La sécurité matérielle : le cas des consoles de jeux (*modchip*)

Cédric Lauradoux

INRIA Rocquencourt, Projet CODES
78153 Le Chesnay Cedex, FRANCE
www-rocq.inria.fr/codes/
cedric.lauradoux@inria.fr

Résumé La sécurité matérielle est devenue une préoccupation importante pour les fabricants de consoles de jeux depuis une dizaine d'années. Cet aspect de la conception des systèmes électroniques est bien connu dans les domaines bancaires et militaires. Cependant les enseignements tirés de l'expérience bancaire ou militaire ne peuvent s'appliquer que de façon très limitée dans le contexte des consoles de jeux vidéos. En effet, la sécurisation physique d'une console est bien plus complexe que celle d'une carte à puce. De plus, toute la conception d'une console de jeux est orientée vers la performance (prix et vitesse). Les algorithmes et les protocoles cryptographiques introduisent des pertes significatives de performance. Nous verrons que les mécanismes mis en place par les fabricants de console de jeux ont connu des fortunes diverses. Nous aborderons un certain nombre de failles qui ont permis aux attaquants de contourner toutes les mesures de protection grâce aux *modchips*.

1 Introduction

La sécurité matérielle doit garantir l'intégrité d'un système contre des sondes physiques. Cette propriété est capitale pour les systèmes d'information bancaires ou militaires, c'est pourquoi les investigations en sécurité matérielle se sont restreintes à ces deux domaines. Cependant l'avènement des réseaux haut débit et les avancées dans le domaine du multimédia et de la réplication ont engendré de nouvelles problématiques en sécurité. La principale problématique est devenu la lutte contre le piratage. Pour empêcher les copies illicites, les producteurs de contenu ont essayé de renforcer la sécurité des supports d'exécution. C'est particulièrement vrai pour l'industrie du jeu vidéo, qui souffre depuis longtemps de ce problème. Les fabricants de consoles tentent depuis de nombreuses années de modifier le support d'exécution et le support des jeux afin d'empêcher l'exécution des médias illicites.

Les problématiques matérielles rencontrées dans les consoles de jeux sont très différentes des systèmes habituellement traités en sécurité matérielle. En effet, les domaines bancaires ou militaires peuvent se permettre d'employer des *System On Chip* (SOC) sécurisés. Ces équipements permettent d'intégrer dans un composant sécurisé de la mémoire ainsi que de la puissance de calcul. C'est le cas par

exemple des cartes à puce. Les performances de ces systèmes sont relativement modestes et leur coût d'intégration peut être important. Ceci est clairement en contradiction avec les contraintes de conception d'une carte mère de console. La puissance et le coût sont déterminants dans le succès d'une console. Les objectifs sécuritaires que l'on désire atteindre pour les consoles sont différents de ceux du domaine bancaire. Il y une grande différence entre protéger des transactions bancaires (haut niveau de sécurité) et protéger une console de jeu. Le but des fabricants n'est pas la sécurité inconditionnelle mais un compromis entre le coût de sécurisation et le coût de piratage. Le mécanisme de sécurité idéal pour un fabricant rendrait le piratage marginal en obligeant l'attaquant pour le casser à employer des équipements relativement onéreux. A l'heure actuelle, le coût de piratage est relativement modeste, il consiste principalement en un *modchip* dont la valeur correspond à celle d'un FPGA bas de gamme (1-10 dollars).

L'objectif de ce document est de présenter certains aspects de la sécurité lié au processeur. Dans une première partie, nous étudierons les flux de communication en direction du processeur. Nous verrons comment les *modchips* peuvent leurrer les processeurs et permettre l'exécution de copies pirates. Ensuite, nous verrons comment les mécanismes d'optimisation intégrés dans les processeurs peuvent mettre en péril la sécurité des systèmes en créant des canaux cachés ou en provoquant des fuites d'informations.

2 Analyse du *Modchiping*

Un *modchip* est un équipement matériel capable de contourner les sécurités des consoles. Historiquement, les *modchips* étaient conçus pour contourner les restrictions de zonage. Cependant, il n'a pas fallu attendre longtemps avant de voir des *modchips* permettant l'exécution de copies illicites ou de détourner l'utilisation des consoles. Le niveau technologique des *modchips* est très élevé. Ceci vient principalement du fait que les processus de production sont proches des processus industriels. Pour comprendre comment fonctionnent les *modchips*, nous allons étudier l'architecture générique d'une console (Figure 1).

