CryptoPage

Une architecture efficace combinant chiffrement, intégrité mémoire et protection contre les fuites d'informations permettant du calcul distribué sûr

—

Version étendue

Guillaume Duc — Ronan Keryell

Institut TÉLÉCOM – TÉLÉCOM Bretagne
CS 83818
29238 Plouzané
France

{guillaume.duc,ronan.keryell}@telecom-bretagne.eu

RÉSUMÉ. *Durant ces dernières années, plusieurs architectures informatiques sécurisées ont été proposées. Elles chiffrent et vérifient le contenu de la mémoire afin de fournir un environnement d'exécution résistant aux attaques. Quelques architectures, comme notamment* HIDE*, ont aussi été proposées pour résoudre le problème de la fuite d'informations via le bus d'adresse du processeur. Cependant, malgré l'importance de ces mécanismes, aucune solution pratique combinant le chiffrement, la vérification de l'intégrité mémoire ainsi qu'une protection contre la fuite d'informations n'a encore été proposée, à un coût raisonnable en termes de performances. Dans cet article, nous proposons* CRYPTOPAGE*, une architecture qui implémente ces trois mécanismes avec un impact faible sur les performances (de l'ordre de 3 %). Nous présentons enfin un exemple d'utilisation pour construire des grilles de calcul sécurisées.*

ABSTRACT. *Several secure computing hardware architectures using memory encryption and memory integrity checkers have been proposed during the past few years to provide applications with a tamper resistant environment. Some solutions, such as* HIDE*, have also been proposed to solve the problem of information leakage on the address bus. We propose the* CRYPTOPAGE *architecture which implements memory encryption, memory integrity protection checking and information leakage protection together with a low performance penalty (3 % slowdown on average) by combining the Counter Mode of operation, local authentication values and Merkle trees. We end with the description of secure computation grids that would take benefit from this kind of secure processors.*

RSTI - TSI – 27/2008. Architecture des ordinateurs, pages 779 à 814

MOTS-CLÉS : *informatique sécurisée, chiffrement mémoire, intégrité mémoire*
KEYWORDS: *computer security, cryptoprocessor, cryptography,* MERKLE *tree, replay attack.*

1. Introduction

De nombreuses applications informatiques nécessitent un certain niveau de sécurité qui est hors de portée des architectures actuelles. Bien sûr, de nombreux algorithmes cryptographiques, des protocoles, des applications et des systèmes d'exploitation sécurisés existent, mais ils reposent tous sur une hypothèse forte : le matériel sous-jacent doit lui-même être sécurisé. Or cette hypothèse critique n'est jamais vérifiée, excepté pour de petites applications pouvant loger sur des cartes à puce par exemple.

Un moyen de sécurisation purement logiciel est d'obscurcir les programmes dans le but de rendre incompréhensible leur fonctionnement ainsi que les données manipulées (Collberg et Thomborson, 2002). Le programme est transformé au niveau du code source ou même parfois au niveau du binaire et en général cela ralentit l'exécution même si cela n'empêche pas à quelqu'un de très motivé de faire de la rétroingénierie.

Pour contourner ces limitations, durant ces dernières années, plusieurs architectures (comme XOM (Lie *et al.*, 2000; Lie *et al.*, 2003; Lie, 2004), AEGIS (Suh *et al.*, 2003b; Suh *et al.*, 2005) et CRYPTOPAGE (Lauradoux et Keryell, 2003; Duc *et al.*, 2005; Duc, 2004)) ont été proposées pour fournir aux applications un environnement d'exécution sécurisé. Ces architectures utilisent des mécanismes de chiffrement et de protection mémoire pour empêcher un attaquant de perturber le bon fonctionnement d'un processus sécurisé, ou l'empêcher d'obtenir des informations sur le code ou les données de celui-ci. Elles essaient de prévenir des attaques physiques contre les composants de l'ordinateur (par exemple, la X-BOX, la console de jeu de Microsoft, a été attaquée dans (Huang, 2002) par l'analyse des données transitant sur le bus de son processeur) ou des attaques logiques (comme par exemple un administrateur malveillant qui essaierait de voler ou de modifier le code ou les données d'un processus).

De telles architectures sécurisées peuvent être utiles dans de nombreux domaines comme par exemple le calcul distribué. Actuellement, des entreprises ou des centres de recherche peuvent hésiter à utiliser la puissance de calcul fournie par des ordinateurs d'une tierce partie car ils ne les contrôlent pas. En effet, les propriétaires de ces nœuds peuvent voler ou modifier les algorithmes ou les résultats de l'application distribuée. En revanche, si chaque nœud de la grille utilisait un processeur sécurisé qui garantit l'intégrité et la confidentialité de l'application et de ses résultats, ce problème de sécurité disparaîtrait.

Cependant, dans ces propositions d'architectures sécurisées, le bus d'adresse n'est pas ou peu modifié, et donc, les motifs d'accès à la mémoire sont visibles par un attaquant. La connaissance de ces motifs d'accès peut être suffisante pour identifier certains algorithmes utilisés (Zhuang *et al.*, 2004), et donc, pour obtenir de l'information sur le code de l'application sécurisée, malgré la présence du chiffrement. Afin de résoudre ce problème, ils présentent HIDE (*Hardware-support for leakage-Immune Dynamic Execution*), une infrastructure permettant de se protéger efficacement contre ces fuites d'informations sur le bus d'adresse (Zhuang *et al.*, 2004). Cependant, l'intégration du chiffrement et de la vérification mémoire n'a pas été étudiée.

Dans cet article, nous proposons CRYPTOPAGE, une extension de l'infrastructure HIDE pour fournir, en plus de la protection contre les fuites d'informations, un mécanisme de chiffrement et de vérification de l'intégrité de la mémoire avec un faible coût en termes de performances et ce sans hypothèse sur le système d'exploitation. Nous décrirons aussi comment un système d'exploitation, qui n'a pas besoin d'être de confiance, peut prendre part à certains mécanismes de sécurité, afin d'en réduire le coût, sans compromettre la sécurité de l'ensemble de l'architecture.

Le reste de l'article s'organise comme suit : la section 2 décrit notre proposition d'implémentation du chiffrement et de la vérification mémoire au-dessus de l'infrastructure HIDE ; la section 3 présente les résultats en termes de performances de ce système, la section 4 montre un exemple d'utilisation de CRYPTOPAGE pour construire des grilles de calcul distribué sûr et la section 5 présente les autres travaux menés dans ce domaine.

2. Architecture

Dans cette section, nous présentons tout d'abord les objectifs en termes de sécurité de notre architecture. Nous résumons ensuite les concepts-clés de l'infrastructure HIDE dont nous avons besoin ensuite pour présenter notre architecture CRYPTOPAGE.

2.1. *Objectifs de l'architecture et modèle de sécurité*

L'objectif de notre architecture est de permettre l'exécution de processus sécurisés matériellement. Elle doit principalement garantir à ces processus les deux propriétés suivantes :

– *confidentialité* : un attaquant doit pouvoir obtenir le moins d'information possible sur le code ou les données manipulées par un processus sécurisé ;

– *intégrité* : l'exécution correcte d'un processus sécurisé ne doit pas pouvoir être altérée par une attaque. En cas d'attaque, le processeur doit interrompre l'exécution du processus.

Le processeur, de type généraliste à hautes performances, doit pouvoir exécuter en parallèle des processus sécurisés et des processus normaux. Le système d'exploitation, standard, n'a pas besoin d'être sécurisé, ni même de confiance et peut même être compromis, voire usurpé. C'est une hypothèse forte de notre approche par rapport à d'autres projets : on ne fait pas confiance au système d'exploitation, car, comme il s'agit d'un système d'exploitation généraliste, il est gros (des millions de lignes de code auxquelles se rajoutent toutes les applications annexes) et impossible à prouver sans faille de sécurité [1].

1. Certes parce qu'il y a des trous de sécurité, mais aussi parce que c'est hors de portée des preuves faisables en sécurité.

Définition 1 *Un* processus sécurisé *est un processus dont l'exécution est protégée contre toute action physique ou logique extérieure au processeur et toute action logique interne au processeur. Les espaces mémoire extérieurs au processeur (données, instructions) sont chiffrés et protégés contre les modifications et le bus d'adresse est partiellement randomisé pour éviter des fuites d'information confidentielle et les intrusions logiques.*

Définition 2 *L'*exécution sécurisée *d'un processus sécurisé ne peut être que correcte (le processus n'a pas subi d'attaque par modification de son état jusqu'à présent) ou avortée (une attaque active a été détectée et tout état interne concernant le processus est détruit).*

Définition 3 *Un* attaquant *est*

– soit un programme concurrent (processus sécurisé ou pas, système d'exploitation avec tous les privilèges) pouvant espionner ou exécuter des instructions quelconques pour manipuler les états internes du processeur (registres, caches...) ou l'extérieur (mémoire, périphériques...) ;

– soit un humain utilisant des moyens physiques ou logiques pour corrompre ou espionner tout ce qui est extérieur au processeur.

Un attaquant peut être passif (espionnage) ou actif (manipulateur).

Nous considérons que tout ce qui est à l'extérieur du circuit intégré contenant le processeur (comme le bus mémoire, les unités de stockage de masse, le système d'exploitation, etc.) peut être sous le contrôle total d'un attaquant. Il peut, par exemple, injecter des données erronées en mémoire, modifier le comportement du système d'exploitation, surveiller le bus du processeur, etc.

Cependant, l'attaquant ne peut pas accéder, directement ou indirectement, à tout ce qui se trouve à l'intérieur du processeur. L'attaquant ne doit pas pouvoir accéder à des données en relation avec des processus dans le mode d'exécution protégée : registres internes d'un processus sécurisé, caches de données ou instructions, compteurs de débogages ou de statistiques, bus internes, etc.

En particulier, nous ne considérons pas les attaques par canaux cachés comme les attaques temporelles (Kocher, 1996) (et leurs cas particuliers comme les délais d'accès à des caches partagés ou des prédicteurs de branchement), les attaques par mesure de la consommation électrique (DPA (Kocher *et al.*, 1999)), etc. Tous ses aspects importants doivent être évidemment considérés d'un point de vue matériel et logiciel par un architecte global prenant en compte la sécurité (Molnar *et al.*, 2005) mais sont en dehors de l'objet précis de cet article.

Sous ces hypothèses, le seul moyen d'injecter des choses à l'intérieur du processeur est de modifier l'état extérieur (mémoire, bus...) lors du fonctionnement du processeur..

De plus, nous ne considérons pas les attaques par déni de service car elles sont inévitables (un attaquant peut choisir de ne pas alimenter le processeur...). L'attaquant peut également modifier l'exécution des appels système mais nous considérons ce type d'attaque comme un déni de service et nous pensons que ce problème doit être pris en compte au niveau de l'application elle-même. Celle-ci doit contenir des fonctions, qui seront exécutées de manière sûre par l'architecture, pour vérifier la cohérence des actions réalisées par le système d'exploitation afin de détecter ces attaques. Par exemple si un processus sécurisé écrit dans un fichier, il peut stocker une signature de ces données dans sa mémoire protégée par l'architecture et lors de la relecture des données, vérifier que la signature correspond bien.

