Vers une approche de construction de virus pour cartes à puce basée sur la résolution de contraintes

Samiya Hamadouche [1,3], Mohamed Mezghiche[1], Arnaud Gotlieb[2]
Jean-Louis Lanet[3]

[1] Laboratoire LIMOSE, département d'informatique, Faculté des Sciences, Université M'Hamed Bougara, Boumerdès, Avenue de l'Indépendance, 35000 Algérie
{hamadouche-samiya, mohamed-mezghiche}@umbb.dz

[2]Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norvège
arnaud@simula.no

[3]Université de Limoges, 123 rue Albert Thomas, 87000 Limoges, France,
jean-louis.lanet@unilim.fr

Résumé. En tant que supports sécurisés pour l'exécution d'applications et le stockage d'informations, les cartes à puce détiennent et manipulent des informations hautement sensibles. De ce fait, elles sont devenues la cible d'attaques, visant à contourner leurs mécanismes de sécurité afin de s'approprier des données sensibles qu'elles contiennent. Dans notre travail, nous nous intéressons spécifiquement aux *virus activables par attaque en faute*, c'est à dire aux programmes malveillants, pouvant être chargés dans la carte, sans être détectés par les mécanismes de sécurité et qui sont activés uniquement lorsqu'une faute est injectée. Notre objectif est, d'une part, de trouver une méthodologie pour construire de tels programmes, et d'autre part de développer les contre-mesures permettant de se prémunir contre ces virus. La difficulté de notre projet réside dans la construction d'un programme correct vis-à-vis de la spécification initiale, tant de par sa structure que de par sa sémantique, tout en respectant un ensemble précis de contraintes pour le choix de la séquence d'instructions à exécuter. Pour cette construction, nous adoptons une approche fondée sur la Programmation par Contraintes en développant un modèle de satisfaction de contraintes qui caractérise le comportement attendu. Ce papier présente les enjeux de notre projet et détaille les premiers éléments de notre approche.

1 Introduction

Les cartes à puce appartiennent à un domaine très sensible dans lequel l'apport des méthodes de développement formelles peut apporter des garanties, hors de portée des méthodes de développement traditionnelles. Un aspect particulier de la sécurité des cartes à puce concerne la sensibilité à de nombreuses attaques. Par exemple, dans le contexte de la sécurité des cartes Java Card, il a été récemment montré la possibilité de charger du code illégal, au sens où ce code ne respecte pas la sémantique de Java [9]. En principe, ce type de programmes ne doit pas franchir la phase initiale de l'opérateur en charge du déploiement des applications. En effet, ce dernier doit s'assurer que tout programme devant être chargé, respecte les règles de typage Java et de plus, que des règles ad-hoc de programmation sont respectées (e.g., non utilisation de certaines APIs, valeur interdite de certains paramètres, etc.). L'ensemble de ces vérifications font qu'il est, en principe, impossible de charger un code malveillant dans une carte à puce dans un contexte industriel. Cependant, une forme d'attaque dite *attaque combinée*, où l'attaquant modifie l'environnement de fonctionnement de la carte par une perturbation très ciblée, a récemment été mise au point [10]. Ce type

d'attaque permet de charger un code sain, mais capable très subtilement d'exécuter un code hostile, ce dernier étant, soit modifié de manière permanente après son chargement, soit de manière transitoire durant son exécution.

Jusqu'à présent, ces attaques utilisaient un code donné et tentaient de le muter de telle sorte à ne pas exécuter un test de sécurité. La problématique que nous adressons dans ce papier est relative à la conception d'un code malveillant embarqué dans un autre code sain et pouvant muter dynamiquement afin de réaliser des actions malveillantes. Autrement dit, peut-on concevoir un code malveillant caché dans un code sain et ne pouvant être activé que suite à une attaque ?

Notre papier est organisé de la manière suivante : dans la section ci-après (section 2) nous donnons un aperçu des modèles de faute ainsi que des attaques combinées. Ensuite, nous présentons un exemple de construction d'un virus activable par attaque en faute (section 3), qui constitue notre preuve de concept. Puis, nous évoquons les premiers éléments de notre approche (section 4) qui s'appuie sur la Programmation par Contraintes. Enfin, nous concluons par les travaux futurs (section 5).