La Figure 1 nous montre que les contraintes de performance sont le cœur du problème en sécurité matérielle. En effet, on ne voit aucun composant dédié à la sécurité comme par exemple le chiffrement des bus. Cette observation n'est plus complètement vraie avec les consoles de dernière génération puisque qu'elles disposeront toutes du *Trusted Platform Module* [3]. Nous allons nous concentrer sur les premiers mécanismes de sécurité des consoles. L'implantation de ces mécanismes est purement logicielle, c'est à dire qu'il s'agit de faire exécuter du code au processeur prouvant l'intégrité du support du jeu. Cette vérification est faite à l'insertion du jeu et elle est constituée de deux éléments :

– un signal émis par le lecteur lors de l'insertion du jeu. Ce signal peut correspondre à une signature placée sur le média du jeu. Cette signature ne doit pas être reproductible et ne peut être calculée que par des éditeurs de jeu

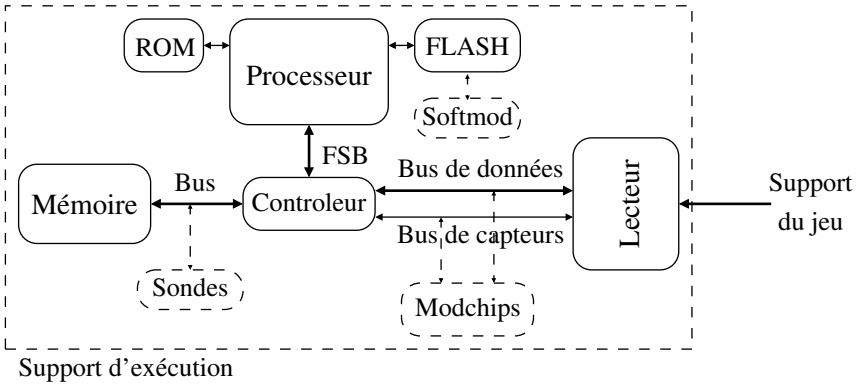

Fig. 1. Architecture générique d'une console de jeu

agréés (signature type RSA). Le signal traverse les bus, pour atteindre le processeur qui sera capable de vérifier la présence d'une signature valide. A ce signal d'intégrité, on associe un certain nombre de signaux annexes comme la détection d'un support, la fermeture du boîtier de lecture...

– une configuration particulière du processeur (*bootload*) qui permet la vérification du signal émis par le support du jeu. Cette configuration consiste principalement dans le chargement du système d'exploitation de la console. Le chargement du *bootload* doit être rapide, ceci impose des contraintes fortes sur le type de mémoire à employer pour le stocker (ROM, FLASH...) [8].

On peut remarquer que cette phase de vérification fait intervenir tous les composants du système (mis à part la mémoire centrale). Cette caractéristique est en contradiction avec les principes de développement d'un système sécurisé. En effet, la quantité de communication croît avec le nombre de participants ce qui signifie que le volume de données critiques qui doit circuler sur les bus sera d'autant plus important. Un point de vue pragmatique de la sécurité matérielle voudrait que les données critiques soient manipulées le moins possible. C'est dans cette perspective que sont employés les *SOC* dans le milieu bancaire car ils éliminent certains transferts d'information sur des bus extérieurs. Pour une console, un *SOC* permettrait de lier le processeur et le *bootload* de manière immuable, empêchant ainsi toute modification par un pirate du *bootload*. L'ajout de ce type de matériel ne serait malheureusement pas suffisant pour garantir l'intégrité d'un média. Les signaux en provenance des périphériques peuvent être détournés par l'attaquant.

Nous allons maintenant détailler les attaques possibles contre la phase de *bootload* et principalement détailler les attaques de rejeu. Ensuite, nous détaillerons les attaques contre la chaîne relayant le signal émis par le support du jeu.

