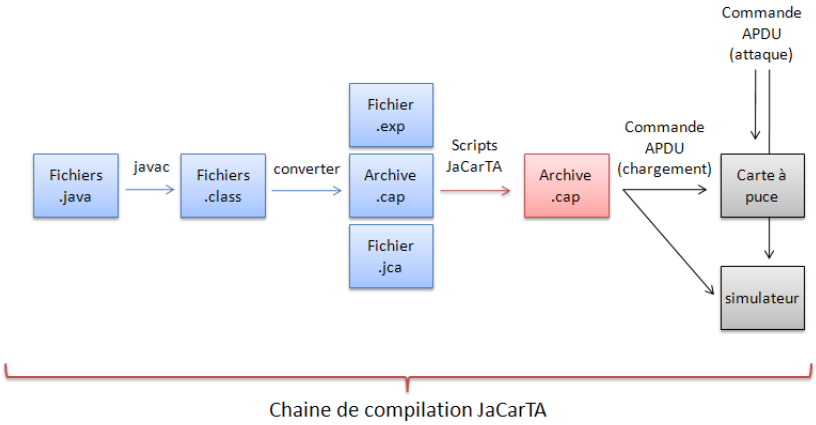

Figure 1: Chaîne de compilation JaCarTA.

2.2 chaîne de compilation

La mise en œuvre d'une attaque Java Card sur une plateforme embarquée se compose de nombreuses étapes dont certaines nécessitent plusieurs interventions manuelles, rendant cette opération laborieuse et peu automatisable. La figure 1 présente une chaîne de compilation complète où les zones bleues représentent les éléments d'une chaîne de compilation classique, et les zones rouges les éléments propres au framework JaCarTA. Les zones noires représentent quant-à-elles l'outillage de communication avec les cartes à puce et de simulation.

Une chaîne de compilation classique pour le chargement d'une applet Java Card se compose des opérations suivantes :

- Compilation : La compilation d'un fichier java en fichier *class* est réalisée à l'aide de l'exécutable *javac* fourni dans le kit de développement Java.
- Conversion : La conversion d'un package java en archive *cap* est réalisée à l'aide de l'exécutable *convertisseur* fourni dans le kit de développement Java Card.
- Chargement : Le chargement d'une applet consiste à envoyer à la carte à puce une suite de commandes APDU suivant la norme GlobalPlatform [Glo06]. L'envoi d'APDU à une carte à puce au travers d'un terminal peut se faire à l'aide de différents outils et API [lanb, lang, lane].
- Exécution : L'exécution d'une applet consiste à envoyer à la carte à puce une suite de commandes APDU suivant la norme ISO7816 [INT01].

Le framework JaCarTA utilise une approche générative afin de simplifier la mise en œuvre de cette chaîne de compilation. À partir d'un package Java contenant un ensemble de fichiers *.java*, le framework JaCarTA génère un fichier *Makefile* qui automatise entièrement la chaîne de compilation, incluant la mise en œuvre des scripts Python permettant la création des applets hostiles (présentés en section 1), mais aussi le chargement et l'exécution des applets. Ce générateur est largement paramétrable (version de Java Card, de GlobalPlatform, paramètres d'installation, ...) au travers d'options en ligne de commande afin d'être compatible avec la majeure partie des cartes à puce du commerce.

Concernant la communication avec la carte à puce, le framework offre le choix entre

l'envoi d'APDU basé sur notre bibliothèque Python propriétaire de communication avec les cartes à puces (*Triton*) pour le chargement et l'exécution, ou l'utilisation du programme *gpshell* [lanb] pour le chargement des applets et du programme *apdutool* inclus dans le kit de développement Java Card pour l'envoi de commandes aux applets. Le framework génère également les classes Python permettant d'envoyer des commandes aux différents modules enregistrés dans les applets modulaires, et présente ces différentes fonctionnalités sous forme d'un menu interactif facilitant l'activation des fonctionnalités d'attaques présentes dans les modules.