2 Attaques combinées sur des cartes à puces

Les *attaques par perturbation* sont aujourd'hui considérées comme étant les plus puissantes [3]. Elles peuvent prendre la forme simple de perturbation sur l'alimentation, l'horloge, un ajout d'énergie par sonde électro magnétique ou par effet photo électrique. Ces attaques ont été utilisées essentiellement pour fauter le comportement d'algorithmes cryptographiques mais se sont répandues au système d'exploitation de la carte, à l'algorithme de chargement, à la machine virtuelle et même à l'application elle-même. Tous les éléments de la carte sont attaquables, en fonction de l'intérêt particulier de l'attaquant. L'attaque la mieux maîtrisée est l'attaque par impulsion laser, où les photons accèdent à la couche dopée du silicium et font basculer les transistors. Dès lors, les registres ou bien les cellules mémoire se saturent et peuvent prendre des valeurs précises. Ce type d'attaque est actuellement l'un des moyens les plus efficaces pour obtenir de l'information.

2.1 Modèle de fautes

De nombreux modèles de faute ont été introduits dans la littérature, comme en témoigne le Tableau1. Actuellement, le modèle *Precise Byte Error* est communément admis comme étant à la portée d'un attaquant. Il est donc capable de viser une seule cellule mémoire choisie, à un instant, synchronisé généralement avec le début de la commande. L'effet de l'ajout d'énergie sera la mise à un ou à zéro de tous les éléments de la cellule (bsr : bit set or reset) ou un résultat aléatoire si la mémoire est cryptée.

Tableau1. Les modèles de fautes existants

Modèle de faute	Précision	Position	Timing	Type de faute	Difficulté
Precise bit errors	Bit	full control	full control	bsr	++
Precise byte errors	Byte	full control	full control	bsr, random	+
Unknown byte errors	Byte	lose control	full control	bsr, random	-
Random errors	Variable	no control	partial control	Random	--

Jusqu'à récemment on considérait qu'une seule faute pouvait être réalisée durant une commande. Auquel cas, une simple redondance temporelle permettait de détecter

une attaque. Avec l'usage des diodes laser, de multiples fautes synchronisées peuvent avoir lieu dans la même commande annihilant l'effet d'une telle redondance.

2.2 Attaques combinées

À partir de ce mécanisme primaire (saturation des cellules ou registres) l'effet de la faute se propage dans le système et se transforme en erreur, dès son activation. Barbu *et al.* ont proposé [1] un moyen de parvenir à utiliser une arithmétique de pointeur sur une plate-forme Java Card. Le principe de l'attaque repose sur la perturbation du code natif exécuté lors d'une conversion de type entre une instance d'une classe et une instance d'une autre classe n'appartenant pas à la même branche du treillis de types. Au final, l'attaquant pourra accéder au même objet avec deux références de type différent. Dans [2], les auteurs décrivent plusieurs attaques basées sur la perturbation de la pile d'opérande, en particulier, si la valeur est une variable booléenne précédent l'exécution d'un saut conditionnel. Bouffard *et al.* ont proposé dans [4] de tirer parti de la rupture du flot d'exécution à la fin d'une boucle *for* en mettant à profit le sens du déplacement d'un offset pouvant amener jusqu'à exécuter des opérandes. Dans sa thèse, Kauffmann-Tourkestanky [8] montre que la probabilité d'avoir un code désynchronisé long, *i.e.* un décalage dans le flot d'instructions interprétées avant de retrouver le flot initial, est faible. Mais son approche nommée *durée de vol* est basée sur une distribution uniforme des byte codes et sur la distribution équiprobable des arguments sur le domaine d'un octet. Il montre que la désynchronisation ne peut excéder 1,53 instruction et donc l'exploitation est faible. Cependant pour un attaquant, le code est choisi et ne suit donc pas une distribution uniforme comme nous le montrons dans la section suivante.