L'architecture propose aussi une petite zone de stockage persistant interne au processeur qui permet à un processus de construire des moyens de stockages protégés et restreints, pour se protéger des attaques par rejeu global. Ces aspects ne seront pas décrits ici mais on peut se référer à (Duc, 2007).

Un exécutable est chiffré lors de l'édition des liens par le fabriquant ou l'éditeur du programme avec un chiffrement à clé symétrique pour des raisons de performances. Afin que l'exécutable ne puisse fonctionner que sur un processeur donné, pour que personne ne puisse récupérer la clé symétrique et par voie de conséquences toutes les informations concernant le processus, la clé symétrique est chiffrée avec un algorithme asymétrique avec la clé publique du processeur. Ces informations ne sont donc accessibles ensuite que par le processeur qui possède la clé privée que l'on considère inattaquable dans cet article. On suppose que le fabriquant du processeur oublie la clé privée après fabrication [2] ou que des mécanismes permettant de former ce couple de clés après fabrication sans divulgation possible de la clé privée sont mis en place.

Avec un tel système, un (gros) logiciel peut être diffusé largement de manière chiffrée avec une clé symétrique. Pour l'utiliser, il faudra juste récupérer un (petit) descripteur chiffré avec la clé publique du processeur cible et contenant la clé symétrique qui permettra de lancer le programme sur le processeur.

On suppose qu'un éditeur de programme ne chiffrera pas un programme pour un faux processeur avec une fausse clé publique car cela permettrait à quiconque fournissant une clé publique dont il connaît la clé secrète associée de récupérer le descripteur d'un programme dont la clé symétrique permettant de le lire. Cela implique par exemple la mise en place de certificats identifiant les vrais processeurs, une infrastructure de gestion de clés publiques (PKI), etc. Pour plus d'information on consultera (Duc, 2007).

Puisque un attaquant a le droit de construire un programme sécurisé pour un processeur sécurisé donné et l'exécuter, il faut que le système résiste aux attaques à texte clair choisi (*indistinguishability under chosen plaintext attack*, IND-CPA) (Goldwasser et Micali, 1984), c'est-à-dire qu'un attaquant ne puisse pas récupérer des clés de chif-

2. Cela pose clairement un problème de confiance, surtout avec des processeurs fabriqués dans des usines hors de contrôle.

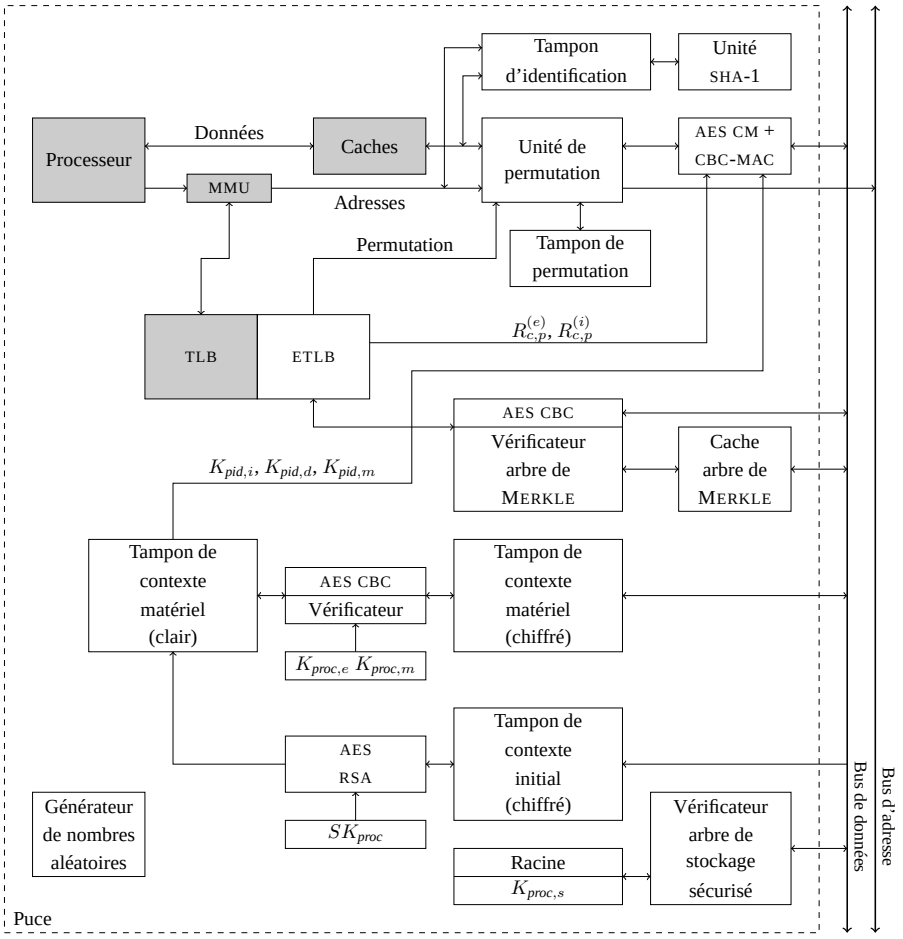


Figure 1. *Architecture globale résumée de* CRYPTOPAGE

frement spécifiques au processeur en compilant des binaires pour le processeur à attaquer.

Comme on veut garantir aussi l'intégrité des processus sécurisés, il faut aussi que les systèmes de chiffrement soient non malléables (Bellare et Sahai, 2006) pour éviter qu'une modification dans l'état externe d'un processus sécurisé soit encore un processus sécurisé, suffisamment valide pour continuer un certain temps son exécution.

Dans la suite nous allons détailler quelques aspects de notre architecture qui est résumée sur la figure 1.

2.2. *L'infrastructure* HIDE

Notre proposition est partiellement basée sur l'infrastructure HIDE que nous décrivons brièvement ici. Cette infrastructure, présentée dans (Zhuang *et al.*, 2004), garde en mémoire la séquence des adresses accédées par le processeur et permute l'espace mémoire avant qu'une adresse ne soit accédée de nouveau. Plus précisément, l'espace mémoire à protéger est divisé en blocs. La protection est réalisée en modifiant le comportement du cache du processeur. Lorsqu'une ligne est lue depuis la mémoire (lors d'un défaut de cache), elle est stockée dans le cache, comme normalement, mais est également verrouillée. Tant qu'une ligne est verrouillée, elle ne peut pas sortir du cache. Quand le cache est plein, HIDE réalise une permutation des adresses d'un bloc.

Durant cette permutation, toutes les lignes appartenant au bloc sont lues (depuis la mémoire ou depuis le cache) et stockées dans un tampon dédié, puis les adresses internes de ce bloc sont permutées et enfin, toutes les lignes du bloc sont déverrouillées et rechiffrées. Ainsi, entre chaque permutation, une ligne donnée n'est écrite et lue qu'une seule fois en mémoire. De plus, le rechiffrement des lignes empêche un attaquant de deviner la nouvelle adresse d'une ligne après la permutation. Avec ce mécanisme, un attaquant ne peut pas savoir qu'une ligne donnée est lue ou écrite plus souvent qu'une autre.

Afin de réduire le coût des permutations, (Zhuang *et al.*, 2004) proposent de les réaliser en tâche de fond avant le remplissage complet du cache. Avec ce mécanisme, l'impact sur les performances est négligeable (1,3 % selon (Zhuang *et al.*, 2004)).

2.3. *Implémentation du chiffrement et de la vérification mémoire*

Nous allons à présent décrire notre proposition permettant d'implémenter un mécanisme de chiffrement et de protection de l'intégrité mémoire à un faible coût au-dessus de l'infrastructure HIDE. Dans le reste de cette section, nous supposons que les blocs protégés par HIDE sont confondus avec les pages du système de gestion de la mémoire virtuelle afin de simplifier les explications (mais ce n'est pas une obligation). Dans la suite, pour simplifier, la notion de page sera utilisée pour parler de page du point de vue gestion de mémoire virtuelle et de HIDE. La notion de bloc sera gardée pour décrire des morceaux de données manipulés, comme dans les algorithmes de chiffrement. Nous utiliserons également les notations suivantes :

– $\|$ représente la concaténation de deux chaînes de bits et $\oplus$ l'opération *ou exclusif* bit-à-bit (XOR) ;

– $L_{a,c} = L_{a,c}^{(0)} \| \ldots \| L_{a,c}^{(l-1)}$: le contenu de la ligne de cache numéro a dans la page mémoire c, divisée en l blocs de même taille que celle des blocs utilisés par l'algorithme de chiffrement ;

– $E_K(D)$: le résultat du chiffrement du bloc de données D avec la clé symétrique K;

– K_e et K_m: les clés symétriques utilisées pour chiffrer et déchiffrer le code et les données du processus sécurisé (K_e) et pour calculer le code d'authentification de message (MAC, *Message Authentication Codes*) utilisé pour authentifier le code et les données du processus (K_m). Ces deux clés sont spécifiques à un processus sécurisé donné et sont stockées de manière sécurisée (au sens protégées selon l'état de l'art contre les attaques physiques, l'exploitation de canaux cachés) dans le contexte matériel [3] de ce dernier. Le contexte matériel doit donc être chiffré de manière forte dès qu'il est en dehors du processeur.

Premièrement, nous allons décrire comment le chiffrement et la vérification sont effectués au niveau d'une ligne de cache puis comment les informations sur les pages sont protégées.

2.3.1. *Chiffrement et vérification des lignes de cache*

Durant chaque permutation, le processeur choisit aléatoirement deux nombres, $R_{c,p}$ et $R'_{c,p}$ (où c est le numéro de la page et p le numéro de la permutation), et les stocke avec les autres informations liées à la page (comme par exemple la table de permutation utilisée par HIDE).

