RENPAR'15 / CFSE'3 / SympAAA'2003
La Colle sur Loup, France, 15 au 17 octobre 2003

CryptoPage-2 : un processeur sécurisé contre le rejeu

Cédric LAURADOUX et Ronan KERYELL

Laboratoire d'Informatique & Télécommunications de
l'École Nationale Supérieure des Télécommunications de Bretagne, CS 83818
F-29238 PLOUZANÉ CEDEX

Résumé

Les ordinateurs actuels ne sont pas aussi sécurisés que leur développement ubiquitaire et leur interconnexion le nécessiteraient. En particulier, on ne peut même pas garantir l'exécution sécurisée et confidentielle d'un programme face à un attaquant logiciel (l'administrateur système) ou matériel (analyseur logique sur les bus, contrôle des ressources de l'ordinateur,...).
Dans cet article est proposée une architecture utilisant du chiffrement fort pour se rapprocher de ces objectifs, y compris en ce qui concerne des attaques qui rejoueraient de vieilles données déjà chiffrées et signées par le cryptoprocesseur.
Afin de prouver que les accès du processeur à sa mémoire extérieure sont globalement corrects, on rajoute au niveau du cache un vérificateur utilisant un arbre de hachage de MERKLE stocké lui-même en cache afin d'en accélérer le calcul.
Enfin, un chiffreur d'adresses est rajouté pour diminuer les informations exploitables par analyse du bus d'adresse.

Mots-clés : Sécurité informatique, cryptoprocesseur, cryptographie, arbre de MERKLE, attaques par rejeu.

1. Introduction

De nombreuses applications informatiques nécessitent un niveau de sécurité, de confidentialité et de confiance qui sont malheureusement impossibles à garantir actuellement.

Il existe certes des algorithmes cryptographiques, des protocoles réseaux, des systèmes d'exploitation sécurisés, des applications qui utilisent ces méthodes mais toutes reposent sur une hypothèse forte : le support matériel d'exécution doit être lui-même sécurisé. Or cette hypothèse si critique n'est *jamais* vérifiée, sauf pour de petites applications qui peuvent loger dans une carte à puce par exemple.

En particulier, rien n'empêche un pirate d'instrumenter matériellement avec un analyseur logique un routeur sécurisé pour en extraire des clés de chiffrement (cas de IPsec par exemple) ou plus trivialement à un utilisateur administrateur mal intentionné de tracer pas à pas sur sa machine l'exécution d'un processus, même s'il appartient à un utilisateur distant, afin d'en espionner le programme ou les données, falsifier des dossiers médicaux ou des casiers judiciaires,...

Dans le domaine du calcul distribué, le calcul sur la grille permet typiquement à un individu ou à une société de disposer d'un ensemble d'unités de calcul fournies par des tiers. Or aujourd'hui un utilisateur est forcé de faire totalement confiance au propriétaire d'un tel matériel pour pouvoir y exécuter son code.

La notion d'exécution sécurisée de processus remonte au moins aux années 1960 dans le but d'empêcher l'espionnage durant la guerre froide [17] et le piratage informatique par simple copie : un processeur est spécialisé avec une clé et ne peut exécuter qu'un programme non chiffré ou chiffré avec cette clé. Un voleur récupérant un exemplaire d'un programme pour ce processeur ne pourra rien en faire [1].

Cette problématique est restée très éloignée des considérations du grand public mais avec le développement de processeurs intégrant de plus en plus de transistors, il reste suffisamment de transistors pour intégrer désormais de tels mécanismes de sécurisation et on constate depuis peu une explosion du domaine. Et il est intéressant de constater que si ces notions reviennent à la mode de nos jours [16] c'est aussi pour les mêmes raisons que les motivations d'origine de [1].

Dans cet article ayant pour but de protéger le cryptoprocesseur CRYPTOPAGE-1 [8] contre les attaques par rejeu, on abordera tout d'abord le contexte : la problématique des attaques et les solutions existantes en § 2. Ensuite, on survolera les fonctions de hachages cryptographiques et en particulier les arbres de MERKLE (§ 3) avec comme objectif leur implantation dans un processeur moderne (§ 4). On terminera avec le rajout d'un système de chiffrement des adresses (§ 5).

2. Contexte

Les attaques physiques contre les systèmes informatiques restent relativement méconnues du grand public même si, depuis l'engouement des faussaires pour les cartes à puces et l'apparition de cryptoprocesseur dans les consoles de jeux vidéos (X-BOX,...), le sujet devient de plus en plus en vogue.

