Attaques physiques à haut niveau pour le test de la sécurité des cartes à puce

P. Berthomé*, K. Heydemann†, X. Kauffmann-Tourkestansky‡, J.-F. Lalande*
*Centre-Val de Loire Université, Ensi de Bourges, LIFO, 18000 Bourges
†Université Pierre et Marie Curie, LIP6, 75252 Paris
‡Oberthur Technologies, Ensi de Bourges, LIFO, 92726 Nanterre
Emails : pascal.berthome@ensi-bourges.fr, karine.heydemann@lip6.fr,
x.kauffmann@oberthur.com, jean-francois.lalande@ensi-bourges.fr

***Résumé*—Dans cet article, nous proposons de décrire les hypothèses d'attaques physiques contre les cartes à puce afin de modéliser ces attaques à haut niveau. Cette modélisation cherche à représenter l'attaque au niveau du langage C par l'injection d'un morceau de code qui simule ses effets. L'intérêt du modèle est qu'il permet de simuler les attaques possibles à un niveau où le programmeur peut comprendre les effets sur le code qu'il développe. Cependant, le nombre d'attaques possibles est très grand ce qui empêche la réalisation exhaustive de tous les tests. Les résultats expérimentaux montrent comment identifier par simulation les attaques par saut qui aboutissent. Enfin, nous présentons nos perspectives de travaux qui concernent la vérification statique de ces codes attaqués.**

I. Introduction

Les attaques contre les cartes à puce peuvent être physiques [1]. En surveillant l'activité d'une carte à puce, des impulsions lasers peuvent réussir à perturber l'exécution du programme qui s'y déroule. Les conséquences sont difficilement contrôlables mais, à force de tentatives, peuvent conduire l'attaquant à réussir une perturbation qui lui donne un avantage par rapport aux sécurités implémentées dans la carte.

Contrecarrer ces attaques est un problème proche de l'étude du domaine de la tolérance aux fautes. [6] donne une comparaison très précise de ces deux problèmes. La différence majeure réside dans le fait qu'une faute est une erreur aléatoire alors qu'une attaque est une faute semi-contrôlée qui cherche à perturber un point précis du programme. De plus, les logiciels tolérants aux fautes cherchent souvent à être robustes à ces fautes. Pour la carte, la détection de l'attaque ou le crash du programme sont préférables à la continuation de l'exécution.

Dans le processus de développement du code pour carte à puce, représenté sur la figure 1, l'étude des attaques peut se faire en attaquant la carte physiquement, par exemple en étudiant la consommation énergétique ou le champ magnétique de celle-ci [4], [5]. Le matériel dévolu à ce type d'étude est coûteux et le processus long. Il est aussi possible de réaliser une campagne de tests dans un environnement simulé, par exemple en perturbant le binaire obtenu après compilation. Si une telle étude est plus aisée à mettre en pratique, l'étude des attaques physiques n'en est pas moins difficile.

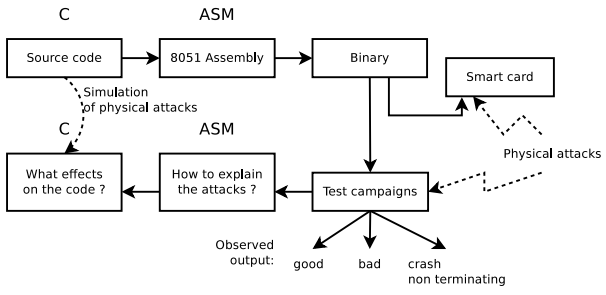

FIGURE 1. Méthodologie proposée

Comme représenté sur la figure 1, l'évaluation d'une attaque peut être classée dans trois catégories. La catégorie «good» est une exécution où l'attaque ne fait pas dévier le comportement du programme, par exemple le contenu de la sortie attendue. Dans le cas de la carte à puce, on peut considérer les trames de transaction avec le terminal. La catégorie «bad» correspond aux exécutions où l'attaque permet d'obtenir une plus-value pour l'attaquant. Cette plus-value peut être la fuite d'une information exploitable dans les trames de transaction ou un comportement déviant du comportement attendu. Dans ce cas, l'attaquant est susceptible de raffiner son attaque et d'arriver à produire une attaque encore plus dangereuse. Une dernière catégorie regroupe les exécutions où l'attaque produit un crash ou bloque le programme : ces deux comportements sont «bons» d'un point de vue de la sécurité.