Après une permutation, quand une ligne de cache est réécrite en mémoire, le processeur la chiffre, calcule un MAC et stocke en mémoire la concaténation $C_{a,c} \| H_{a,c}$ de la ligne chiffrée $C_{a,c}$ et du code d'authentification $H_{a,c}$ avec :

$$C_{a,c} = C_{a,c}^{(0)} \| C_{a,c}^{(1)} \| \cdots \| C_{a,c}^{(l-1)} \quad [1]$$

$$C_{a,c}^{(i)} = L_{a,c}^{(i)} \oplus PAD_{a,c}^{(i)} \quad [2]$$

$$PAD_{a,c}^{(i)} = E_{K_e}(R'_{c,p} \| a \| i) \quad [3]$$

$$H_{a,c} = H_{a,c}^{(l)} \quad [4]$$

$$H_{a,c}^{(i)} = E_{K_m}(C_{a,c}^{(i-1)} \oplus H_{a,c}^{(i-1)}), \quad i \in [1, l-1] \quad [5]$$

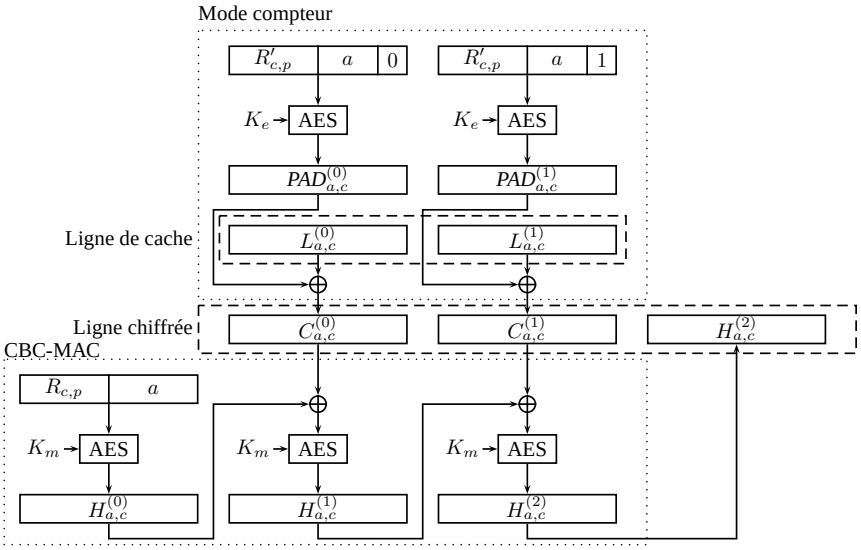
$$H_{a,c}^{(0)} = E_{K_m}(R_{c,p} \| a) \quad [6]$$

Une ligne de cache est chiffrée en utilisant le mode compteur (NIST, 2001) (équation [2]). Les masques (*pads*) $PAD_{a,c}$ utilisés dépendent de $R'_{c,p}$ et de a (équation [3]). Les équations de 4 à 6 définissent un CBC-MAC (NIST, 1985) [4] calculé sur la ligne de cache chiffrée, $R_{c,p}$ et l'adresse a de la ligne de cache. Ce mécanisme de chiffrement est résumé par la figure 2a.

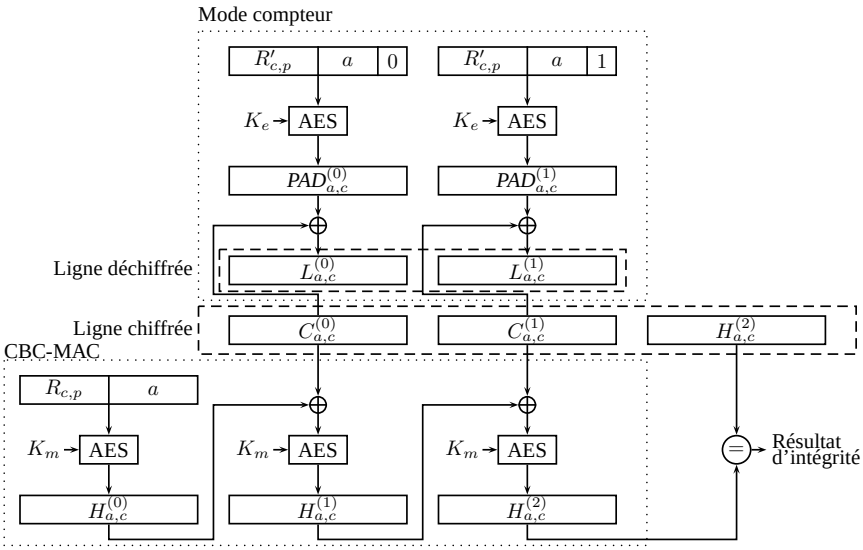
---

3. Le contexte matériel comprend le contenu des registres nécessaires à faire tourner un processus. Il peut contenir des registres cachés de l'utilisateur normal concernant des états internes du processeur, liés à des modes sécurisés, des états du pipeline. Pour stopper et relancer un processus, le système d'exploitation doit manipuler ces contextes matériels.

4. Nous utilisons un CBC-MAC car c'est un algorithme relativement rapide qui permet aussi d'utiliser le même matériel pour le chiffrement et la protection de l'intégrité, mais tout autre bon MAC pourrait convenir.



(a) Chiffrement



(b) Déchiffrement

Figure 2. *Les opérations de chiffrement et de déchiffrement d'une ligne de cache*

Le mode compteur est sûr (Bellare *et al.*, 1997) à condition que le compteur ne soit utilisé qu'une seule fois avec la même clé [5]. Entre deux permutations, la ligne de cache a n'est chiffrée qu'une seule fois [6] et donc le masque $E_{K_e}(R'_{c,p}\|a\|i)$ n'est utilisé qu'une seule fois, sauf si le même nombre aléatoire $R'_{c,p}$ est choisi durant deux permutations différentes.

Si $R'_{c,p}$ fait par exemple 119 bits [7], le paradoxe des anniversaires nous indique que la probabilité de collision est élevée ($> 1/2$) après $\sqrt{2^{119}} = 2^{59.5}$ tirages aléatoires. Donc après $2^{59.5}$ permutations, le risque d'une collision, et donc le risque d'utiliser deux fois le même masque est élevé. Cependant, même si le processeur exécutait une permutation par cycle d'horloge à 1 GHz, il faudrait 25 années en moyenne pour parvenir à une collision, donc ce point n'est pas critique au niveau sécurité. De plus, afin de monter cette attaque, un adversaire devrait pouvoir identifier une collision. Or ce n'est pas possible car $R'_{c,p}$ est chiffré avec les autres informations sur la page.

Dans notre proposition, la protection de l'intégrité (assurée par un MAC) est appliquée sur les données chiffrées. Ce mécanisme est connu sous le nom de *Encrypt-Then-MAC* dans la littérature. Il a été montré que cette construction n'affaiblit pas la confidentialité des données chiffrées et que, si le MAC utilisé est suffisamment robuste, elle garantit l'intégrité des données (Bellare et Namprempre, 2000).

Avant de pouvoir utiliser une ligne de cache lue depuis la mémoire, le processeur doit la déchiffrer et la vérifier. Pour la déchiffrer, le processeur calcule les masques nécessaires et réalise un XOR entre le contenu chiffré de la ligne et ces masques (l'opération inverse de l'équation [2]). Les masques peuvent être calculés en parallèle à l'accès mémoire car ils ne dépendent que de $R'_{c,p}$, a et K_e qui sont déjà disponibles. Si le temps nécessaire au calcul des masques est inférieur à la latence mémoire, les masques sont prêts avant l'arrivée des données chiffrées et donc la fin de l'opération de déchiffrement est réalisée au plus en un cycle (le XOR, qui peut être transparent s'il n'est pas dans le chemin critique du pipeline). Pour vérifier l'intégrité de la ligne, le processeur calcule le MAC (équations [4] à [6]) sur les données chiffrées, et le compare avec la valeur $H_{a,c}$ lue depuis la mémoire. S'ils sont identiques, la ligne n'a pas été corrompue. Ce mécanisme de déchiffrement et de vérification d'intégrité est résumé par la figure 2b.

Ce mécanisme permet d'empêcher trois types d'attaques. Premièrement, un attaquant ne peut pas modifier une valeur en mémoire car il devrait calculer le MAC correct pour celle-ci, ce qui n'est pas possible puisqu'il ne connaît pas la clé K_m. De plus,

---

5. En effet, si deux lignes $L_{a,c}$ et $L'_{a,c}$ sont chiffrées avec le même masque $PAD_{a,c}$, on obtient la relation $C_{a,c} \oplus C'_{a,c} = (L_{a,c} \oplus PAD_{a,c}) \oplus (L'_{a,c} \oplus PAD_{a,c}) = L_{a,c} \oplus L'_{a,c}$ et donc on peut obtenir des informations sur le contenu des deux lignes en comparant simplement les deux lignes chiffrées.

6. En effet, avec HIDE, entre deux permutations, une ligne est lue et écrite au plus une fois en mémoire.

7. C'est notamment le cas si l'on utilise les paramètres suivants : l'algorithme de chiffrement est AES qui utilise des blocs de 128 bits, lignes de cache de 32 octets (256 bits), donc i est réduit à un seul bit, pages de 8 ko, donc a fait 8 bits.

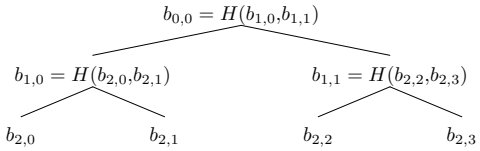


Figure 3. *Arbre de* MERKLE

un attaquant ne peut pas copier une ligne et son MAC associé à un autre endroit en mémoire car le MAC dépend de l'adresse virtuelle de la ligne. Enfin, si un attaquant ne peut pas rejouer $R_{c,p}$, il n'est pas en mesure de monter une attaque par rejeu [8] car le MAC dépend de $R_{c,p}$ dont la valeur change à chaque permutation.

2.3.2. *Protection des informations sur les pages*

Ainsi, pour empêcher les attaques par rejeu, on doit protéger $R_{c,p}$ contre le rejeu. Pour se faire, on protège les structures de données contenant les informations sur les différentes pages ($R_{c,p}$, $R'_{c,p}$, la table de permutation, etc.) à l'aide d'un arbre de MERKLE (Merkle, 1989).

Le principe est de construire un arbre de hachage dont les feuilles sont ces structures de données à protéger. Chaque nœud de l'arbre contient un résumé cryptographique calculé sur le contenu de ses nœuds fils. La racine de l'arbre est stockée dans une zone mémoire sécurisée à l'intérieur même du processeur et ne peut donc pas être altérée ni rejouée. Quand le processeur met à jour les informations sur une page (à la suite d'une permutation par exemple), il met à jour le contenu des nœuds situés sur le chemin entre la feuille modifiée et la racine. Quand le processeur lit les informations sur une page (en cas de défaut de TLB par exemple), il vérifie le contenu des nœuds jusqu'à la racine.

Si l'on veut protéger une zone de n (où n est une puissance de 2) éléments (dans notre cas ces éléments sont les informations sur les pages) représentés par $b_{\log_2 n,0}$ à $b_{\log_2 n,n-1}$ (voir figure 3), les algorithmes utilisés pour effectuer une lecture vérifiée ($\mathcal{R}_{\mathcal{V}}(i,j)$) ou une écriture vérifiée ($\mathcal{W}_{\mathcal{V}}(b_{i,j},i,j)$) sont donnés par les algorithmes 1 et 2 (Lauradoux et Keryell, 2003; Duc *et al.*, 2005), où H est une fonction de hachage à sens unique, $\mathcal{R}(i,j)$ la fonction qui retourne la valeur du nœud $b_{i,j}$ depuis la mémoire (excepté pour $b_{0,0}$ qui est stocké dans une mémoire sécurisée), $\mathcal{W}(b_{i,j},i,j)$ la fonction qui écrit la valeur du nœud $b_{i,j}$ en mémoire (excepté pour $b_{0,0}$).

Cet arbre de MERKLE permet d'empêcher un attaquant de rejouer les informations sur les pages. Afin de réduire le temps nécessaire pour effectuer la vérification de l'arbre de MERKLE durant un défaut de TLB, on peut utiliser un petit cache spécialisé dans le stockage de quelques nœuds de l'arbre. Pendant l'opération de vérification de

8. Une attaque par rejeu consiste à sauvegarder une valeur et son MAC et à les replacer à la même adresse ultérieurement.