Un cryptoprocesseur est un système capable de traiter des données cryptographiques. Pour un adversaire, casser le système revient à combiner des attaques contre l'interface physique et logique. Les principales cibles de ces attaques sont les clefs de chiffrement qui sont stockées dans la puce. Malheureusement les mémoires actuelles peuvent faire l'objet de nombreuses attaques :

- les mémoires (flash) peuvent imprimer définitivement les valeurs qu'elles sauvegardent ;
- on peut refroidir les composants jusqu'à atteindre la rémanence de la mémoire [13] ;
- l'exposition des composants aux radiations (fours à micro-ondes,...) impriment les données en mémoires [18] ou les altèrent ;
- dans le domaine des infrarouges, le silicium des composants devient transparent et on peut accéder au contenu mémoire par imagerie [14] ;
- le parasitage des diodes utilisées comme générateur de bruit aléatoire par des sources électromagnétiques peut provoquer la génération de clefs biaisées [18].

Une technique de rétro-ingéniérie brutale mais classique consiste à retirer le boîtier du circuit intégré qui est alors susceptible d'être modifié. On peut ainsi affaiblir les algorithmes et provoquer des failles dans le système.

Pour contrer toutes ces attaques, des techniques classiques sont mises en œuvre comme la mesure de la température, le niveau de radiation, déplacer régulièrement les données, utiliser des matériaux opaques aux infrarouges, vérifier l'intégrité du système et de l'emballage,...

Si ces techniques d'attaques ont été testées sur des microcontrôleurs ou des cartes à puce, elles pourraient l'être encore plus contre des processeurs génériques car ceux-ci n'offrent à la base aucune protection sur ces points. Mais ce n'est même pas la peine : il suffit d'espionner ou de modifier les données directement sur leur bus, analyser le fonctionnement des programmes, exécuter pas à pas ou utiliser les modes de mise au point avec le *scan-path* JTAG et retrouver les clefs de chiffrement ou tout autre secret !

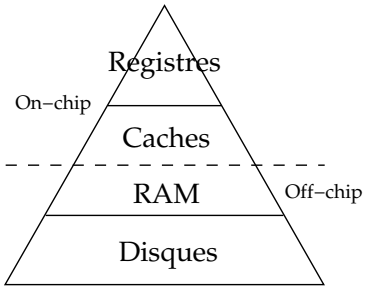


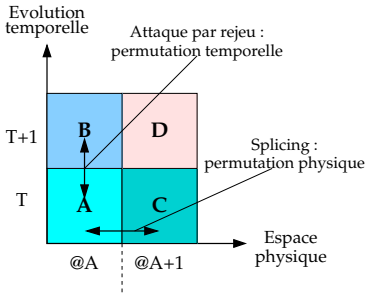
FIG. 1 – La hiérarchie mémoire devient la principale cible d'attaque.

En fait, le principal problème d'un ordinateur moderne est que toute la partie extérieure au boîtier du processeur de la hiérarchie mémoire (figure 1) est incontrôlée : périphériques mémoires, bus, contrôleurs ou bien unités de stockage,...

On peut définir principalement deux catégories d'attaques à ce niveau :

- génération : on cherche à effectuer des attaques avec texte choisi pour casser les algorithmes. Pour cela l'attaquant doit pouvoir injecter ses propres données. C'est ainsi que le microprocesseur DALLAS DS5002FP a été cassé [9]. L'antidote consiste à rajouter à chaque mot mémoire une signature électronique liée de manière unique au processeur [6, 8, 11] : si le mot n'est pas signé avec la signature du processeur l'exécution s'arrête. Ce problème est donc résolu et n'est pas abordé ici ;
- permutation : ce type d'attaque vise plutôt à modifer le comportement des programmes afin d'obtenir des privilèges ou d'effectuer des attaques par débordement. En effet, puisque le point précédent empêche l'attaquant de générer des données correctement signées, il peut essayer de substituer des données générées par le processeur, donc correctement signées, avec d'autres (figure 2). Ce sont des attaques par rejeu car consistant à rejouer une vieille données à une adresse différente ou pas.