Outre la mise en œuvre concrète sur des plateformes Java Card embarquées sur cartes à puce, la framework JaCarTA permet également de tester les attaques logiques et combinées sur une plateforme Java Card simulée.

3 Validation par simulation

La conception d'attaques logiques et combinées sur les plateformes Java Card est souvent complexe et sujette à erreurs. Le risque est considérable car les plateformes modernes incluent de nombreuses contre-mesures, et une erreur dans la conception d'une attaque peut provoquer la destruction irréversible d'une carte à puce. Afin de diminuer ces risques, le framework d'exploit JaCarTA inclut un simulateur de plateforme Java Card qui permet de tester ses attaques sur un exemple d'implémentation de machine virtuelle JavaCard sans contre-mesure, sur une architecture logicielle classique (système d'exploitation Microsoft ou Linux).

3.1 Simulation des attaques logiques

La simulation des attaques logiques s'appuie sur l'implémentation de référence de machine virtuelle Java Card *cref* fournie dans le kit de développement. L'exploitation des fonctionnalités de cette implémentation de référence *cref* est complexe et laborieuse, c'est pourquoi nous avons développé une API Python haut niveau permettant de scripter les interactions avec *cref*. De plus, nous avons intégré le simulateur dans la chaîne de compilation générée, ce qui permet de tester les applets malveillantes de manière automatique avant de les charger sur une carte physique. L'intégration du simulateur dans la chaîne de compilation complète est présentée dans la figure 1.

3.2 Simulation des attaques par fautes

Dans le cadre des attaques combinées, il est également important de pouvoir valider le comportement des modules d'attaques en présence d'injection de fautes physiques. Afin de permettre cette validation préalablement à la réalisation des campagnes d'injections de fautes, le simulateur intégré dans le framework JaCarTA possède deux plugins qui simulent des injections de fautes physiques. Le premier plugin permet de simuler des injections de fautes dans la mémoire persistante (EEPROM), le deuxième plugin permet de simuler des injections de fautes dans la pile d'exécution en mémoire volatile (RAM).

Les deux plugins de simulation d'injection de fautes sont développés en Python et accessibles via une API du framework. Cette API peut être utilisée pour simuler des campagnes d'injection de fautes complètes, et permet également d'étendre les modèles de fautes offerts par le simulateur. Suite à l'exécution d'une campagne d'injections de fautes dans la mémoire permanente ou volatile, le simulateur peut représenter une cartographie

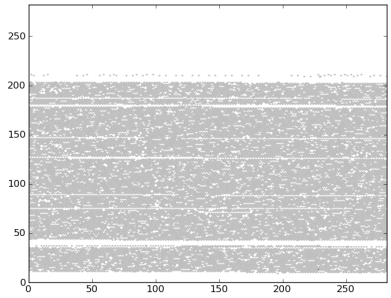

Figure 2: Cartographie des résultats d'une campagne d'injection de fautes sur la mémoire non volatile produite par le simulateur.

de la mémoire où apparaissent les injections ayant mené à une attaque réussie et celles ayant raté. La figure 2 représente la corrélation des attaques réussies (en gris) avec la position physique (x,y) dans la mémoire de l'injection simulée, pour une campagne de test composée de 64000 injections [Lan12b].

Grâce au simulateur et aux plugins d'injections de fautes, les attaques peuvent être validées avant d'être mise en œuvre sur une plateforme Java Card embarquée sur carte à puce. La simulation des attaques logiques et physique est intégrée dans la chaîne de compilation, facilitant leur utilisation, et la communication avec le simulateur est transparente si les classes Python générées par le framework sont utilisées pour exécuter les applets.

4 travaux connexes

Comme présenté précédemment, il n'existe actuellement pas de framework d'exploit complet permettant de mettre en œuvre les attaques connues sur les plateformes Java Card. Toutefois, certains éléments inclus dans le framework JaCarTA existent par ailleurs.