2.1 Attaques contre le *bootload*

L'initialisation d'un processeur consiste à charger un *bootload* depuis un composant mémoire. Comme tout logiciel complexe, le *bootload* est appelé à évoluer (correctif, nouvelle version...). Cette condition impose que la mémoire puisse être réécrite. Il faut aussi que la programmation initiale du composant mémoire soit simple (impératif économique). Les mémoires FLASH sont tout à fait désignées pour ce type de tâche car leur utilisation est très flexible. Dans un système sécurisé, le chargement du *bootload* a un aspect critique. En effet, si un attaquant est capable de modifier le *bootload*, alors il peut configurer le processeur pour son propre intérêt. L'attaquant peut modifier le *bootload* en exploitant une faille logicielle (*softmodding*) ou en reconfigurant physiquement le composant mémoire. Dans ce contexte, les mémoires FLASH ont le désavantage d'être faciles à extraire et faciles à modifier. Le *bootload* pirate peut permettre d'éliminer la phase de vérification ou d'engendrer une réponse toujours positive lors du test du support du jeu. Pour palier ce problème, certains constructeurs ont choisi d'employer une mémoire ROM qui va contenir le code permettant de vérifier l'intégrité d'une mémoire FLASH (Figure 2). L'intérêt d'employer une mémoire ROM est qu'elle est beaucoup plus difficiles à extraire. L'intégrité de la mémoire FLASH est garantie par le chiffrement et le hachage. Le contenu de la mémoire ROM consiste donc principalement en un algorithme de chiffrement symétrique (déchiffrement) et une fonction de hachage. Lors de la vérification, le contenu de la mémoire FLASH sera déchiffré avec la clef du processeur et les hachés vérifiés.

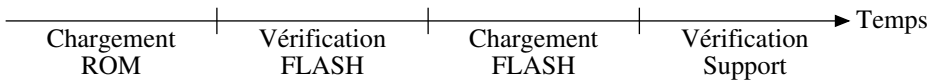

Fig. 2. Phase de vérification d'un support de jeu

L'ajout de la ROM oblige l'attaquant à forger un *bootload* conforme aux algorithmes de vérification. L'analyse de la ROM peut lui fournir l'algorithme employé pour le déchiffrement [4] mais il a aussi besoin de la clef de déchiffrement. Andrew Huang a démontré dans [4] que cette clef ne devait en aucun cas circuler en clair sur un bus. En effet, une sonde peut très bien espionner le bus durant la phase de vérification de la mémoire FLASH et ainsi intercepter la clef.

Ce schéma de *bootload* emploie les mêmes mécanismes de vérification (chiffrement et hachage) que le système XOM [7]. Malheureusement, le chiffrement et le hachage ne permettent pas d'empêcher les attaques de rejeu.

2.2 Aspect théorique : les attaques de rejeu

Le rejeu est une attaque classique en cryptographie. On considère un canal de transmission par lequel deux participants (Alice et Bob) veulent communiquer de façon sûre malgré la présence d'un espion (Eve). Pour cela, Alice et Bob peuvent se mettre d'accord pour employer un algorithme de chiffrement symétrique E et pour choisir une clef secrète k qu'ils partagent tous les deux (je ne considère pas ici les problèmes d'authentification qui sont bien connus). L'algorithme étant sûr, et Eve n'ayant pas la connaissance de la clef, elle n'est pas capable de connaître le contenu des messages. Pour Eve, une attaque par rejeu (Figure 3) consiste, pour une séquence de messages m_i entre Alice et Bob, à enregistrer les communications puis à effacer certaines réponses de Bob (respectivement d'Alice) et à les remplacer par des messages de Bob (Alice) qu'elle a enregistrés.

Alice	Canal	Eve	Bob
$y_1 = E_k(m_1)$	Alice $\rightarrow$ Bob	-	$m_1 = D_k(y_1)$
$m_2 = D_k(y_2)$	Alice $\leftarrow$ Bob	enregistre y_2	$y_2 = E_k(m_2)$
$y_3 = E_k(m_3)$	Alice $\rightarrow$ Bob	-	$m_3 = D_k(y_3)$
$m_2 = D_k(y_2)$	Alice $\leftarrow$ Eve	renvoi y_2	$y_4 = E_k(m_4)$

Fig. 3. Exemple d'attaque par rejeu

La solution classique en cryptographie consiste à employer des numéros de trame (*initial value*) valides pendant la durée d'une transmission donnée (Figure 4). Quand Alice envoie le message m_i, elle le chiffre avec E qui est initialisé avec la clef secrète et un numéro de trame t_i grâce à la fonction f. Le numéro de trame de chaque participant ($t_{A,i}$ et $t_{B,j}$) est incrémenté après chaque transmission, et les numéros de trame de départ doivent être décidés par les participants. Cette solution est implantée grâce au mode compteur (CTR) d'un algorithme de chiffrement à clef secrète.