Dans un système à mémoire classique, la valeur récupérée lors d'un accès en lecture à l'adresse a correspond (heureusement) à la valeur écrite lors du dernier accès en écriture effectué à cette adresse. Cela signifie donc qu'un accès en écriture révoque la donnée précédemment stockée. Cette méthode de révocation par écrasement physique est satisfaisante en temps normal, mais si on considère que tout le chemin de données n'est pas sûr, alors ce mode de révocation n'est plus suffisant : un attaquant pourrait très bien se contenter de bloquer certaines écritures en mémoire et si, comme sur l'exemple de la figure 2, c'est la zone qui contient le compteur de boucle `i` faire itérer le processeur plus que nécessaire en gardant `i` constant et donc faire sortir des informations confidentielles par débordement de la zone pointée par `p`. Il s'agit donc d'un cas d'attaque par permutation trivial.




```
for(i = 0; i < TAILLE; i++)
   affiche(*p++);
```

FIG. 2 – Les différentes attaques par permutations.

De façon similaire, en exécutant pas à pas un programme, on peut très bien aller modifier les mémoires en cours d'utilisation. Comme aucun mécanisme, mis à part l'effacement physique de la donnée, ne permet de révoquer celle-ci, cela signifie qu'un adversaire enregistrant les valeurs intermédiaires prises par une adresse physique, est capable d'écraser sa valeur finale par une valeur intermédiaire [5].

C'est la seule faille connue dans des systèmes tels que XOM[1] [11] ou CRYPTOPAGE-1 [8] qui utilisent de la cryptographie forte pour chiffrer et signer les données et les instructions. Comme il n'y a pas de protocole assurant la révocation des données, ces systèmes sont vulnérables au rejeu.

3. Signatures et arbres de Merkle

Dans les systèmes précédents la faiblesse est liée au fait que les données signées en mémoire le sont par morceaux pour éviter d'avoir à revérifier la signature de *toute* la mémoire lors de *chaque* lecture, ce qui plomberait les performances. Du coup un attaquant peut très bien rejouer un vieux bloc signé par le processeur pour en modifier le comportement.

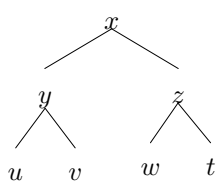
Mais en utilisant des fonctions de signature plus hiérarchiques dans ce contexte, [3, 2] ont proposé d'employer une variantes des arbres de MERKLE comme méthode pour effectuer des vérifications sur les mémoires.

Un arbre de MERKLE est une structure hiérarchique récursive capable de certifier et de révoquer des données. C'est une structure très employée dans les systèmes de fichiers distribués sécurisés (SFS) ou les réseaux pair à pair pour vérifier des données. Dans notre cas (figure 3(a)), on munit les données (nœuds à la base de l'arbre) d'une fonction de hachage à sens unique et d'une structure arborescente ainsi que d'une zone mémoire protégée qui va nous permettre de conserver le nœud racine x.

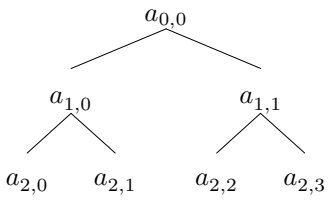
Pour chaque nœud parent y, on aura $y = H(u, v)$ avec par exemple $y = h(u \parallel h(u) \parallel v \parallel h(v))$ où $\parallel$ est la concaténation de chaînes de bits et h une fonction de hachage à sens unique.

Pour valider une donnée u qu'on lit en mémoire il faut vérifier qu'en recalculant la valeur x on retrouve celle stockée en mémoire protégée. Si on ne retrouve pas la même valeur c'est que les données

[1] eXecute Only Memory



(a) Concept



(b) Arbre des données

FIG. 3 – Arbres de MERKLE.

TAB. 1 – Algorithmes de vérification et de mise à jour de la mémoire.

Algorithme de lecture vérifiée $\mathcal{R}_\mathcal{V}(i,j)$	Algorithme d'écriture vérifiée $\mathcal{W}_\mathcal{V}(a_{i,j},i,j)$
$a_{i,j} = \mathcal{R}(i,j)$ **tant que** $(i > 0)$ $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$ $a_{i,f} = \mathcal{R}(i,f)$; $a_{i-1,p} = \mathcal{R}(i-1,p)$ **si** $(a_{i-1,p} \neq H(a_{i,\min(j,f)}, a_{i,\max(j,f)}))$ **erreur** $i = i-1$; $j = p$ `# On remonte l'arbre` **renvoie** $a_{i,j}$	**si** $(i > 0)$ $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$ $a_{i,f} = \mathcal{R}_\mathcal{V}(i,f)$ $a_{i-1,p} = H(a_{i,\min(j,f)}, a_{i,\max(j,f)})$ $\mathcal{W}_\mathcal{V}(a_{i-1,p}, i-1, p)$ `# On remonte l'arbre` $\mathcal{W}(a_{i,j},i,j)$

lues sont corrompues.