$\mathcal{R}_{\mathcal{V}}(i,j)$:
  $b_{i,j} = \mathcal{R}(i,j)$
  tant que $i > 0$:
    $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$
    $b_{i,f} = \mathcal{R}(i,f)$ { *Lecture du frère* }
    $b_{i-1,p} = \mathcal{R}(i-1,p)$ { *Lecture du père* }
    si $b_{i-1,p} \neq H(b_{i,\min(j,f)},b_{i,\max(j,f)})$:
      erreur
    $i = i - 1$; $j = p$ { *On remonte* }
  renvoie $b_{i,j}$

Algorithme 1. *Lecture vérifiée*

$\mathcal{W}_{\mathcal{V}}(b_{i,j},i,j)$:
  si $i > 0$:
    $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$
    $b_{i,f} = \mathcal{R}_{\mathcal{V}}(i,f)$ { *Lecture et vérification du frère* }
    $b_{i-1,p} = H(b_{i,\min(j,f)},b_{i,\max(j,f)})$
    $\mathcal{W}_{\mathcal{V}}(b_{i-1,p},i-1,p)$ { *Écriture et vérification du père* }
  $\mathcal{W}(b_{i,j},i,j)$ { *Écriture du nœud* }

Algorithme 2. *Écriture vérifiée*

l'arbre, le processeur peut s'arrêter dès qu'un des nœuds est présent dans ce cache. En effet, pour être présent dans ce cache, un nœud doit nécessairement avoir été vérifié durant une opération de vérification précédente et de plus, ce cache étant situé dans le processeur, il est, par hypothèse, inaltérable, et donc le nœud est forcement correct.

Les structures de données concernant la sécurité des informations sur les pages mémoire doivent également être chiffrées et authentifiées afin d'empêcher un attaquant d'accéder ou modifier la table de permutation, ce qui rendrait inopérant le mécanisme de protection contre les fuites d'informations. Ces structures sont donc chiffrées à l'aide d'un algorithme de chiffrement symétrique (par exemple AES utilisé en mode CBC avec un vecteur d'initialisation (IV) aléatoire) et avec la clé secrète K_p connue uniquement par le processeur.

Les données concernant la traduction d'adresse étant gérées par le système d'exploitation, elles restent en clair sans mettre en péril la sécurité des données utilisateur. En effet, si le système d'exploitation veut changer les pages mémoire d'un processus sécurisé dans son dos, le $R_{c,p}$ associé à la page et stocké de manière sécurisée dans le processeur est utilisé pour authentifier *via* le MAC les données de la page avec l'adresse virtuelle [9] et permettra de détecter que les données sont incorrectes.

9. Et non physique, car cela permettrait de faire une attaque par permutation sur les adresses virtuelles vers des données physiques authentifiées. En outre cela empêcherait le système d'ex-

2.4. *Gestion des données d'authentification*

Comme nous l'avons vu dans la section 2.3.1, pour chaque ligne de cache ($C_{a,c}$) stockée en mémoire, une valeur d'authentification ($H_{a,c}$) est également stockée.

Afin de ne pas modifier la gestion de la mémoire virtuelle par le système d'exploitation ni par les processus sécurisés, l'unité de gestion de la mémoire du processeur (MMU) et les fonctions d'allocation mémoire de la bibliothèque standard sont modifiées. Quand un processus sécurisé alloue de la mémoire, la bibliothèque demande au système d'exploitation de la mémoire supplémentaire afin de stocker les valeurs d'authentification. De plus, quand un processus sécurisé accède à la mémoire, la MMU modifie automatiquement l'adresse logique demandée afin de prendre en compte la présence de ces valeurs d'authentification. Avec ce mécanisme, la taille des pages mémoire manipulées par le système d'exploitation n'est pas modifiée et ce dernier les manipule sans avoir besoin de savoir ce qu'elles contiennent (des données en clair ou des données chiffrées mélangées avec des valeurs d'authentification. De plus, les processus sécurisés continuent de croire que la mémoire qu'ils manipulent est contiguë malgré la présence de ces valeurs d'authentification (Duc *et al.*, 2005).

2.5. *Gestion déléguée de l'arbre de* MERKLE

Dans la section 2.3.2, nous avons vu que le processeur doit gérer un arbre de MERKLE couvrant l'ensemble des structures de données contenant les informations sur les pages mémoire. Il doit également implémenter les algorithmes de lecture et d'écriture afin de vérifier l'intégrité de ces structures.

Premièrement, toutes les pages de l'espace mémoire d'un processus ne sont généralement pas toutes utilisées, ce qui signifie que de nombreuses branches de l'arbre sont vides. Afin de réduire la place mémoire nécessaire afin de stocker l'arbre de MERKLE, ces branches sont tronquées.

De plus, les algorithmes de lecture et d'écriture peuvent être implémentés directement par le processeur en matériel ou en logiciel, *via* le système d'exploitation par exemple. L'implémentation en matériel, bien que plus sécurisée, est difficile à cause de la complexité des algorithmes et implique un stockage relativement simple et peu optimisé de l'arbre en mémoire.

2.5.1. *Implémentation en matériel*

Toutes les opérations requises pour manipuler l'arbre de MERKLE peuvent être implémentées en matériel ou en microcode. Bien que cette solution offre un haut niveau de sécurité car c'est le processeur, qui est supposé sécurisé et inattaquable physiquement, qui gère tous les aspects de la protection mémoire, elle présente plusieurs

ploitation de bouger les pages en mémoire physique alors que son travail est justement de gérer correctement la mémoire physique.

inconvénients. Premièrement, la méthode de stockage de l'arbre en mémoire doit être relativement simple puisque le processeur lui-même doit calculer l'adresse des différents nœuds de l'arbre. De plus, comme les algorithmes de vérification sont relativement complexes, l'implémentation en matériel ou en microcode est difficile. Donc le meilleur choix est de déléguer une partie de ces opérations au système d'exploitation à condition de ne pas rajouter de failles de sécurité.

2.5.2. *Implémentation en logiciel*

Afin de réduire les modifications matérielles, on veut pouvoir déléguer une partie des opérations de chargement sécurisé des informations sur les pages mémoire *LoadPageInfo*(p) ainsi que le stockage de l'arbre de MERKLE au système d'exploitation sans que ce dernier puisse compromettre la sécurité.

Le matériel fournit au système d'exploitation deux nouvelles instructions de sécurisation, réservées en plus au mode superviseur : `LoadNode` et `HashCheck`, ainsi qu'un tampon spécial appelé tampon de vérification (VB, *verification buffer*) qui peut contenir exactement $n - 1$ couples de nœuds de l'arbre (où $n = \log_2($*nombre maximum de pages*$)$).

La première instruction, `LoadNode` prend en entrée trois paramètres : l'adresse physique où est stocké le nœud $b_{i,j}$, la profondeur i de ce nœud dans l'arbre et sa position horizontale j. Cette instruction vérifie tout d'abord si le nœud $b_{i,j}$ est déjà disponible dans le cache d'arbre de MERKLE. Dans ce cas, elle positionne un drapeau dans les registres de contrôle du processeur pour informer le système d'exploitation. Sinon, l'instruction lit le nœud $b_{i,j}$ et son frère $b_{i,j\oplus 1}$ situés à l'adresse donnée en paramètre et les stocke ensemble à la ligne i du VB.

La seconde instruction, `HashCheck` prend un seul paramètre : le numéro de la ligne à vérifier dans le VB. Cette instruction calcule le résumé cryptographique des nœuds $b_{i,j}$ et $b_{i,j\oplus 1}$ stockés à la ligne i du VB, et compare le résultat avec le nœud père, $b_{i-1,\lfloor \frac{j}{2} \rfloor}$, qui doit être stocké dans le cache d'arbre de MERKLE (et non pas dans le VB). Si la comparaison réussit, les nœuds $b_{i,j}$ et $b_{i,j\oplus 1}$ sont corrects et donc l'instruction peut les copier dans le cache d'arbre de MERKLE. S'ils sont différents, ou si le nœud père n'est pas déjà dans le cache d'arbre de MERKLE, l'instruction déclenche une exception de sécurité et interrompt l'exécution du processus sécurisé.

Avec ces deux nouvelles instructions sécurisées et le VB, une partie des opérations nécessaires pour effectuer une lecture vérifiée peut être effectuée par le système d'exploitation. Quand le processeur a besoin des informations sur une page p qui n'est pas déjà disponible dans le TLB, le processeur génère une exception spéciale pour indiquer au système d'exploitation qu'il doit récupérer et faire vérifier $b_{n,p}$. Le système d'exploitation exécute alors l'algorithme 3. À la fin, il relance l'exécution du processus sécurisé et le processeur s'attend à trouver $b_{n,p}$ dans le cache d'arbre de MERKLE. Si le processeur ne trouve pas cette page, il y a double faute et arrêt immédiat du processus par le processeur. Cela signifie que la mémoire a été attaquée ou que le système d'exploitation n'a pas joué le jeu (déni de service).

LoadPageInfo(p) : { *Le processeur a besoin d'un* TLB *sécurisé d'une page mémoire* p }
{ *Cerne la branche de l'arbre à vérifier depuis le bas :* }
$d = n; q = p$
tant que $d > 0$:
$A_{d,q}$ = *getNodeAddress*(d,q) { *L'*OS *trouve l'adresse du nœud de l'arbre selon son bon plaisir* }
`LoadNode` $A_{d,q}$,d,q { *Demande au processeur de charger un nœud dans le* VB }
si $b_{d,q}$ est déjà dans le cache:
break { *On a trouvé un nœud déjà certifié en cache : on a cerné* }
$d = d - 1; q = \lfloor \frac{q}{2} \rfloor$ { *Remonte dans l'arbre* }
$d = d + 1$ { *Redescend sur le premier nœud qui manque* }

{ *Vérifie en descendant l'éventuelle branche manquante :* }
tant que $d \leq n$:
`HashCheck` d { *Demande au processeur de vérifier nœud dans* VB *et si correct bascule dans cache certifié. Si arrivé en bas déverrouille* TLB }
$d = d + 1$ { *Descend sur le nœud suivant qui manque* }
`ReturnFromInterrupt` { *Reprend l'exécution :*
- si attaque, le TLB *manquera* $\rightsquigarrow$ *double faute et dénis de service détecté*
- sinon le TLB *est chargé après déchiffrement et exécution normale* }

Algorithme 3. *Vérification déléguée de manière sécurisée à l'*OS *des données d'une page mémoire en utilisant 2 nouvelles instructions sécurisées*

L'algorithme 3 réalise les opérations suivantes. Premièrement, il charge, à l'aide de l'instruction `LoadNode`, les nœuds sur le chemin entre la feuille contenant les informations sur la page, et la racine de l'arbre. Dès qu'il essaie de charger un nœud déjà présent dans le cache d'arbre de MERKLE, il commence la vérification des nœuds, en commençant par le nœud manquant, c'est-à-dire celui qui suit celui où il s'est arrêté en rencontrant un nœud qui était dans le cache. Si d est la profondeur du dernier nœud absent dans le cache, l'algorithme exécute l'instruction `HashCheck` d qui vérifie que $H(b_{d,2i},b_{d,2i+1}) = b_{d-1,i}$. Si cette vérification réussit, les deux nœuds à la profondeur d sont corrects. L'instruction `HashCheck` peut donc déplacer ces nœuds du VB vers le cache d'arbre de MERKLE. L'algorithme effectue cette opération plusieurs fois jusqu'à la feuille de l'arbre. À la fin de l'algorithme, si aucune exception de sécurité n'a été levée, le nœud demandé par le processeur est situé dans le cache d'arbre de MERKLE et donc l'exécution du processus sécurisé peut reprendre, avant d'être interrompue pour absence du TLB demandé si le système d'exploitation a fait du déni de service.