Alice	Canal	Eve	Bob
$y_1 = E_{f(k,t_{A,0})}(m_1)$	Alice $\rightarrow$ Bob	-	$m_1 = D_{f(k,t_{A,0})}(y_1)$
$m_2 = D_{f(k,t_{B,0})}(y_2)$	Alice $\leftarrow$ Bob	enregistre y_2	$y_2 = E_{f(k,t_{B,0})}(m_2)$
$y_3 = E_{f(k,t_{A,1})}(m_3)$	Alice $\rightarrow$ Bob	-	$m_3 = D_{f(k,t_{A,1})}(y_3)$
$D_{f(k,t_{B,1})}(y_2) \neq D_{f(k,t_{B,0})}(y_2)$	Alice $\leftarrow$ Eve	renvoi y_2	$y_4 = E_{f(k,t_{B,1})}(m_4)$

Fig. 4. Détection du rejeu avec les numéros de trame

Dans le scénario précédent, on disposait de deux parties ayant une puissance de calcul équivalente. La modélisation des interactions processeur - bus - mémoire est complètement différente. En effet, le seul composant actif de ce scénario est

le processeur. La vérification des accès mémoire a été introduite par Blum et al. [1]. L'objectif de Blum et al. était de décrire les mécanismes à inclure dans un processeur pour vérifier une mémoire et le trafic sur les bus lorsqu'un attaquant dispose du contrôle total de ces deux composants (Figure 5).

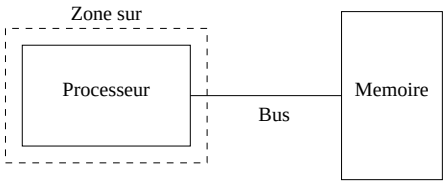

Fig. 5. Processeur : modèle de sécurité

Pour se faire, on peut décrire les actions du processeur comme une séquence d'ordre de lecture/écriture ($load(d_i)/store(d_i)$) à une adresse donnée d_i. La mémoire quant à elle peut faire des écritures ou renvoyer une donnée ($send(d_i)$). L'attaquant peut modifier le contenu des mémoires ou intercepter tout ce qui transite sur le bus. L'attaquant se comporte comme dans une attaque *man in the middle*. Dans ce contexte, on peut représenter le problème du rejeu (Figure 6). Pour ne pas surcharger la notation, le chiffrement des données est implicite dans notre schéma.

Processeur	Bus	Attaquant	Mémoire
$store(d_1)$	Processeur $\rightarrow$ Mémoire	$store(d_1)$	$store(d_1)$
$store(d_2)$	Processeur $\rightarrow$ Mémoire	$store(d_2)$	$store(d_2)$
$load(d_1)$	Processeur $\rightarrow$ Mémoire	-	-
-	Processeur $\leftarrow$ Mémoire	-	$send(d_1)$
$store(d'_1)$	Processeur $\rightarrow$ Mémoire	-	$store(d'_1)$
$load(d'_1)$	Processeur $\rightarrow$ Attaquant	-	-
-	Processeur $\leftarrow$ Attaquant	$send(d_1)$	-

Fig. 6. Combinaison *Man In The Middle* et attaque de rejeu

Pour éliminer ce type d'attaque, on peut tout à fait employer des numéros de trame comme dans le cadre d'un canal de transmission classique. Ainsi on aura un numéro de trame correspondant pour chaque adresse de la mémoire. Cette méthode est malheureusement très coûteuse en mémoire. En effet, pour chaque adresse de la mémoire, le processeur doit stocker en interne le numéro de trame correspondant. Par exemple, si on prend des numéros de trame de longueur 64-bits, avec 10Ko de numéros de trame on peut protéger seulement 1280 adresses

mémoire.