Pour faire ce calcul on doit accéder à tous les nœuds situés entre u et la racine x ainsi qu'à leurs nœuds frères (v et z), soit une complexité en $\mathcal{O}(\log n)$ pour une mémoire à n éléments.

On a bien hiérarchisé une fonction de hachage globale de la mémoire car si à un moment donné on est sûr d'une valeur d'un nœud cela suffit pour vérifier les valeurs en dessous.

Attaquer un tel système consiste à modifier les données (u,v,w,t) à signature constante x. Comme on utilise une fonction de hachage cryptographique à sens unique résistant à la 2ème pré-image, connaissant w et t (figure 3(a)), le pirate ne sait pas trouver t' associé au w' qu'il veut injecter qui donnera la même valeur z pour passer inaperçu. Pour peu qu'en plus H ne soit pas connue (par exemple en faisant intervenir une clef secrète et l'adresse comme dans CRYPTOPAGE-2), on complique la tâche de l'attaquant.

On peut noter que si H ne commute pas, on ne peut pas tenter une attaque par permutation spatiale (inverser les u et v précédents dans la mémoire). On pourrait donc se passer de faire intervenir l'adresse dans le calcul de la clé de chiffrement contrairement à [8]. En fait on garde ce calcul pour éviter des attaques à texte connu (typiquement une zone de 0 en mémoire serait trop visible).

4. Réalisation

4.1. Sans cache

On considère qu'on veut protéger une zone de n éléments mémoire (qui auront par exemple la taille des lignes de cache dans une réalisation efficace) qui seront les éléments terminaux de l'arbre représentés $a_{\log_2 n,0}$ à $a_{\log_2 n,n-1}$ sur la figure 3(b). Par mesure de simplification on considère n comme étant une puissance entière de 2. Les nœuds de l'arbre $a_{i,j}$ sont numérotés comme sur la figure 3(b), $i \in [0, \log_2 n]$ étant la profondeur dans l'arbre.

On obtient donc comme algorithmes de lecture vérifiée $\mathcal{R}_\mathcal{V}$ et d'écriture vérifiée $\mathcal{W}_\mathcal{V}$ ceux de la table 1 en définissant :

H fonction de hachage à sens unique ;

$\mathcal{R}(i,j)$ la fonction de lecture dans la mémoire qui renvoie la valeur du nœud $a_{i,j}$ sauf pour $a_{0,0}$, la racine de l'arbre du processus en train de tourner, qui est elle stockée de manière sûre dans le processeur et sauvegardée dans le descripteur de processus chiffré au sens de [8] ;

$\mathcal{W}(a_{i,j},i,j)$ la fonction d'écriture dans la mémoire de la valeur de $a_{i,j}$ sauf pour $a_{0,0}$ qui est écrite dans sa zone réservée ;

$\oplus$ le « ou » exclusif bit à bit sur la représentation binaire de deux entiers.

La variable f représente la seconde coordonnée du nœud frère et p celle du père.

Dans $\mathcal{W}_\mathcal{V}$, la vérification sur la valeur du nœud frère est là pour empêcher qu'un attaquant puisse fournir celle-ci en même temps qu'il y a une écriture par le processeur car sinon le processeur calculerait une nouvelle de valeur de hachage prenant en compte la valeur de l'attaquant.

On constate que pour chaque accès vérifié il faut remonter toute une branche de l'arbre, soit dans le cas de la lecture une complexité en $\mathcal{O}(\log n)$. Pour l'écriture, comme on a à chaque étage une lecture véri-

fiée du nœud voisin cela fait une complexité $\mathcal{O}(\log^2 n)$ mais en supprimant la récursion de l'algorithme on peut descendre aussi en $\mathcal{O}(\log n)$.

4.2. Avec cache

Ce type de vérificateur est coûteux en temps. Aussi, pour augmenter les performances, on peut pipeliner le mécanisme et le paralléliser. On peut aussi effectuer une exécution et une utilisation spéculative des instructions et des données à vérifier, le vérificateur générant une interruption qui bloquera le processeur et les effets de bord en cas d'erreur.