Avec cette solution, le système d'exploitation peut choisir la meilleure stratégie de stockage et de gestion de l'arbre de MERKLE.

2.6. *Discussion sur la sécurité de la méthode*

Il ne s'agit pas de discuter en profondeur la sécurité de la méthode qui demanderait de faire des preuves de sécurité bien au-delà de nos moyens humains mais d'esquisser uniquement des aspects à considérer.

L'attaquant peut connaître le programme en clair qui est exécuté parce que c'est un programme connu (logiciel libre...) ou parce que lui-même l'a soumis.

Lors du chargement initial d'un exécutable chiffré, le processus déchiffre avec sa clé privée la clé symétrique du contexte d'exécution chiffré et vérifie que le contexte est intègre. Ensuite, avec la clé symétrique, il peut déchiffrer plus rapidement le reste du descripteur de processus chiffré. Dans ce descripteur on va récupérer d'autres clés qui serviront à déchiffrer d'autres clés aléatoires par page mémoire qui serviront à (dé)chiffrer les données des pages mémoire.

Si tous les algorithmes de (dé)chiffrement à clé secrète ou à clé publique utilisés résistent à des attaques à texte clair choisi, l'idée est que globalement tout l'édifice résistera à une attaque à texte clair choisi (Goldwasser et Micali, 1984). C'est-à-dire qu'un attaquant ne récupérera pas d'information sur les clés de chiffrement s'il construit son propre programme sécurisé qu'il fait exécuter. Le fait d'utiliser un algorithme de chiffrement probabiliste fait qu'un attaquant ne récupérera pas non plus d'information significative sur un processus qu'il ne connaît pas.

Le système est protégé contre les attaques contre les modifications des données en mémoire par un attaquant. Outre le fait que les algorithmes de chiffrements utilisés ont des propriétés de non-malléabilité (*non-malleability under adaptive chosen ciphertext attack*, NM-CCA2) (Bellare et Sahai, 2006), toutes les informations sont protégées par un code d'authentification qui empêche à un attaquant d'utiliser le processeur comme oracle de déchiffrement : chaque ligne de cache, les descripteurs de processus chiffrés, etc. sont protégés par un MAC de type AES CBC ou GCM. Les clés de chiffrement des pages sont chiffrées et protégées par un arbre de MERKLE, dont la racine est elle-même chiffrée dans le contexte du processus chiffré qui est lui-même authentifié par un AES CBC ou GCM. Cela conduit à avoir globalement une architecture assurant une résistance aux attaques.

Les attaques contre le rejeu de vieilles données authentifiées par le processeur sont empêchées par le chiffrement probabiliste et le fait qu'une donnée écrite à un endroit de la mémoire n'est jamais chiffrée de la même manière. Les clés de chiffrement étant elles-mêmes protégées par un arbre de MERKLE dont la racine est stockée de manière sécurisée dans le contexte d'exécution du processus chiffré. Pour rejouer des données, il faut être en phase avec cette racine, donc remettre tout le processus dans l'état correspondant à une sauvegarde précédente du descripteur de processus. Si on veut éviter le rejeu d'un processus globalement à partir d'un état passé, il suffit de rajouter dans le programme du processus une communication sécurisée avec un élément extérieur qui jouerait le rôle de notaire ou plus simplement la mémoire protégée permanente interne au processus à accès réservé à un processus.

Moins fort est le chiffrement des adresses. Certes la technique HIDE couplée à notre chiffrement probabiliste des lignes de cache randomise les adresses à l'intérieur d'une page mémoire et ce à chaque fois que des données doivent être sorties du cache interne vers la mémoire, mais les adresses de poids fort ne sont pas randomisées à chaque fois. Ce choix est fait afin de garder une compatibilité avec les systèmes d'exploitation qui reposent sur une hiérarchie mémoire orientée pages, vu que l'on vise les applications et processeurs généralistes, par opposition à des techniques plus générales mais irréalistes en termes de performance (Goldreich et Ostrovsky, 1996). On a donc une résistance contre la fuite d'information par les adresses à l'intérieur d'une page de mémoire mais pas au niveau des pages de mémoire. Il faut donc programmer de telle manière que les motifs d'accès aux pages ne créeront pas de fuites d'informations secrètes *via* le compteur de programme au sens du *Program Counter Security Model* (Molnar *et al.*, 2005) ou encore pire au niveau des adresses des données manipulées. Mais comme cette fuite n'a lieu qu'au niveau des numéros de pages, cela minimise la quantité d'information qui fuit, mais ce n'est pas à négliger comme canal caché.

On peut résoudre ce problème en rajoutant

– soit une couche logicielle telle que la méthode décrite dans (Molnar *et al.*, 2005) qui utilise des techniques d'*if-conversion*, utilisées sur certaines architectures pour supprimer les branchements (Keryell et Paris, 1993) et donc des fuites d'informations sur le bus d'adresse par le compteur de programme, mais au niveau d'une page ici ;

– soit en interne au processus des phases de mélange des données entre pages de mémoire.

Plusieurs autres mécanismes de sécurité nécessaires n'ont pas été décrits dans cet article, comme par exemple la protection du contexte matériel d'un processus durant une interruption ou durant l'exécution d'un autre processus, ou la création et le chargement d'un exécutable chiffré. Tous ces points sont décrits dans plusieurs autres publications (Suh *et al.*, 2003b; Lie *et al.*, 2003; Lie *et al.*, 2000; Duc, 2004; Zhuang *et al.*, 2004; Duc, 2007; Duc et Keryell, 2008). La sécurité de ces aspects n'est donc pas évoquée ici.

Évidemment tous ces aspects devraient être prouvés. Mais sur des architectures avancées comme celle-ci, cela n'a jamais encore été entrepris à notre connaissance et nous laissons ces preuves comme problèmes ouverts.

3. Évaluation et résultats

3.1. *Résultats*

Dans cette section, nous utilisons les mêmes paramètres architecturaux que dans (Zhuang *et al.*, 2004) pour évaluer l'impact de notre proposition. Ces paramètres sont résumés dans le tableau 1.

Paramètre architectural	Spécifications
Fréquence d'horloge	1 GHz
Taille d'une ligne de cache	256 bits
Bus mémoire	200 MHz, 8 octets de large
Latence mémoire	80 (premier), 5 (suivants) cycles
Page mémoire	8 ko
Algorithme de chiffrement	AES
Taille d'un bloc de chiffrement	128 bits (donc $l = 2$)
Latence de chiffrement	11 cycles

Tableau 1. *Paramètres architecturaux*

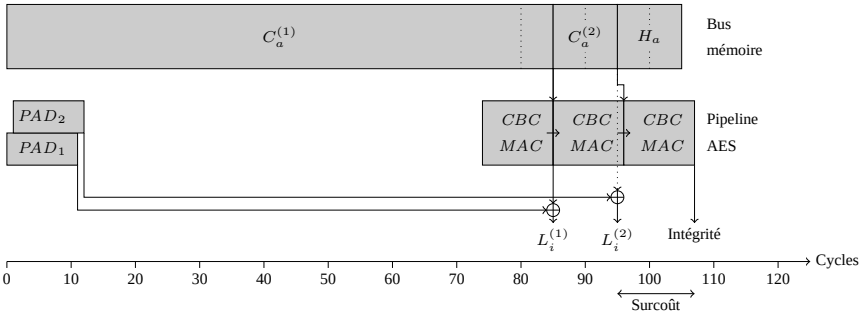


Figure 4. *Exemple du chargement d'une ligne de cache depuis la mémoire*

3.2. *Analyse théorique*

La figure 4 décrit la chronologie d'une opération de lecture dans le cas d'un défaut de cache, avec l'hypothèse que les informations sur la page mémoire sont déjà disponibles dans le TLB. Le surcoût lié au chiffrement et à la vérification d'intégrité de la ligne est de 13 % (107 cycles au lieu de 95 sans chiffrement ni vérification).

Cependant, le déchiffrement par lui-même est réalisé en un seul cycle. Une ligne de cache peut être utilisée de façon spéculative dès qu'elle est déchiffrée (avec dans ce cas une pénalité en termes de performances de seulement 1 %). Dans ce cas, le processeur doit vérifier qu'aucune action critique du point de vue de la sécurité (comme par exemple une écriture non chiffrée vers la mémoire) n'est effectuée avant d'obtenir le résultat de la vérification d'intégrité.

Durant un défaut de TLB, le processeur doit vérifier l'arbre de MERKLE qui protège les informations concernant la page mémoire demandée. Si un processus sécurisé utilisait entièrement son espace d'adressage de 32 bits, la profondeur de l'arbre de MERKLE serait de 19 et donc, dans le pire des cas, le processeur devrait calculer

19 résumés cryptographiques, ce qui représenterait 1 520 cycles [10]. Heureusement, cette opération n'est pas très fréquente. De plus, les valeurs des nœuds intermédiaires sont mises en cache dans le processeur afin d'accélérer les vérifications ultérieures.

Le processeur a également besoin de stocker $R_{c,p}$ et $R'_{c,p}$ avec les informations sur les pages. Ces deux nombres sont relativement courts (maximum 120 bits pour $R_{c,p}$ et 119 bits pour $R'_{c,p}$). Le surcoût en termes d'espace mémoire nécessaire pour stocker ces informations supplémentaires (y compris les informations pour HIDE comme la table de permutation) n'est que de 3,9 %.

3.3. *Évaluation par le biais de simulations*

3.3.1. *Programmes étalons*

Afin d'obtenir des résultats sur différents types d'applications, nous avons utilisé les programmes étalons de la suite SPEC CPU2000int (Henning, 2000; SPEC, 2007). Cette suite est composée de quatorze programmes étalons : `gzip`, `vpr`, `gcc`, `art`, `mcf`, `equake`, `crafty`, `parser`, `eon`, `perlbmk`, `gap`, `vortex`, `bzip2`, `twolf`. Parmi ces programmes, seuls dix ont pu être utilisés car `crafty` et `eon` ne compilaient pas avec la version des compilateurs C et C++ disponibles sur la machine ALPHA qui a été utilisée pour compiler ces applications [11], et car `art` et `equake` donnaient des sorties différentes des sorties normalement attendues.

Ces programmes se composent de 40 à 500 milliards d'instructions chacun. L'exécution d'un programme à l'aide de SIMPLESCALAR étant très lente, deux stratégies différentes ont été employées pour obtenir des résultats.

La première série de simulations a été réalisée en exécutant seulement une petite partie de chaque programme étalon à l'aide de SIMPLESCALAR afin d'obtenir plus rapidement des premiers résultats. La méthode consistait à exécuter rapidement le début du programme (1,5 milliards d'instructions) puis à simuler finement uniquement 200 millions d'instructions. L'exécution rapide du début du programme permet de passer les différentes phases d'initialisation de ce dernier. Cette méthode a déjà été utilisée dans la littérature, comme par exemple dans (Suh *et al.*, 2003a).