3 Conception expérimentale

3.1 Principe général

Notre objectif est de pouvoir cacher un code hostile dans un code sain tel que ce dernier ait une sémantique correcte, vis-à-vis de la machine virtuelle (*i.e.* considéré correct par le vérifieur de byte code), même après l'injection de la faute. Considérons un modèle de faute *Precise Byte Error* et une mémoire non cryptée. Lorsque la faute arrive, l'instruction stockée dans la case mémoire-cible se transforme en une instruction NOP (`0x00`) et son opérande devient potentiellement une instruction valide. Ainsi, pour cacher le code du virus, il faut insérer une/plusieurs instruction(s) (qui fera l'objet de l'injection de la faute) juste avant son début de telle sorte que certaines contraintes soient respectées afin de contourner les contre-mesures de la carte (elles ne détecteront pas le virus lors de l'exécution du programme).

3.2 Preuve de concept : exemple de virus

Nous avons dans un premier temps invalidé l'hypothèse de Kauffmann-Tourkestanky [8] en montrant qu'avec un code choisi la désynchronisation peut être longue amenant l'exécution d'un code exploitable.

Nous avons considéré une application dans laquelle le programme utilise une clé cryptographique initialisée lors de la phase de personnalisation et donc inconnue pour le concepteur de l'application. Le but du virus est d'envoyer cette clé en clair à l'extérieur de la carte. Le code java de l'exemple considéré est le suivant :

```
public void process(APDU apdu) {
     ... // variables locales
     byte[] apduBuffer= apdu.getBuffer();                          B1
     if (selectingApplet()) {return;}
     byte readByte = (byte) apdu.setIncomingAndReceive();
-------------------------------------------------------------------------
     DES_keys.getKey(apduBuffer, (short) 0);                       B2
-------------------------------------------------------------------------
     apdu.setOutgoingAndSend((short) 0, DES_keys.getSize());       B3
}
```

Ce code peut être décomposé en trois blocs :

- B1 et B3 le code sain qui doit être exécuté.
- B2 la commande qui décrypte la clé et la dépose en clair dans le buffer de sortie de l'application. C'est le code à cacher et qui sera exécuté suite à l'injection de la faute uniquement.

Nous avons pu montrer dans [7] comment cacher manuellement un tel code. Nous avons commencé par la résolution statique des liens à l'extérieur de la carte afin de récupérer la référence de la méthode `getKey` (à travers l'attaque de Hamadouche *et al*, [6]). Après cela, une instruction supplémentaire a été insérée juste avant le code à cacher (on opère au niveau byte code). Le code final obtenu est un code valide, il vérifie les règles de typage d'où il peut être chargé dans la carte sans être détecté par le vérifieur de byte code. De plus, il satisfait les règles de programmation Java Card donc il n'est pas considéré comme étant dangereux. Après un tir laser modifiant le byte code exécuté, la transformation de la sémantique permet à l'attaquant de renvoyer la clé à l'extérieur de la carte.

4 Approche proposée

4.1 Éléments de modélisation

Cacher un code hostile dans un code sain revient à trouver, parmi l'ensemble des instructions possibles, une séquence d'instructions qui permette, à partir d'un état donné de la mémoire (*i.e.*, le début du code à cacher) de rejoindre un fragment de code sain comme illustré par la figure ci-dessous (Fig.1). Nous devons garantir que l'insertion de ces instructions se fasse en respectant les contraintes qui seront vérifiées ultérieurement par le vérifieur de byte code. Par exemple, la pile ne doit pas dépasser la valeur stockée dans le header de la méthode, elle ne doit pas créer de dépassement par le bas, le nombre de paramètres locaux est fixé, les types produits et consommés doivent être compatibles avec l'état courant de la pile, la pile doit être vide au début et à fin de l'exécution, *etc*.