La solution proposée par Blum et al. [1] considère le cas où le processeur a pu exécuter au moins une fois de façon sûre la séquence de lecture/écriture. Il peut ainsi construire une signature du programme à partir de tous les ordres donnés à la mémoire. Pour vérifier un ordre, le processeur devra dans un premier temps recalculer la signature en entier pour chaque ordre donné à la mémoire. Puis, le processeur réalise la comparaison entre la signature recalculée et la signature d'origine. Cette solution impose de stocker la signature sur une zone mémoire protégée. Le schéma de signature proposé par Blum et al. repose sur les arbres de Merkle (Figure 7). La racine de l'arbre constitue la signature et les feuilles représentent l'ensemble des transactions que l'on désire protéger. Les nœuds pères des feuilles sont calculés à partir des feuilles grâce à une fonction de hachage h. En terme de rajout matériel, la solution de Blum et al. est praticable si on peut rajouter de la mémoire sécurisée dans le processeur et si on peut rajouter une fonction de hachage. La question du choix de la fonction de hachage est ouverte. En effet les propositions d'implantation de se schéma ont toutes recours aux fonctions de hachage cryptographique qui ont des coûts de réalisation matérielle rédhibitoires.

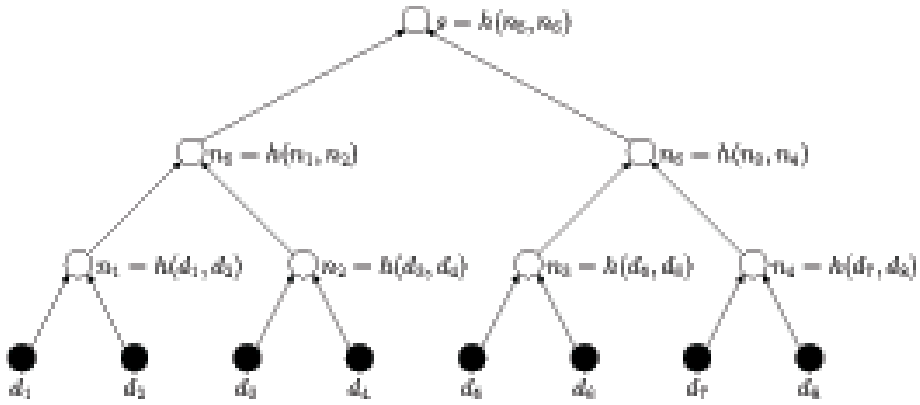

Fig. 7. Un arbre de Merkle

L'accroissement du trafic sur les bus est l'aspect qui pénalise le plus cette solution. En effet, pour remonter dans l'arbre de signature, on doit accéder à toutes les feuilles. Pour une séquence de n ordres à la mémoire, on aura n^2 transactions sur le bus ce qui est tout simplement inacceptable en terme de performance. Un compromis plus acceptable consiste à stocker dans la mémoire principale, les feuilles et tous les nœuds pères à l'exception de la racine. Ceci permet de recalculer la racine à partir de deux feuilles et $O(\log_2(n))$ pères. On

obtient ainsi pour n ordres, $n \times \log_2(n)$ transactions sur le bus en doublant le coût de la mémoire principale.

Nous disposons donc de deux types de solutions pour éviter les attaques de rejeu. L'une des solutions (les numéros de trame) permet de maintenir un trafic des bus normal. Cependant, la taille de la mémoire interne du processeur augmente de manière excessive. La solution de la signature fonctionne pour des processeurs avec une mémoire interne de faible capacité (taille d'une signature). La contrepartie est que la vérification d'un ordre nécessite une forte augmentation du trafic des bus. La résistance aux attaques de rejeu est encore à ces débuts. La question principale est de savoir s'il est possible de construire un schéma de signature sûr ayant un coût en bande passante constant.

2.3 Attaques contre la chaîne de transmission

L'insertion d'un support de jeu provoque l'émission d'un signal d'authentification et la vérification par le processeur de la validité du support. Une fois qu'un signal d'authentification valide est émis, le support est considéré comme étant un original. La détection de l'insertion d'un support est effectuée grâce à un capteur de présence et à un capteur d'ouverture/fermeture du lecteur. La Figure 8 représente les signaux émis par ces différents capteurs lors d'une insertion.

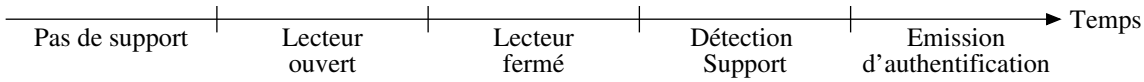

Fig. 8. Détection d'un support de jeu

Le schéma de vérification employé dans les consoles (une seule vérification à l'insertion) rend cette chaîne de transmission très faible. En effet, il suffit de remplacer un support original par une copie illicite pour leurrer le système. Cette attaque requiert de pouvoir tromper les capteurs ou d'éliminer certains signaux qu'ils émettent. C'est ce que permettent les *modchips* de type *swap*. D'une certaine façon, les *swap modchips* transforment le bus des capteurs en un canal à effacement. Ils éliminent certains signaux d'ouverture et de détection de support.