Si on considère que notre processeur dispose de deux niveaux de cache (L1 et L2), alors on peut aussi bien disposer notre vérificateur entre les deux niveaux de cache ou entre le dernier niveau et la mémoire principale (figure 1). Cependant si la première solution est choisie, à chaque échec dans le cache L1, on devra traverser tout l'arbre pour vérifier la donnée ce qui va nuire considérablement au performance du processeur. Les données que doit vérifier notre mécanisme (les nœuds de l'arbre) sont donc des lignes de cache du cache de plus haut niveau contenu dans le boîtier du processeur.

L'implantation des arbres de MERKLE dépend principalement des caractéristiques du cache. On doit donc trouver les meilleurs paramètres pour les dimensions du cache, les meilleures politiques pour l'écriture sur échec et sur succès dans le cache, etc.

CRYPTOPAGE a une architecture de type HARVARD pour des raisons de performance et on utilise un chiffrement du bus de données et d'instructions avec des clefs différentes pour empêcher que les instructions ne puissent s'extraire par erreur sous forme de données comme lors de l'attaque de [9]. On peut donc dupliquer les vérificateurs (figure 4), avec une simplification pour la partie instructions qui n'a pas besoin de gérer les écritures.

Dans le vérificateur de la table 1, il faut effectuer des mises à jour de toute la branche contenant la donnée écrite, ce qui est relativement coûteux. Pour accélérer le processus on va utiliser le cache pour cacher l'arbre de MERKLE lui-même avec comme hypothèse de régime stationnaire que tout ce qui est dans le cache est vérifié comme étant correct. C'est donc lors de chaque écriture par le processeur dans le cache de plus haut niveau qu'il faudra remettre à jour l'arbre et lors de la lecture de la mémoire qu'il faudra vérifier la correction.

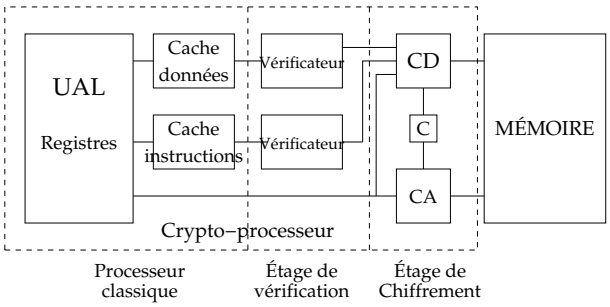


FIG. 4 – CRYPTOPAGE-2.

En considérant les fonctions suivantes, on obtient comme algorithmes ceux de la table 2 avec :

$\mathcal{C}(a_{i,j}) =$ **hit** si la donnée est dans le cache et **miss** sinon ;

$\mathcal{R}_{\mathcal{C}}(i,j)$ la fonction de lecture dans le cache qui renvoie la valeur du nœud $a_{i,j}$, avec un raccourci pour $a_{0,0}$;

$\mathcal{W}_{\mathcal{C}}(a_{i,j},i,j)$ la fonction d'écriture dans le cache de la valeur de $a_{i,j}$ avec un raccourci pour $a_{0,0}$.

TAB. 2 – Algorithmes de vérification et de mise à jour de la mémoire utilisant un cache de valeurs vérifiées.

Algorithme de lecture vérifiée $\mathcal{R}_{\mathcal{VC}}(i,j)$	Algorithme d'écriture vérifiée $\mathcal{W}_{\mathcal{VC}}(a_{i,j},i,j)$

si $(i = 0 \vee \mathcal{C}(a_{i,j}) =$ hit$)$
  renvoie $\mathcal{R}_{\mathcal{C}}(i,j)$
$f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$
$a_{i-1,p} = \mathcal{R}_{\mathcal{VC}}(i-1,p)$
$a_{i,j} = \mathcal{R}(i,j)$; $a_{i,f} = \mathcal{R}(i,f)$
si $(a_{i-1,p} \neq H(a_{i,\min(j,f)}, a_{i,\max(j,f)}))$
  erreur
$\mathcal{W}_{\mathcal{C}}(a_{i,j},i,j)$; $\mathcal{W}_{\mathcal{C}}(a_{i,f},i,f)$
renvoie $a_{i,j}$
	si $(i > 0)$
	  $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$
	  $a_{i,f} = \mathcal{R}_{\mathcal{VC}}(i,f)$
	  $a_{i-1,p} = H(a_{i,\min(j,f)}, a_{i,\max(j,f)})$
	  $\mathcal{W}_{\mathcal{VC}}(a_{i-1,p},i-1,p)$ `# On remonte l'arbre`
	$\mathcal{W}_{\mathcal{C}}(a_{i,j},i,j)$

Les politiques classiques de gestion du cache ne sont pas détaillées ici et sont cachées (*sic*) dans les fonctions $\mathcal{R}_\mathcal{C}$ et $\mathcal{W}_\mathcal{C}$.