L'étude d'une attaque donnant un résultat classé «bad» est un problème difficile puisqu'il s'agit de comprendre les effets d'une perturbation du code exécutable, par exemple la modification d'une opération arithmétique ou le saut d'un point à un autre du programme, comme présenté en Section II-A. Il est encore plus difficile, mais pas impossible, de revenir au code assembleur puis au code source C originel pour comprendre ce que l'attaque physique a perturbé au niveau des variables ou du contrôle de flot. Pour contourner cette difficulté, nous proposons de simuler les attaques directement au niveau du code C afin de s'affranchir des étapes passant par le code assembleur.

II. Méthodologie

Cette section présente un exemple d'attaque et les grands principes de la méthodologie de simulation proposée.

A. Exemple d'attaque

On considère le code représentant la vérification du PIN :

```
CLR A
MOV R5,A
MOV R4,A
MOV R7,#0D2H
MOV R6,#04H
LCALL _check_pin
MOV  pin_status,R7
RET
```

```
pin_status = check_pin(1234,0000);
return pin_status;
```

La dernière instruction MOV pourrait être attaquée au travers de son second argument, comme montré ci-après. Les effets se propageraient alors jusqu'au RET qui renverrait un mauvais résultat. Du point de vue du C, on peut simuler cette attaque comme dans le listing de droite.

```
CLR A
MOV R5,A
MOV R4,A
MOV R7,#0D2H
MOV R6,#04H
LCALL _check_pin
MOV  pin_status ,#OK
RET
```

```
pin_status = check_pin(1234,0000);
pin_status = OK;
return pin_status;
```

D'autres types d'attaques sont possibles, comme la perturbation du code opération d'une instruction (MOV ⇒ NOP), le remplacement d'un code opération par un JUMP, l'écrasement de la valeur d'un registre.

B. Injection d'attaques

L'injection d'attaques simulées au niveau du code source C permet de couvrir un sous-ensemble des attaques possibles au niveau de l'exécutable en imitant les conséquences que l'attaque réelle produit. Les points d'attaque et la nature de l'attaque font exploser le nombre d'attaques à simuler. Par exemple, si l'on considère une fonction à n instructions C et une attaque perturbant l'une des k variables ayant chacune une prise de valeur dans un domaine D, on doit simuler $n.k.|D|$ attaques du type *inst_1 ; attaque ; inst_2 ; ... inst_n ;*. Ceci étant, nous montrons dans [2] comment réduire les points d'attaques aux points pertinents en éliminant les points d'attaques inutiles c'est-à-dire sans effet. Par exemple *... x = attaque; x = 5; ...* est un exemple trivial d'attaque inutile.

Le deuxième type d'attaques intéressantes à couvrir sont les attaques par saut forcé qui se simulent aisément au niveau C en utilisant des *goto*. Il devient raisonnable de simuler l'ensemble des attaques possibles sur une fonction de taille n puisqu'il suffit de tester n^2 attaques possibles.

C. Exploitation du modèle d'attaque

La simulation des attaques physiques au niveau C permet d'obtenir des nouveaux codes C que l'on peut exploiter à l'aide de méthodologies de test ou de vérification. L'élaboration de batteries de tests permettent de s'assurer que le code fonctionnel s'exécute correctement et tolère l'attaque en l'ignorant (cas «good») ou en provoquant un crash (cas «crash»). Le code sécuritaire peut aussi être mis à l'épreuve et se révéler capable de réagir à l'attaque. Dans le cas contraire, il s'agit d'une attaque qui fait dévier le comportement attendu sans qu'une réponse sécuritaire ne se déclenche (cas «bad»).