L'équipe SSD du laboratoire Xlim offre en téléchargement un parseur d'archive *cap* [SSDSTX] qui, comme le parseur intégré dans JaCarTA, maintient la cohérence de l'archive lors des modifications. Ce parseur est programmé en Java ce qui complique son intégration dans un framework en Python. Cette même équipe dispose également d'un simulateur de faute propriétaire sur plateforme Java Card [MMLC11], que nous n'avons pas pu évaluer du fait qu'il n'est pas disponible librement.

Le framework JaCarTA intègre un parseur de bytecode permettant la modification du code java compilé dans l'archive *cap*. Il existe en Python différents module permettant de parser du bytecode [Jyt08, Pau, Vic], mais ces différents modules n'offrent pas de solution simple pour modifier les fichiers et les régénérer après modification. Outre les modules Python, certaines librairies Java permettent de manipuler les fichiers *class* issus de la compilation des fichiers Java [BLC02, Dah02]. Ces librairies ne permettent toutefois pas d'implémenter notre solution basée sur les méthodes marqueur offrant un accès à la manipulation de bytecode au niveau de l'applet JavaCard, et nécessiteraient l'implémentation d'une couche *wrapper* afin de s'interfacer avec notre framework en Python.

5 Conclusion

Le framework JaCarTA intègre une suite complète d'outils pour la conception, le test, la mise en œuvre et la capitalisation des attaques sur les plateformes Java Card. L'utilisation du langage Python dans ce framework permet de s'appuyer sur l'expressivité de ce langage pour faciliter la programmation des scripts d'attaques et offrir des API de haut niveau aux attaquants.

Nous avons exploité les fonctionnalités de ce framework afin de valider la résistance de plusieurs plateformes disponibles dans le commerce. La mise en œuvre des attaques via le framework JaCarTA a permis d'identifier de manière rapide et efficace plusieurs attaques logiques exploitables.

L'objectif de ce framework, qui est d'offrir un socle commun pour faciliter la mise en œuvre des attaques logiques et combinées sur les systèmes Java Card et ainsi simplifier l'évaluation de la sécurité des plateformes Java Card, est donc pleinement vérifié. L'utilisation de ce framework à plus grande échelle devrait permettre d'améliorer à moyen terme la sécurité des plateformes Java Card en identifiant en amont les failles sécuritaires présentes sur ces plateformes.

Références

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. The em side-channel(s). In Burton Kaliski, çetin Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer Berlin / Heidelberg, 2003.

[BECN+04] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. 2004.

[BICL11a] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In *Proceedings of the 10th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications*, CARDIS'11, pages 283–296, Berlin, Heidelberg, 2011. Springer-Verlag.

[BICL11b] Guillaume Bouffard, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Combined software and hardware attacks on the java card control flow. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications*, volume 7079 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin / Heidelberg, 2011.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM : A code manipulation tool to implement adaptable systems. In *Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France, November 2002.

[BTG10] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 148–163. Springer Berlin / Heidelberg, 2010.

[Dah02] Markus Dahm. Byte code engineering library (bcel). `http://jakarta.apache.org/bcel/manual.html`, 2002.

[Glo06] GlobalPlatform, Foster City, USA. *GlobalPlatform Card Specification*, version 2.2 edition, March 2006.

[GT04] Christophe Giraud and Hugues Thiebeauld. A survey on fault attacks. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas El Kalam, editors, *Smart Card Research and Advanced Applications VI*, volume 153 of *IFIP International Federation for Information Processing*, pages 159–176. Springer Boston, 2004.

[HM03] Jip Hogenboom and Wojciech Mostowski. Full memory read attack on a java card, 2003.

[IC09] Lanet J.L Iguchi-Cartigny, J. Évaluation de l'injection de code malicieux dans une java card. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication*, SSTIC, 2009.

[ICL10] Julien Iguchi-Cartigny and Jean-Louis Lanet. Developing a trojan applets in a smart card. *Journal in Computer Virology*, 6 :343–351, 2010.