On utilise ici l'astuce de stocker systématiquement dans le cache le frère de tout nœud qui doit y être puisque le vérificateur utilise systématiquement les frères. Cela signifie qu'on a aussi intérêt à ranger en mémoire les frères consécutivement pour en accélérer le chargement.

Une autre solution consiste à considérer des caches séparés pour les données utiles et pour les données de vérification. En effet, avec une solution à cache unique, on peut penser que les nœuds de révocation chargés dans le cache entraînent une pollution plus importante diminuant les performances. Cela donne l'architecture découplée de la figure 5.

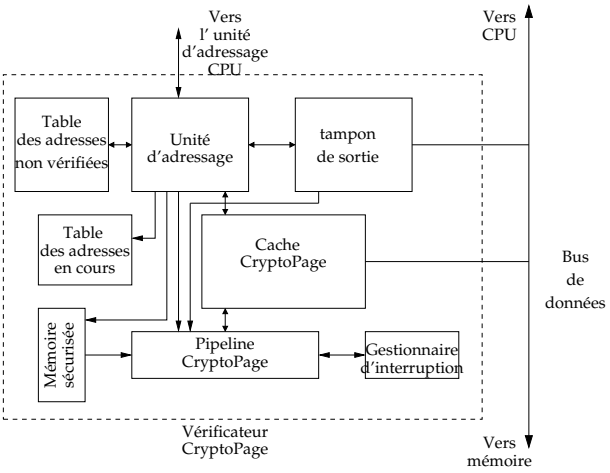


FIG. 5 – Vérificateur avec cache séparé.

5. Chiffrement des adresses

Le flux d'adresses sortant d'un processeur donne des informations sur le comportement d'un programme : flux de contrôle, structures des données, etc. Ce sont autant d'informations qui peuvent aider un attaquant passif à reconstruire des informations secrètes : structures de boucles, de conditions,...

C'est pour ces raisons que le chiffrement des adresses a été étudié d'un point de vue théorique [7] ou réalisé [4]. Le problème est que ces systèmes doivent être compatibles avec l'usage courant d'un processeur général : efficaces, être utilisables par des systèmes d'exploitation classiques et des systèmes de gestion mémoire (MMU).

Comme la gestion mémoire est l'apanage des systèmes d'exploitation et qu'on ne veut pas forcément faire tourner un système par cryptoprocessus, on a intérêt à avoir un système de chiffrement des adresses compatible avec les systèmes d'exploitation classiques. Concrètement le système va voir les adresses dans les TLB, donc il faut qu'elles soient chiffrées avant d'arriver dans la MMU. En outre on ne veut pas casser la hiérarchie du système et on a besoin d'avoir des zones chiffrées et des zones non chiffrées [8], par exemple pour les entrées-sorties en clair.

Dans un système paginé classique une adresse est de type $A = p\|l\|d$, respectivement pour le n° de page, de ligne de cache et le déplacement dans cette dernière. Pour limiter les problèmes, on va coller à cette organisation et trouver un compromis entre obscurantisme et usage.

La solution retenue est de diviser la mémoire en 2 parties, une chiffrée et une non chifrée pour simplifier la plage d'image des adresses chiffrées. Avec les processeurs modernes et leur adressage sur 64 bits ce n'est pas un problème car on a de la place.

L'adresse est chiffrée au niveau de p et de l séparément. Cela signifie que la localité dans la page est repectée mais, par contre, que cela fournira plus d'information à l'attaquant. Au sein de la ligne de cache, c'est le système de chiffrement par bloc standard de CRYPTOPAGE qui brouille les bits de données et donc l'adresse ne sort pas au niveau du bus, tout se gère au niveau du cache. On pourrait imaginer étendre ce mélange des données au niveau d'une page mais cela nécessiterait d'avoir une page entière dans le processeur pour pouvoir lire ou écrire ne serait-ce qu'un bit dans une page.

En mode chiffré, les appels systèmes d'allocation mémoire de style `mmap()` ne peuvent plus fonctionner avec des adresses quelconques puisque le système d'exploitation lui-même ne connaît pas la clé du processus mais peuvent encore être utilisés au niveau de pages entières car, dans ce cas, le système ne rentre pas au niveau des l ou d.