La Figure 2 montre l'exploitation de la méthodologie sur un cas concret : il s'agit de l'étude du code de bzip2 issu des benchmarks SPEC 2006 au travers de l'attaque exhaustive par injection de sauts dans la fonction *BZ2_compressBlock*. Les axes donnent le numéro des lignes source et destination de l'attaque par saut qui est injectée pour chaque test. On repère aisément la séparation en diagonale entre les attaques devant être interrompues (kill) car provoquant des boucles infinies, et les attaques classées «good» c'est-à-dire ne perturbant pas la sortie de l'exécution de bzip2. Quelques attaques particulières réussissent (cas «bad»), l'exécution se termine et est assortie d'un résultat de compression perturbé. Ce sont précisément

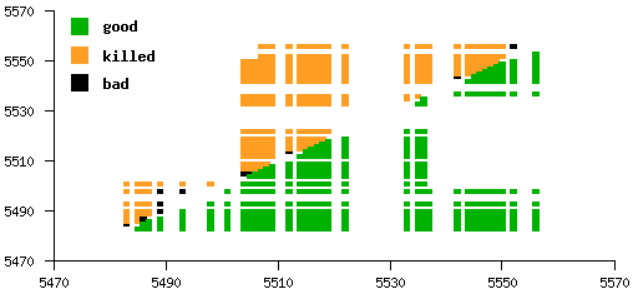

FIGURE 2. Profil de BZ2_compressBlock : classement de l'attaque en fonction de la ligne source et de la ligne destination du saut de l'attaque

ces attaques qu'il convient ensuite d'investiguer pour pouvoir implémenter d'éventuelles contre mesures sécuritaires.

III. CONCLUSION ET PERSPECTIVES

Ce papier présente la modélisation d'attaques physiques au niveau du code source C afin de simuler les effets d'attaques réelles contre le code embarqué dans des cartes à puce. Comme le montre les résultats expérimentaux, la modélisation des attaques à haut niveau simulant leur effet permet de déterminer les attaques qui réussissent.

Par la suite, nous envisageons d'investiguer les possibilités offertes par les outils de vérification statique, tels que Frama-C [3] qui pourront permettre de prouver formellement si une propriété du code source, auparavant garantie sur le code fonctionnel, devient violée pour le code sur lequel l'attaque simulée est injectée. La difficulté majeure de ce type d'approche réside dans l'ampleur du code à analyser et dans la modélisation des propriétés à vérifier.

RÉFÉRENCES

[1] R. Anderson and M. Kuhn. Tamper resistance-a cautionary note. In *Proceedings of the second USENIX Workshop on Electronic Commerce*, volume 2, pages 1 – 11, 1996.

[2] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann Tourkestansky, and Jean-François Lalande. Attack model for verification of interval security properties for smart card C codes. In *5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security PLAS '10*, ISBN :978-1-60558-827-8, pages 1–12, Toronto Canada, 2010. ACM.

[3] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A Value Analysis for C Programs. *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 123–124, 2009.

[4] T.S. Messerges, E.A. Dabbish, and R.H. Sloan. Investigations of Power Analysis Attacks on Smartcards. *Power*, 1999.

[5] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA) : Measures and Counter-measures for Smart Cards. In Isabelle Attali and Thomas Jensen, editors, *Smart Card Programming and Security*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer Berlin / Heidelberg, 2001.

[6] Philippe Teuwen. How to Make Smartcards Resistant to Hackers' Lightsabers ? In J. Guajardo and B. Preneel and A.-R. Sadeghi and P. Tuyls, editor, *Foundations for Forgery-Resilient Cryptographic Hardware*, pages 1–8, Dagstuhl, 2010.

Jean-François Lalande est Maître de conférences à l'ENSI de Bourges dans le Laboratoire d'Informatique Fondamentale d'Orléans (LIFO) depuis 2005. Il s'intéresse à la sécurité des systèmes d'exploitation et des programmes embarqués. Docteur en informatique de l'Université de Nice, il a travaillé sur le dimensionnement de réseaux de télécommunication dans l'équipe Mascotte (INRIA/I3S/CNRS/UNS).