La seconde série de simulations a consisté à simuler l'exécution complète des programmes étalons dans les différentes configurations matérielles à tester. Donc les dix programmes étalons ont été simulés dans vingt-six configurations matérielles différentes, soit un total de deux cent soixante simulations. Comme chaque simulation dure plusieurs dizaines de jours sur un microprocesseur classique (*Intel Pentium IV*), il a fallu utiliser un nombre important d'ordinateurs pour les effectuer. Les détails de

10. Le calcul d'un résumé en utilisant l'algorithme SHA-1 prend 80 cycles.
11. Les binaires exécutés avec SIMPLESCALAR doivent avoir été compilés sur DIGITAL ALPHA OSF UNIX. Il n'est pas possible d'utiliser un simple compilateur croisé pour ALPHA sur LINUX. La seule machine de ce type à laquelle nous avons pu avoir accès est NYMPHÉA, le supercalculateur de l'IFREMER.

l'architecture utilisée pour réaliser ces simulations sont décrits dans la section 3.4. Malgré cette puissance de calcul importante, seulement cinq programmes étalons ont pu être simulés entièrement de cette façon : `gzip`, `gcc`, `perlbmk`, `gap` et `bzip2`.

Une autre méthode a été envisagée. En effet, la première méthode est rapide (moins de deux heures par programme étalon) mais très peu représentative du fonctionnement du programme en entier contrairement à la seconde méthode qui simule complètement le programme mais qui est extrêmement longue. Dans (Sherwood *et al.*, 2002), Sherwood, Perelman, Hamerly et Calder décrivent une méthode pour identifier des sections d'un programme qui sont représentatives du programme en entier. Il suffit ainsi de ne simuler finement que ces sections afin d'obtenir des résultats proches des résultats qui auraient été obtenus si le programme entier avait été simulé finement. Ils ont appliqué cette méthode aux programmes étalons de la suite SPEC CPU2000 et ont publié les sections ainsi obtenues. Il aurait donc été intéressant d'utiliser ces résultats pour nos simulations. Cependant, la position des sections est très dépendante du compilateur utilisé pour compiler les programmes étalons, puisque les coordonnées de ces sections sont données en nombre d'instructions. Or il a été impossible, au moment du début des simulations, de trouver les binaires pour lesquels les résultats publiés dans (Sherwood *et al.*, 2002) ont un sens et donc cette méthode n'a malheureusement pas pu être utilisée.

3.3.2. *Performances de l'architecture*

Les figures 5 et 6 présentent une comparaison des performances de trois architectures :

– une architecture utilisant HIDE mais sans mécanisme de chiffrement ni de protection de l'intégrité ;

– une architecture CRYPTOPAGE de base, sans mécanisme de cache pour stocker l'arbre de hachage protégeant les informations sur les pages mémoire, sans exécution spéculative des instructions pendant leur vérification, et utilisant une valeur d'authentification (MAC) pour chaque ligne de cache pour l'intégrité (CRYPTOPAGE 0 MAC 1) ;

– une architecture CRYPTOPAGE plus avancée, disposant d'un cache de 1 024 entrées pour stocker des nœuds de l'arbre de hachage, avec exécution spéculative des instructions pendant leur vérification et utilisant une valeur d'authentification par ligne de cache pour l'intégrité (CRYPTOPAGE Spéculatif 1 024 MAC 1).

Le nombre d'instructions exécutées par cycle d'horloge (IPC, *Instruction Per Cycle*) est normalisé à celui obtenu en exécutant les programmes étalons sur une architecture normale ne disposant d'aucun mécanisme de sécurité (l'IPC sur une architecture normale est indiqué pour chacun des programmes étalons en haut de chaque ensemble de barres).

On constate tout d'abord une différence relativement importante entre les résultats obtenus avec les simulations rapides et ceux obtenus avec les simulations complètes. Cette différence est notamment importante dans le cas du programme étalon `gcc`. Alors que lors des simulations rapides, l'architecture CRYPTOPAGE avancée

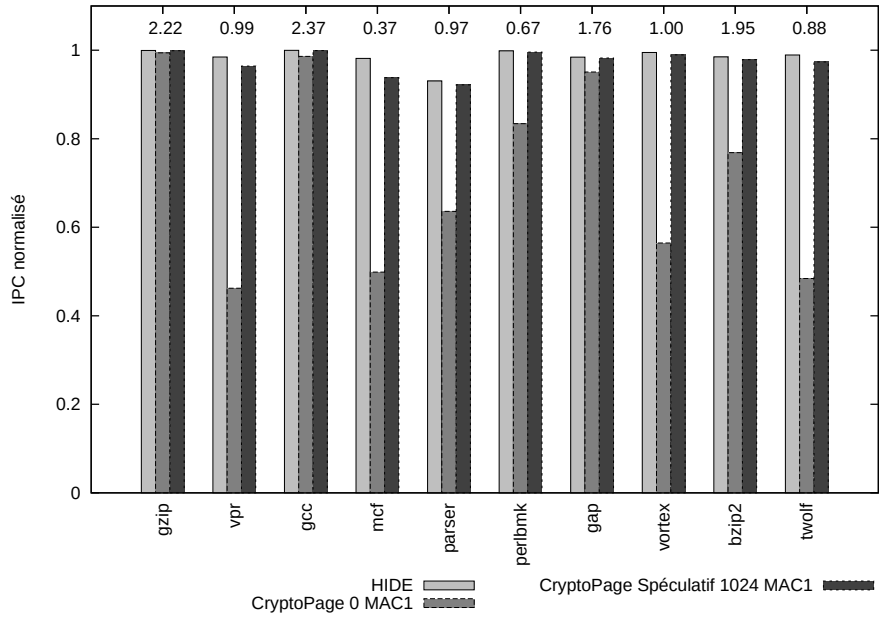


Figure 5. *Performances (simulations rapides,* IPC *normalisés par rapport à une architecture normale)*

(CRYPTOPAGE *Spéculatif* 1 024 MAC 1) n'entraînait qu'une pénalité au niveau des performances de moins de 1 %, cette pénalité monte à 15 % si l'on simule ce programme étalon en entier. Au vu des résultats de HIDE sur ce même programme, on peut en déduire que ce sont les permutations provoquées par HIDE qui posent des problèmes dans ce cas. La simulation rapide a exécuté une zone du programme qui n'était pas représentative de l'ensemble du programme au niveau des accès mémoire (qui entraînent des permutations).

Les performances de l'architecture CRYPTOPAGE de base sont relativement mauvaises (pénalité de 28 % en moyenne pour les simulations rapides et de 25 % pour les simulations complètes) à cause du coût important de la vérification des lignes de cache et de la vérification complète de l'arbre de hachage lors de chaque défaut de TLB.

Dans le cas de l'architecture CRYPTOPAGE améliorée, les performances sont meilleures (pénalité de 3 % en moyenne d'après les simulations rapides et de 8 % en moyenne d'après les simulations complètes, et de seulement 3 % si on exclut `gcc`).

3.3.3. *Impact du cache de l'arbre de hachage*

Nous allons maintenant nous intéresser à l'impact sur les performances de l'ajout d'un cache pour stocker les nœuds les plus utilisés de l'arbre de hachage protégeant

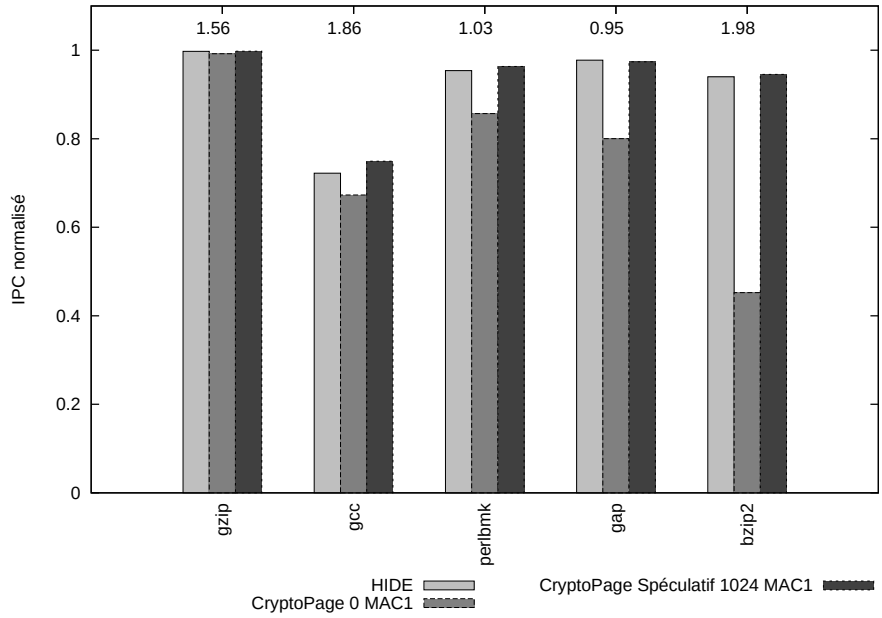


Figure 6. *Performances (simulations complètes,* IPC *normalisés par rapport à une architecture normale)*

les informations sur les pages mémoire. Les figures 7 et 8 présentent les performances d'une architecture CRYPTOPAGE, sans exécution spéculative, utilisant une valeur d'authentification pour chaque ligne de cache et disposant d'un cache pour l'arbre de hachage de 0, 256, 512 ou 1 024 entrées (une entrée peut stocker un nœud et son frère). Les résultats sont normalisés par rapport à ceux obtenus sur une architecture normale non sécurisée.

On remarque que l'ajout d'un cache, même de petite taille, augmente de façon importante les performances (avec les simulations rapides, passage d'une pénalité de 29 % sans cache à 6 % en moyenne avec un cache de 256 entrées, et, avec les simulations complètes, passage d'une pénalité de 25 % à 10 %) de l'architecture en réduisant le nombre d'opérations de vérification des nœuds de l'arbre.

L'augmentation de la taille du cache n'a cependant qu'un impact limité (amélioration de 1 % des performances pour un passage d'un cache de 256 à 1 024 entrées). Pour que l'impact de la vérification de l'arbre de hachage disparaisse, il faudrait que le cache soit suffisamment grand pour stocker tous les nœuds de l'arbre.