[INT01] INTERNATIONAL STANDARD ORGANIZATION FOR STANDARDIZATION (ISO). *Identification cards – Integrated circuit(s) cards with contacts – Part III : Electronic signals and transmission protocols*, 2006-11-01.

[Jyt08] Jython Community. The Jython Tutorial. http ://www.jython.org/docs/tutorial/indexprogress.html, 2008.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[KQ07] Chong Hee Kim and Jean-Jacques Quisquater. Faults, injection methods, and fault attacks. *IEEE Des. Test*, 24 :544–545, November 2007.

[lana] CORE IMPACT. `http://www.coresecurity.com/content/core-impact-overview`.

[lanb] GpShell 1.4.4. `http://sourceforge.net/projects/globalplatform/`.

[lanc] Immunity CANVAS. `http://www.immunitysec.com/products-canvas.shtml`.

[land] Metasploit - Penetration Testing Resources. `http://www.metasploit.com/`.

[lane] MUSCLE. `http://www.linuxnet.com/`.

[lanf] OWASP Xenotix. `https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework`.

[lang] Winscard. `http://msdn.microsoft.com/en-us/library/ms924513.aspx`.

[Lan11] Julien Lancia. Un framework de fuzzing pour cartes à puce : application aux protocoles EMV. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication*, SSTIC, pages 350–368, 2011.

[Lan12a] Julien Lancia. Compromission d'une application bancaire JavaCard par attaque logicielle. In *Symposium sur la Sécurité des Technologies de l'Information et de la Communication*, SSTIC, 2012.

[Lan12b] Julien Lancia. Java Card combined attacks with localization-agnostic fault injection. In *Proceedings of the 11th Smart Card Research and Advanced Applications – CARDIS 2012*, Lecture Notes in Computer Science, 2012.

[MDS02] Thomas S. Messerges, Ezzat A. Dabbish, and Robert H. Sloan. Examining smart-card security under the threat of power analysis attacks. *IEEE Trans. Comput.*, 51 :541–552, May 2002.

[MMLC11] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny. Smartcm a smart card fault injection simulator. In *Information Forensics and Security (WIFS), 2011 IEEE International Workshop on*, pages 1 –6, 29 2011-dec. 2 2011.

[MP08] Wojciech Mostowski and Erik Poll. Malicious code on java card smartcards : Attacks and countermeasures. In Gilles Grimaud and François-Xavier Standaert, editors, *Smart Card Research and Advanced Applications*, volume 5189 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2008.

[Pau] Paul Boddie. Javaclass. http ://www.boddie.org.uk/python/javaclass.html.

[SAKP03] Sergei Skorobogatov, Ross Anderson, çetin Koç, and Christof Paar. Optical fault induction attacks. In Burton Kaliski, editor, *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 31–48. Springer Berlin / Heidelberg, 2003.

[Sko] E. Skoudis. Powerful Payloads : The Evolution of Exploit Frameworks. `http://searchsecurity.techtarget.com/news/1135581/Powerful-payloads-The-evolution-of-exploitframeworks`.

[SSDSTX] Universite de Limoges Smart Secure Devices (SSD) Team XLIM. The CAP file manipulator. `http://secinfo.msi.unilim.fr/`.

[Sun01] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card Platform Security*, 2001. Technical White Paper.

[Sun02a] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Application Programming Interface (API)*, 2002.

[Sun02b] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Runtime Environment (JCRE) Specification*, 2002.

[Sun02c] Sun Microsystems, Inc., Palo Alto/CA, USA. *Java Card 2.2 Virtual Machine Specification*, 2002.

[VF10] Eric Vetillard and Anthony Ferrari. Combined attacks and countermeasures. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application*, volume 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer Berlin / Heidelberg, 2010.

[Vic] Victor Stinner. Hachoir-parser. https ://bitbucket.org/haypo/hachoir/wiki/Home.