La chaîne de transmission peut aussi être complètement court-circuitée si un attaquant rejoue un signal valide. Cette technique est implantée grâce aux *no-swap modchips*. L'idée consiste à capturer un signal valide qui sera réémis par le *modchip* à l'insertion d'un support. Ce type d'attaque est possible seulement si le signal d'authentification n'est pas corrélé au reste du support. Ces deux attaques très simples montrent bien la fragilité de la chaîne de transmission relayant le signal du support.

3 Le processeur : noyau de l'insécurité

Nous avons vu dans la partie précédente, que le trafic en direction du processeur peut être facilement manipulé par un attaquant. L'origine de ce trafic est difficile à analyser. En effet, les mécanismes internes des processeurs peuvent générer des communications additionnelles et difficilement prévisibles. Ces mécanismes sont transparents pour le programmeur qui en a un contrôle très limité. Le processeur peut donc «accidentellement» faire circuler des données critiques en clair sur des bus non protégés.

3.1 Fuites d'information

Il existe deux hiérarchies mémoire sur une carte mère. Nous avons présenté la hiérarchie permettant d'effectuer le démarrage du système dans la section 2.1. La deuxième hiérarchie permet au processeur d'accéder rapidement à une mémoire de grande taille (Figure 9). L'existence des mémoires caches est due au faites que les entrées/sorties sont devenues un aspect limitant dans les performances des processeurs. On sait concevoir des mémoires lentes de grande capacité et des mémoires rapides de faible capacité. La hiérarchie mémoire présentée Figure 9 illustre le compromis entre les mémoires rapides et les mémoires lentes. Le dernier élément de la hiérarchie mémoire contenu dans le package du processeur est la mémoire cache. Une mémoire cache est un ensemble mémoire divisé en bloc de n octets. L'associativité d'un cache détermine la façon dont sont placés les blocs provenant de la mémoire principale. Un cache associatif par ensemble de taille m divise le cache en ensembles de m blocs. Un bloc de la mémoire centrale peut être placé dans n'importe quel bloc d'un ensemble. Quand on accède à un bloc de données, deux situations peuvent survenir. Si le bloc est dans le cache, on a ce qu'on appelle un *hit*. Dans le cas contraire, on doit aller chercher le bloc dans la mémoire principale et évincer un bloc qui était dans le cache, c'est un *miss*. Nous allons maintenant montrer qu'il est possible d'exploiter le cache

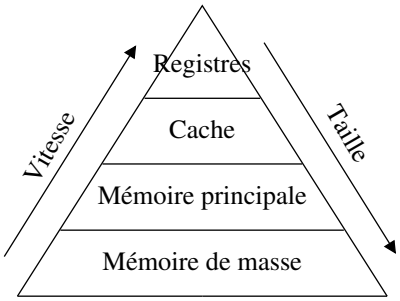

Fig. 9. Hiérarchie mémoire

pour récupérer des informations secrètes. Dans la section 2.1, nous avons vu

que le processeur déchiffre le *bootload* à l'aide du code de la ROM. Les données déchiffrées passent des registres, au cache. Si la taille des données à déchiffrer dépasse la taille du cache alors une partie du *bootload* va transiter sur le bus en direction de la mémoire principale. De manière générale, lorsqu'un processus accède à des données sensibles qui sont placées dans le cache, une interruption ou un processus concurrent peut provoquer la fuite de ces données sur les bus. En effet, l'état du cache est partagé par les processus s'exécutant sur un même support. Les données ne sont pas directement accessibles mais en chargeant un bloc de données, un processus peut provoquer l'éviction d'une donnée ne lui appartenant pas. La recherche de ces canaux cachés a commencé dans les années 90 [13]. Récemment, de nombreuses attaques [10,11] ont été menées contre des algorithmes de chiffrement employant des tables. L'attaquant dispose d'un processus qui inspecte l'état du cache. Plus exactement, ce processus mesure les temps d'accès à une table. Suivant les motifs d'accès mémoire l'attaquant est capable de déterminer la clef secrète employée.