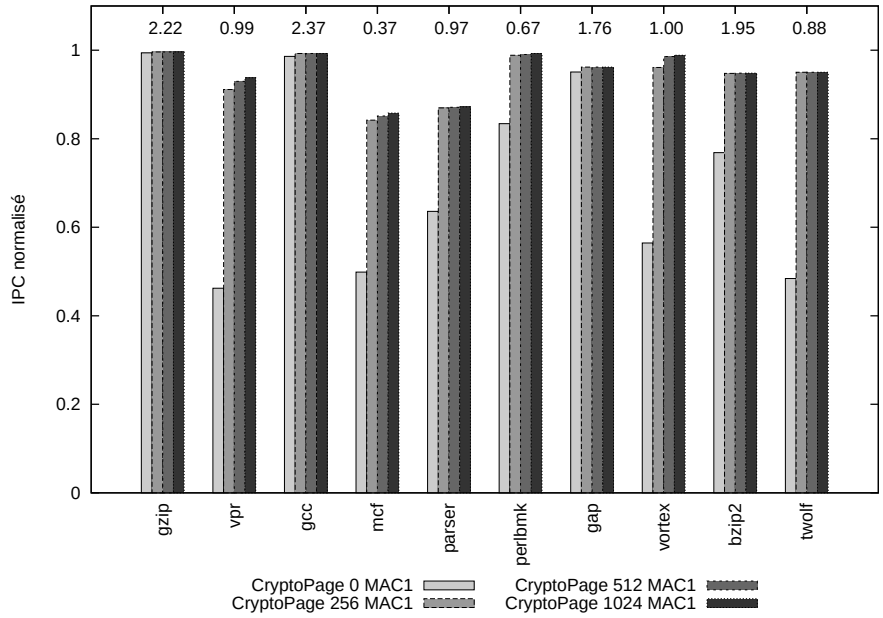


Figure 7. *Impact de la taille du cache de l'arbre de hachage (simulations rapides,* IPC *normalisés par rapport à une architecture normale)*

3.3.4. *Impact de la granularité de l'authentification*

Afin de réduire l'empreinte mémoire d'un processus sécurisé, il est possible de calculer, et donc de stocker, les valeurs d'authentification sur un ensemble de lignes de cache au lieu de le faire pour chaque ligne de cache. Il sera alors nécessaire de récupérer toutes les lignes de cache correspondantes afin de pouvoir vérifier l'intégrité d'une ligne. Cette solution dégrade les performances car la vérification d'intégrité est plus lente (nécessité de récupérer plusieurs lignes de cache pour effectuer la vérification) et car le bus mémoire est plus utilisé (à cause de la récupération des lignes de cache).

Les figures 9 et 10 présentent les performances d'une architecture CRYPTOPAGE, disposant d'un cache pour l'arbre de hachage de 1 024 entrées (afin de minimiser l'impact de la vérification du cache d'arbre de hachage dans ces résultats), sans exécution spéculative et calculant une valeur d'authentification toutes les une, deux ou quatre lignes de cache. Les résultats sont normalisés par rapport à ceux obtenus sur une architecture CRYPTOPAGE avec un cache de 1 024 entrées, sans exécution spéculative et calculant une valeur d'authentification pour chaque ligne de cache.

Pour un calcul d'une valeur d'authentification sur deux lignes de cache, la dégradation des performances est de 5 % en moyenne pour les simulations rapides, et de

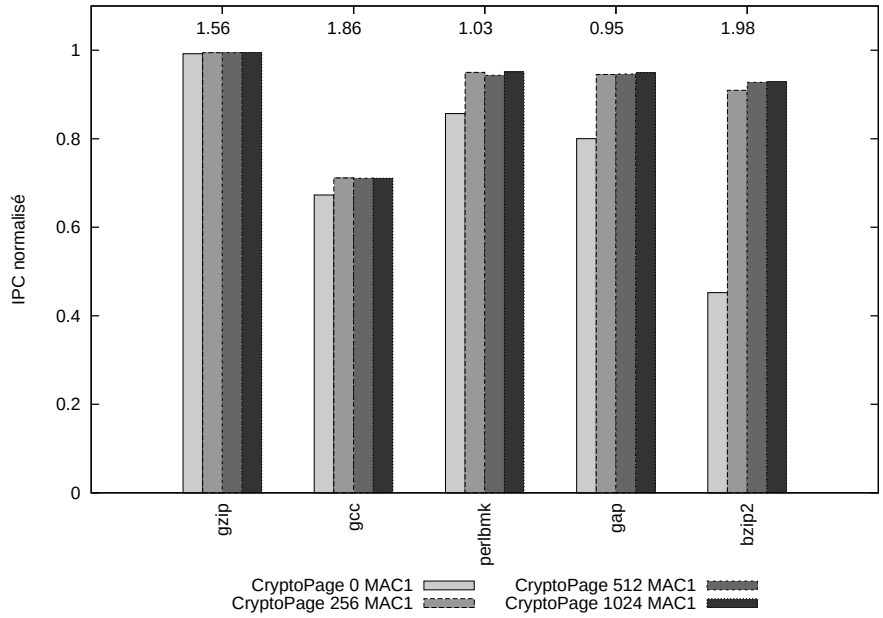


Figure 8. *Impact de la taille du cache de l'arbre de hachage (simulations complètes,* IPC *normalisés par rapport à une architecture normale)*

4 % pour les simulations complètes. Pour quatre lignes de cache, cette dégradation est de 12 % pour les simulations rapides, et de 10 % pour les simulations complètes.

En ce qui concerne l'augmentation de l'empreinte mémoire à cause des valeurs d'authentification, avec les paramètres choisis, elle est de 50 % si elles sont calculées pour chaque ligne de cache, 25 % si elles sont calculées sur deux lignes de cache et 12,5 % si elles sont calculées sur quatre lignes de cache.

Le choix de la granularité du calcul des valeurs d'authentification dépend d'un compromis entre l'empreinte mémoire et les performances.

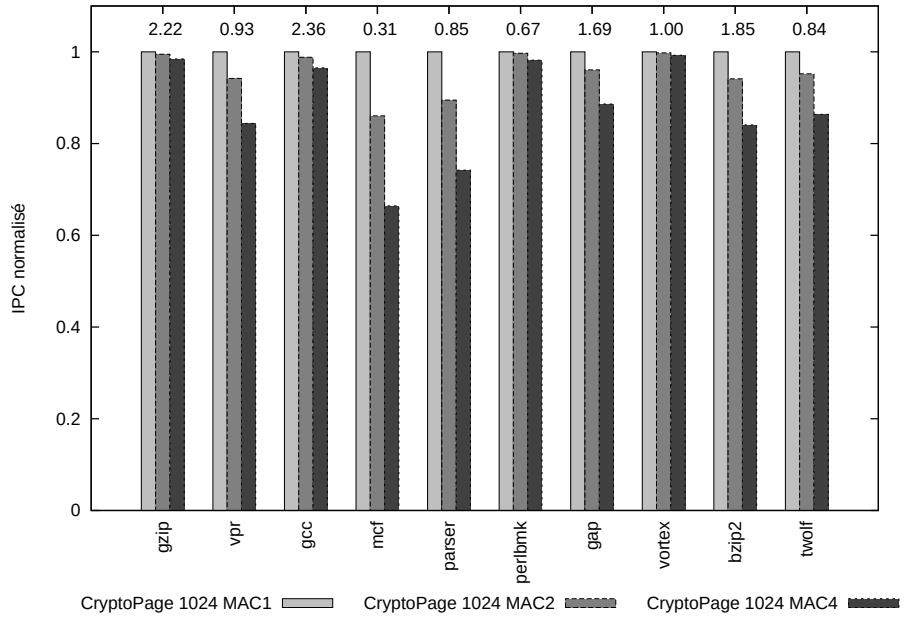


Figure 9. *Impact de la granularité de l'authentification (simulations rapides,* IPC *normalisés par rapport à une architecture* CRYPTOPAGE *1 024 MAC1)*

3.3.5. *Impact de l'exécution spéculative*

Les figures 11 et 12 montrent l'impact de l'exécution spéculative des instructions sur deux architectures différentes :

– CRYPTOPAGE sans cache d'arbre de hachage et calculant une valeur d'authentification sur quatre lignes de cache ;

– CRYPTOPAGE avec un cache d'arbre de hachage de 1 024 entrées et calculant une valeur d'authentification à chaque ligne de cache.

Les résultats sont normalisés par rapport à ceux obtenus sur une architecture normale sans dispositifs de sécurité.

Pour la première architecture, la pénalité moyenne sur les performances passe, avec les simulations complètes, de 30 % sans exécution spéculative à 24 % avec l'exécution spéculative, et avec les simulations rapides, de 33 % à 28 %.

Pour la seconde architecture, la pénalité moyenne sur les performances passe, avec les simulations complètes, de 9 % à 8 %, et, pour les simulations rapides, de 5 % à 3 %.

L'exécution spéculative supprime juste la pénalité entraînée par l'attente de la vérification de l'intégrité des lignes de cache lors de leur lecture depuis la mémoire.

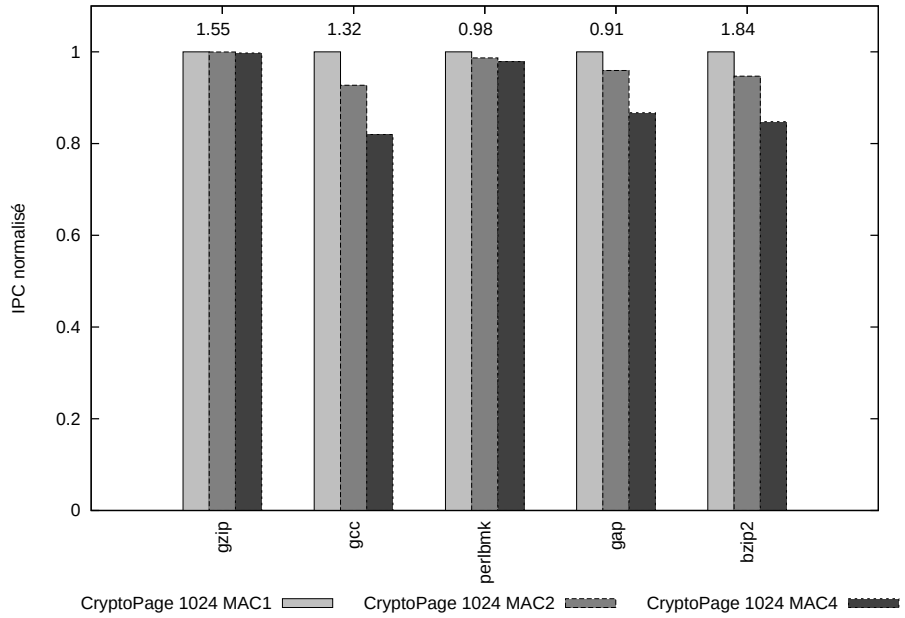


Figure 10. *Impact de la granularité de l'authentification (simulations complètes,* IPC *normalisés par rapport à une architecture* CRYPTOPAGE *1 024 MAC1)*

Quand elle est activée, il reste la pénalité due à la vérification de l'arbre de hachage lors d'un défaut de TLB (cette pénalité est réduite dans l'architecture avec un cache de 1 024 entrées), celle due aux permutations, et celle due à l'encombrement du bus mémoire à cause de ces différents mécanismes.

3.4. *Infrastructure de simulation*

Toutes ces simulations demandent énormément de calcul rien que pour des résultats approximés et pour avoir des résultats plus précis avec des exécutions complètes de ces programmes étalons il faut une infrastructure de calcul puissante.