L'objectif de notre approche est donc, à partir d'un état partiellement connu de la mémoire, représenté par la pile et les paramètres locaux, d'insérer des instructions et de recalculer l'état mémoire précédent afin de converger vers l'état mémoire à rejoindre. L'état connu est partiel puisqu'il résulte de l'exécution des instructions antérieures qui ont pu produire des effets sur la pile. La construction de la séquence doit résoudre deux problèmes : le choix d'une instruction parmi celles qui existent dans le langage Java Card et le calcul de l'état mémoire précédent cette instruction. L'objectif du choix de l'instruction est de se rapprocher de cet état mémoire tout en respectant les contraintes Java. La fonction de décision du choix de l'instruction à insérer doit être accompagnée d'un mécanisme de retour arrière si la séquence choisie ne permet pas de rejoindre l'état désiré.

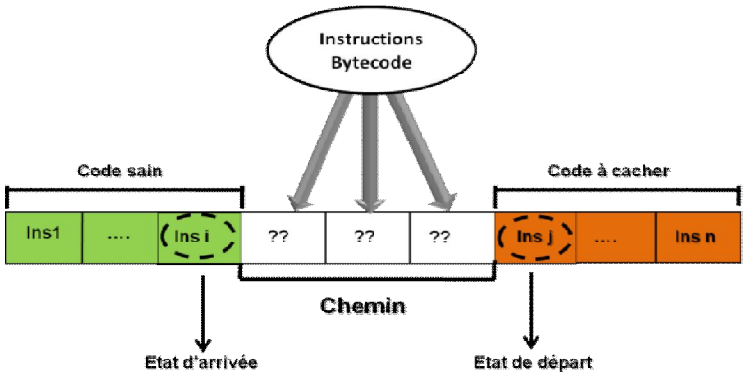

Fig. 1. Principe général de l'approche

4.2 Utilisation de la Programmation par Contraintes

La construction d'un code malveillant, qui sera chargé dans la carte à puce, activable par une attaque en faute se modélise ainsi comme un Problème de Satisfaction de Contraintes (CSP). Bien entendu, les contraintes envisagées sont d'une grande complexité, puisqu'il s'agit de considérer chacune des instructions byte code Java Card comme étant une relation entre deux états mémoire : l'état mémoire avant et l'état mémoire après l'exécution de l'instruction. Dans notre travail, nous avons pris le parti de réutiliser les travaux existants de Charreteur et Gotlieb [5] qui ont défini un modèle relationnel d'un sous-ensemble conséquent des instructions byte code Java dans le but de générer des données de test pour des programmes écrits dans ce langage. Ce modèle, écrit en Prolog avec contraintes, établit pour chaque instruction byte code, une relation entre deux états mémoire, nommées *EMC (Etat Mémoire sous Contrainte)*, et implémente des règles de déduction fonctionnant dans le sens de l'exécution, mais aussi dans le sens inverse. C'est très précisément cette capacité qui nous intéresse afin de calculer l'état mémoire précédant une instruction choisie lors de la construction de notre séquence.

4.3 Calcul des états mémoire

La création des EMCs nécessite la modélisation des données manipulées par la machine virtuelle Java et des zones de stockage de ces données à savoir les registres, la pile d'opérande et le tas. Un état mémoire est défini comme étant un triplet *(f, s, H)* où *f* est une fonction pour les registres, *s* la séquence des variables (de type primitif ou référence) qui modélisent la pile, et *H* une fonction représentant le tas.
L'effet de l'exécution de chaque instruction byte code est exprimé sous forme d'une relation entre deux états de la mémoire. La relation entre les états de la mémoire avant et après l'exécution d'une instruction se traduit notamment par des contraintes qui portent sur les éléments composant les EMCs.