Fig. 10. Trace de de la mémoire cache. L'axe des abscisses représente le numéro de la ligne de cache accédée. Un bloc de couleur foncé signale un *hit* et un bloc de couleur clair indique un *miss*. On voit clairement les zones accédées par le processus espionné (ici un chiffrement AES). La trace a été effectuée sur un Pentium 4 avec un cache de donnée de 8Ko et une taille de bloc de 64 octets.

Les processeurs modernes incluent des fonctionnalités permettant d'éliminer ce type de fuite en employant des verrous (*cache lock*). Quand le verrou est activé, l'état du cache ne peut pas être modifié : tous les accès mémoires seront servis depuis la mémoire principale sans passer par le cache. Les processeurs disposent aussi d'instructions pour invalider le contenu du cache. Lorsqu'une interruption survient le processeur invalide l'intégralité du cache ce qui évite toute fuite de données.

Ces deux types d'instructions permettent d'éliminer toutes les évictions de données sensibles. De tels mécanimes sont inclus dans les processeurs [12] depuis 1999 et employés dans les consoles actuelles.

4 Conclusion

Les cartes mères des consoles de jeux sont difficiles à protéger des attaques matérielles. La raison de cette difficulté vient du fait que la sécurité est encore

secondaire dans le processus de conception. C'est seulement une fois que les performances de la carte mère sont jugées satisfaisantes que les aspects sécuritaires sont envisagés. La complexité des composants apparaît alors aux concepteurs qui sont déjà pieds et poings liés. Les contraintes économiques empêchent le déploiement de composants sécurisés comme le chiffrement transparent des bus ou l'authentification des capteurs. Tous ces facteurs compromettent les chances de voir apparaître des consoles protégeant les droits d'auteur.

Références

1. Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.
2. R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, M. Bardouillet, C. Buatois, and J. B. Rigaud. Hardware engines for bus encryption : A survey of existing techniques. In *Proceedings of the conference on Design, Automation and Test in Europe - DATE '05*, pages 40–45. IEEE Computer Society, 2005.
3. Trusted Computing Group. Trusted computing. `https://www.trustedcomputinggroup.org/home`.
4. Andrew Huang. Keeping Secrets in Hardware : The Microsoft Xbox Case Study. In *4th International Workshop on Cryptographic Hardware and Embedded Systems - CHES '02*, volume 2523 of *Lecture Notes in Computer Science*, pages 213–227. Springer Verlag, 2002.
5. Markus G. Kuhn. Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp. *IEEE Transactions on Computers*, 47(10) :1153–1157, 1998.
6. Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture - ISCA '05*, pages 2–13. IEEE Computer Society, 2005.
7. David Lie, John C. Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *IEEE Symposium on Security and Privacy*, page 166. IEEE Computer Society, 2003.
8. Barr Michael. Memory types. In *Embedded Systems Programming*, pages 103–104, 2001. `http://www.netrino.com/Publications/Glossary/MemoryTypes.html`.
9. Moni Naor and Guy N. Rothblum. The complexity of online memory checking. In *46th Annual IEEE Symposium on Foundations of Computer Science - FOCS 2005*, pages 573–584, 2005.
10. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures : the case of AES. Cryptology ePrint Archive, Report 2005/271, 2005. `http://eprint.iacr.org/`.
11. Colin Percival. Cache missing for fun and profit, 2005. `http://www.bsdcan.org/2005/`.
12. Jim Robertson and Kalpesh Gala. Instruction and data cache locking on the G2 processor core. Technical report, Freescale Semiconductor, Inc., 1999. `http://www.freescale.com/files/netcomm/doc/app_note/AN2129.pdf`.

13. Olin Sibert, Phillip A. Porras, and Robert Lindell. An Analysis of the Intel 80x86 Security Architecture and Implementations. *IEEE Transactions on Software Engineering*, 22(5) :283–293, 1996.
14. Michael Steil. 17 Mistakes Microsoft Made in the Xbox Security System. In *22nd Chaos Communication Congress*, 2005. `https://events.ccc.de/congress/2005/fahrplan/events/559.en.html`.
15. Steve H. Weingart. Physical Security Devices for Computer Subsystems : A Survey of Attacks and Defences. In *Cryptographic Hardware and Embedded Systems - CHES 2000*, Lecture Notes in Computer Science 1965, pages 302–317. Springer Verlag, 2000.