Il est certain que l'analyse macroscopique des flux de pages anonymisées fournira encore de l'information à un attaquant mais beaucoup moins qu'avec l'usage d'adresses non chiffrées. On aura tout intérêt à coupler aux techniques matérielles des techniques logicielles d'offuscation de code (générateur de programmes spaghetti avec des branchements inutiles dans tous les sens,...). Il est clair que les performances pourront en pâtir mais les techniques de cache et de cache de trace devraient les compenser

en grande partie.

6. Travaux futurs

On est loin d'avoir un système fonctionnel, tant au niveau architectural que logiciel.

D'un point de vue matériel, il reste à modéliser plus finement le système à l'aide d'un simulateur de processeur afin de voir l'implication sur les performances en fonction des différentes tailles de cache et du surcoût mémoire acceptable. C'est un travail en cours en utilisant le simulateur SimpleScalar [10].

On peut s'attendre néanmoins à ce que les performances restent bonnes car le système est vu comme un ralentissement entre le cache intérieur au processeur le plus externe et le monde extérieur. De plus la partie chiffrement peut être parallélisée et pipelinée jusqu'à obtenir les performances requises car le nombre de transistors n'est plus un problème.

On a supposé que le processeur était seul dans l'ordinateur mais il faudrait étudier le cas de multi-processeurs à couplage fort ou des processeurs SMT, sans passer par des zones mémoire non chiffrées.

Au niveau logiciel, de manière plus prospective, il faudrait étudier l'adaptation d'un système d'exploitation pour utiliser les primitives de base décrites en [8], comment permettre le clonage de processus, comment gérer les zones non utilisées, etc.

Afin de diminuer les canaux cachés découlant de l'exécution du programme il faudrait développer les techniques de compilation permettant de lisser le comportement d'un programme indépendamment de son contenu : code spaghetti, équilibrage des branches des tests,...

Si certaines applications ont leurs performances trop dégradées on peut imaginer sortir des fonctions ou des bibliothèques du mode sécurisé. Si on veut automatiser et sécuriser cette tâche cela signifie de développer des compilateurs interprocéduraux qui seront capables de prouver qu'un attaquant ne peut pas modifier de manière sensible le comportement du programme en modifiant les parties non chiffrées.

Enfin, en ce qui concerne les usages, comme le concept de cryptoprocesseur est balbutiant, il s'agit d'imaginer les bons (et mauvais) usages qu'on peut faire d'une telle technologie.

7. Conclusion

Nous avons présenté une architecture minimale au sens du RISC permettant de sécuriser l'exécution de processus au sein d'un processeur en préservant la confidentialité des informations échangées avec la mémoire, instructions et données. La mémoire étant signée globalement, il n'est pas possible de modifier discrètement des parties de celle-ci afin de modifier le comportement d'un processus et d'en extraire de l'information.

Le déploiement de ces techniques, que ce soit CRYPTOPAGE ou d'autres [15, 11, 12], dans tout processeur est amené à révolutionner le domaine de l'informatique en permettant de faire émerger de nouvelles applications sécurisées : album de cartes à puce virtuelles, stockage sécurisé de clefs privées, code mobile à exécution garantie, grilles de calcul et ordinateurs parallèles très répartis enfin sécurisés, des services *web* de confiance, des routeurs sécurisés, des systèmes de vérification des droits d'usage, etc.

Il est assez surprenant de constater qu'actuellement les constructeurs grand public sont plus motivés par les problèmes de piratage de contenus que par les problèmes de sécurité informatique en soi, peut-être parce que certains systèmes d'exploitation industriels bien connus ont démocratisé le concept de virus informatique au point que cela semble inhérent à l'informatique, ou plutôt à la bureautique.

Néanmoins ce sont les mêmes techniques matérielles qui sont utilisables en sécurité informatique et en protection contre la copie de contenu et on peut espérer que cela fera progresser la sécurité de conserve : le programme qui vérifie les droits d'usage n'est rien d'autre qu'un cryptoprocessus de plus qui tourne sur la machine.

Comme toute technologie, on peut aussi imaginer d'autres utilisations moins nobles : processeurs vérouillés sur un système d'exploitation, crypto-mouchards indéchiffrables, virus exécutés de manière... sécurisée par des logiciels troués, etc. Le tout garanti par le fabricant du processeur. Pourra-t-on lui faire confiance ? Qu'est-ce qui prouvera que les services secrets d'une grande puissance n'auront pas installé une porte dérobée dans le matériel ? Peut-être sera-ce l'occasion de développer des processeurs libres au même titre que les logiciels libres dans le contexte des logiciels de sécurité qui autorisent un contrôle plus facile du contenu par des pairs.