La méthode de génération des données de test (états mémoire) proposée dans [5] raisonne dans le sens inverse à celui de l'exécution : partant d'un objectif (une instruction à couvrir) elle essaie de trouver un chemin menant vers le point d'entrée du programme en explorant progressivement et à l'envers le graphe de flot de contrôle, puis de déterminer une donnée d'entrée qui puisse activer un tel chemin. Au cours de la construction de ce dernier, les instructions parcourues sont modélisées par des contraintes. Pour ce faire, le modèle relationnel à contraintes est exploité et permet de raisonner automatiquement sur les EMCs. Ces derniers contiennent les variables sous contraintes du CSP et sont progressivement instanciées jusqu'à obtenir

un chemin final complet permettant d'atteindre l'instruction sélectionnée. Afin de mettre en application ce modèle et la méthode de génération, l'outil JAUT (Java Automatic Unit Testing) a été développé [5]. Il prend en entrée un fichier byte code traduit dans un langage intermédiaire, l'objectif de test (instruction à couvrir), ainsi que certains paramètres de génération (pour le parcours du graphe) et restitue des états mémoire en entrée permettant de couvrir l'instruction cible (objectif de test). La réutilisation de cette approche pour déterminer les instructions nécessaires à l'établissement de l'attaque en faute qui nous intéresse reste encore à valider, mais cette piste semble très prometteuse.

5 Conclusions et travaux futurs

Dans ce travail en cours, nous raisonnons sur la possibilité de cacher un code hostile dans un code sain et qui sera activé par une attaque en faute une fois chargé dans la carte à puce. Nous avons montré qu'on pouvait ramener ce problème à un problème de satisfaction de contraintes par la recherche automatique de séquence d'instructions. Notre approche s'appuie sur un modèle à contraintes qui caractérise chaque état mémoire par une structure partiellement connue dont la reconstruction fait l'objet du problème de satisfaction de contraintes. Cependant, le travail antérieur présenté dans [5] va être adapté et complété afin de prendre en compte les spécificités du langage Java Card (notre plateforme cible). De plus, cette approche reste à évaluer précisément, pour en démontrer complètement la faisabilité et l'efficacité. Néanmoins, nous pensons qu'elle est très prometteuse.

Références

1. Barbu, G., Thiebeauld, H., Guerin, V., *Attacks on Java Card 3.0 Combining Fault and Logical Attacks*, Gollmann D., Lanet J.-L., Iguchi-Cartigny J., Eds., Smart Card Research and Advanced Application, vol. 6035 de Lecture Notes in Computer Science, p. 148-163, Springer Berlin / Heidelberg, 2010.
2. Barbu, G., Duc, G., Hoogvorst, P., *Java Card Operand Stack : Fault Attacks, Combined Attacks and Countermeasures* , PROUFF E., Ed., Smart Card Research and Advanced Applications, vol. 7079 de Lecture Notes in Computer Science, p. 297-313, Springer, 2011.
3. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks. *Proceedings of the IEEE, 94*(2), 370-382, Février 2006.
4. Bouffard, G., Lanet, J.-L., Iguchy-Cartigny, J., *Combined Software and Hardware Attacks on the Java Card Control Flow*, PROUFF E., Ed., Smart Card Research and Advanced Applications, vol. 7079 de Lecture Notes in Computer Science, Berlin Heidelberg, Springer, p. 283–296, September 2011.
5. Charreteur, F., Gotlieb. A.: *Constraint-based test input generation for java bytecode*. In Proc. of the 21st IEEE Int. Symp. on Softw. Reliability Engineering (ISSRE'10),San Jose, CA, USA, November 2010.
6. Hamadouche, S., Bouffard, G., Lanet, J.-L., Dorsemaine, B., Nouhant, B., Magloire, A., Reygnaud, A.: *Subverting Byte Code Linker service to characterize Java Card API*, S. SARSSI 2012, Cabourg, France, Mai 2012
7. Hamadouche, S., Lanet, J-L.: *Virus in a smart card: Myth or reality?*, Journal of Information Security and Applications, Septembre 2013.
8. Kauffmann-Tourkestansky, X. : *Analyses sécuritaires de code de carte à puce sous attaques physiques simulées*, Thèse de Doctorat, Université d'Orléans, 2012.
9. Oracle. Java Card 3.0.1 Specification, 2009.
10. Vetillard, E., Ferrari, A.: Combined Attacks and Countermeasures. In *Smart Card Research and Advanced Application*, vol. 6035 of *Lecture Notes in Computer Science*, pages 133–147. Springer, 2010.