Restera le problème que le matériel doit être réalisé pour être utilisable et cette phase peut elle-même être piratée... On peut alors imaginer utiliser plusieurs processeurs réalisés par différents constructeurs de plusieurs pays et de les faire voter de manière sécurisée mais cela protègera des attaques actives mais pas de l'espionnage passif.

8. Remerciements

Nous remercions Mathieu CHEVRIER, élève de 3ème année en 1999–2000, pour sa première étude bibliographique sur la question. Merci aux relecteurs pour leurs remarques constructives et tout particulièrement pour la traduction de *replay* en rejeu au lieu de rejouage.

Bibliographie

1. Robert M. BEST. « Preventing Software Piracy with Crypto-Microprocessors ». Dans *Proc. IEEE Spring COMPCON'80*, pages 466–469, février 1980.
2. Manuel BLUM, William S. EVANS, Peter GEMMELL, Sampath KANNAN, et Moni NAOR. « Checking the Correctness of Memories ». Dans *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.
3. Manuel BLUM et Sampath KANNAN. « Designing programs that check their work ». Dans *STOC89*, pages 86–97, 1989.
4. Dallas Semiconductor. « *DS5002FP Secure Microprocessor Chip* », mai 1999. Récupérable par WWW à `http://www.dalsemi.com/DocControl/PDFs/5002fp.pdf`.
5. Blaise GASSEND, Dwaine CLARKE, G. Edward SUH, Marten van DIJK, et Srinivas DEVADAS. « Caches and Hash Trees for Efficient Memory Integrity Verification ». Dans *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, février 2003.
6. Tanguy GILMONT, Jean-Didier LEGAT, et Jean-Jacques QUISQUATER. « An Architecture of Security Management Unit for Safe Hosting of Multiple Agents ». Dans *Proceedings of COST 254*, pages 79–82, novembre 1998. `http://ldos.fe.uni-lj.si/cost254/papers/022.ps`.
7. Oded GOLDREICH et Rafail OSTROVSKY. « Software Protection and Simulation on Oblivious RAMs ». *Journal of the ACM*, 43(3):431–473, mai 1996.
8. Ronan KERYELL. « CryptoPage-1 : vers la fin du piratage informatique ? ». Dans *Symposium d'Architecture (SympA'6)*, pages 35–44, Besançon, juin 2000. `http://www.cri.ensmp.fr/~keryell/publications/ENSTBr_INFO_2000-001`.
9. Markus G. KUHN. « Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP ». *IEEE Transactions on Computers*, 47(10):1153–1157, octobre 1998.
10. Cédric LAURADOUX. « Les cryptoprocesseurs et la preuve de mémoire ». Diplôme d'étude approfondie, ENSTBr, septembre 2003. `http://www.lit.enstb.org/~keryell/eleves/ENSTBr/2002-2003/DEA/Lauradoux`.
11. David LIE, Chandramohan THEKKATH, John MITCHELL, et Mark HOROWITZ. « Specifying and Verifying Hardware for Tamper-Resistant Software ». Dans *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, mai 2003.
12. David LIE, Chandramohan A. THEKKATH, Mark MITCHELL, Patrick LINCOLN, Dan BONEH, John C. MITCHELL, et Mark HOROWITZ. « Architectural Support for Copy and Tamper Resistant Software ». Dans *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.
13. Serguei P. SKOROBOGATOV. « Low temperature data remanence in static RAM ». Rapport Technique, University of Cambridge Computer Laboratory, juin 2002.
14. Serguei P. SKOROBOGATOV et Ross ANDERSSON. « Optical Fault Induction Attacks ». Dans *Cryptographic Hardware and embedded Systems Workshop*, août 2002.
15. G. Edward SUH, Dwaine CLARKE, Blaise GASSEND, Marten van DIJK, et Srinivas DEVADAS. « The AEGIS Processor Architecture for Tamper-Evident and Tamper-Resistant Processing ». Rapport Technique, MIT Laboratory for Computer Science, février 2003.
16. « Trusted Computer Platform Alliance », 2002. `http://www.trustedpc.com`.
17. Willis H. WARE. « Security Controls for Computer Systems ». Rapport Technique, The Rand corporation, 1970. `http://www.rand.org/publications/R/R609.1/R609.1.html`.
18. Steve H. WEINGART. « Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defenses ». Dans *Cryptographic Hardware Embedded Sytems*, 2000.