Pour mener à bien les simulations, il a fallu utiliser beaucoup plus de ressources calcul que ce que nous avions dans l'équipe. Comme nous sommes dans une école, nous avons accès aussi aux machines des élèves qui ne sont pratiquement pas utilisées en dehors des heures de cours et la proximité de toutes ces ressources étaient tentante pour exécuter nos travaux trivialement parallèles car constitués de nombreuses simulations qui sont des tâches indépendantes. Plutôt que de réinventer un système d'exécution par lots en environnement distribué, nous avons utilisé le système Condor qui

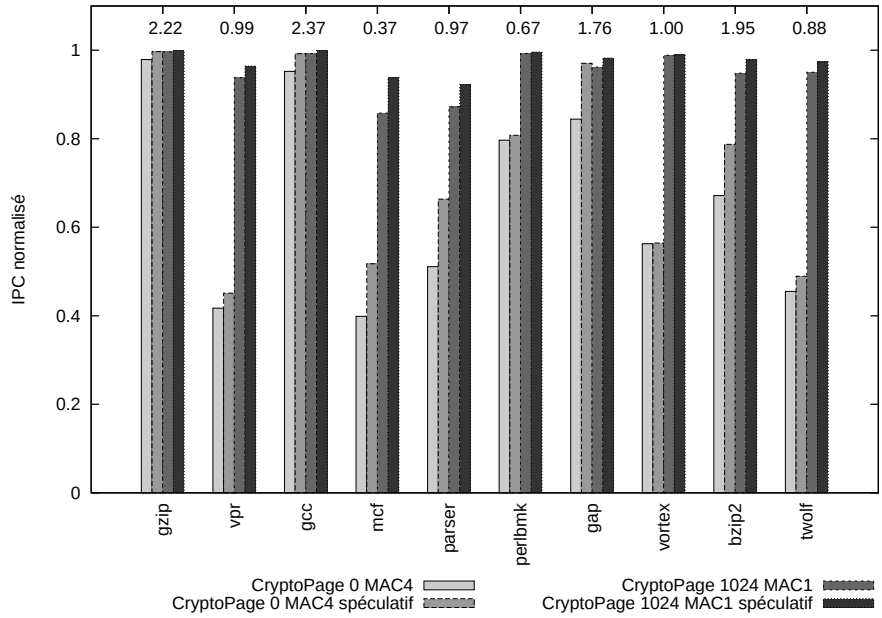


Figure 11. *Impact de l'exécution spéculative (simulations rapides,* IPC *normalisés par rapport à une architecture normale)*

a déjà une vingtaine d'année mais est toujours utilisé et activement maintenu (Mutka et Livny, 1987; The Condor Team, 2007).

Cela nous permet d'avoir une infrastructure de calcul à contrôle centralisé, à entrées sorties centralisées et tolérante aux pannes qui est simple et convient parfaitement à notre besoin de calcul sans trop d'entrées-sorties. Un mécanisme de points de reprise réguliers permet de survivre à la vie instable des ordinateurs des élèves qui ne sont pas à l'abri de redémarrage sauvages ou d'arrêts tout aussi violents. Dans ce cas, Condor relance le processus sur une machine disponible à partir de l'état sauvegardé lors du dernier point de reprise. Pour limiter l'impact sur les travaux pratiques des élèves, Condor est paramétré pour ne faire tourner les processus qu'en dehors des heures de cours, sauf sur les machines bicœur où on suppose qu'un processeur est toujours suffisamment disponible pour travailler sur notre projet. Là encore, c'est le mécanisme de point de reprise qui est utilisé pour arrêter et redémarrer les processus de simulation.

Nous avons utilisé jusqu'à 60 machines de type Intel Pentium 4 (de 2 GHz à 3 GHz, mono- ou bicœur) pendant 6 mois, ce qui représente déjà de l'ordre de 150 000 heures de temps processeur.

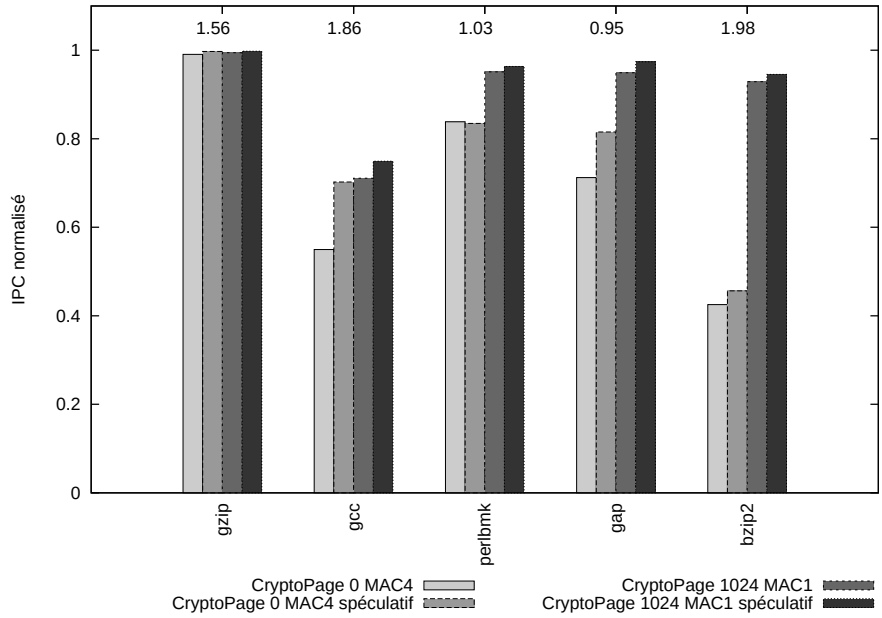


Figure 12. *Impact de l'exécution spéculative (simulations complètes,* IPC *normalisés par rapport à une architecture normale)*

4. Exemple d'application : grille de calcul sécurisée

Une application ambitieuse de ce projet est la réalisation d'une infrastructure de calcul distribuée sécurisée qui permettrait d'une part de certifier des exécutions à distance d'une part et éventuellement des exécutions chiffrées pour garantir la confidentialité.

En effet, si on compare souvent le calcul sur grille à de la ressource de calcul aussi facile à utiliser que de l'électricité *via* les prises du réseau électrique, ce n'est pas vraiment comparable. Les appareils électriques se contentent d'utiliser localement l'énergie potentielle d'électrons distribués par le réseau électrique alors que dans les grilles de calcul il faut envoyer par le réseau (certes) informatique des programmes et des données, puis récupérer les résultats d'exécution. Alors que dans le cas de l'électricité les électrons sont indiscernables, dans le cas des grilles de calcul, les ressources distantes sont discernables. De plus elles sont sous le contrôle de leur propriétaire ou de leur administrateur, plus ou moins bien intentionnés, voire d'attaquants hostiles.

Dans ce cas, il devient important de pouvoir d'abord certifier que le calcul effectué à distance est bien le bon. En effet, dans des algorithmes de crible utilisés par exemple pour trouver les formes de molécules ayant la plus basse énergie, si un nœud de calcul malicieux décide de renvoyer une valeur plus élevée, le destinataire du calcul passera

à côté de ce résultat. Le cas inverse est plus facilement vérifiable, lorsqu'on cherche un cas exceptionnel dans un crible et par exemple qu'un ordinateur malicieux prétend qu'il a trouvé la preuve de signaux extra-terrestres : il suffit que le destinataire du calcul vérifie les quelques résultats positifs potentiels.

Un autre problème du calcul distribué à large échelle est la fuite potentielle de programmes et de données qui peuvent être de grande valeur. Le propriétaire d'un logiciel de simulation complexe n'a pas forcément envie qu'il soit utilisé ou analysé par des tiers. De même, les données manipulées par ces programmes (modèle de voiture, molécules de médicaments potentiels...) ne doivent pas forcément être disséminées dans la nature.

Un moyen de limiter ces problèmes, outre l'obscurcissement de code évoqué dans l'introduction, est de transformer l'algorithme qui produit des données à partir d'entrées en un nouvel algorithme isomorphe qui travaille sur des données chiffrées et produit des résultats chiffrés. Il est aussi possible de rajouter, dans la construction des résultats, des mécanismes de vérification des entrées et d'authentification des sorties mais, malheureusement, cette approche séduisante pose des problèmes d'explosion de la complexité dans la vraie vie (Loureiro *et al.*, 2002). Puisque, dans cette approche, les opérations de base (comme par exemple les opérations sur des nombres flottants qui sont très optimisées dans des processeurs modernes) sont transformées en opérations booléennes élémentaires qui sont exécutées pour simuler un circuit booléen chiffré, les hautes performances attendues dans l'utilisation d'infrastructures de calcul distribué ne sont plus que des vœux pieux.

Une solution plus radicale que nous défendons ici est d'utiliser des ressources sécurisées pour faire du calcul distribué sûr. Ce genre d'approche a déjà été étudié avec des cartes à puces, par exemple en utilisant un réseau de JavaCard comme processeurs de calculs (Chaumette *et al.*, 2003; Chaumette *et al.*, 2006b). Malheureusement dans ce cas les ressources de calcul sont très restreintes ainsi que les moyens de stockage, même si on peut imaginer sous-traiter de manière sécurisée à un hôte local la gestion de mémoire virtuelle supplémentaire (Chaumette *et al.*, 2006a).

Actuellement, il n'y a pas de ressources sécurisées déployées dans le grille et même si dans le futur proche on espère que cela se développera, la majorité des nœuds d'une grille de calcul ne seront pas sécurisés dans un premier temps. C'est pourquoi dans le projet SAFESCALE (SAFESCALE, 2005) nous pensons exploiter différents niveaux de sécurisation avec les différentes équipes du projet.

Afin de bénéficier ne serait-ce que d'un petit nombre de ressources sécurisées, celles-ci vont être utilisées pour vérifier de manière statistique que les ressources non sécurisées ne sont pas compromises (Varrette *et al.*, 2006a). Si aucune ressource sécurisée n'est disponible, nous utilisons une technique de vérification probabiliste avec l'architecture indiquée sur la figure 13. On considère l'application parallèle définie comme un graphe acyclique de tâches distribuées qui produisent des résultats à partir d'entrées fournies par d'autres tâches ou des données. L'idée est de coupler un mécanisme de détection d'erreur avec un mécanisme de points de reprise. Les tâches vont

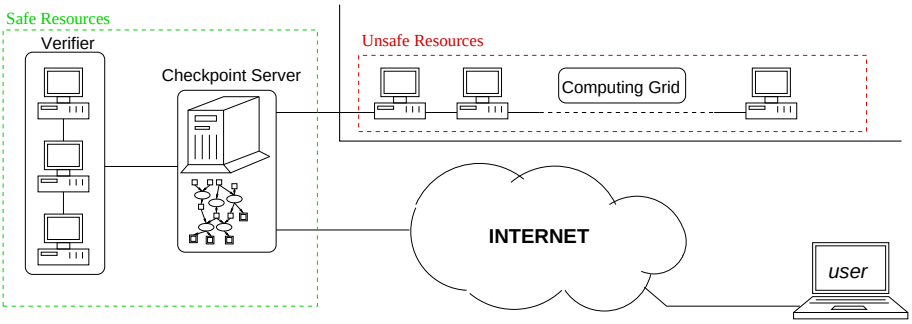


Figure 13. *Hiérarchie de ressources et composants nécessaires pour avoir des algorithmes tolérants aux pannes et détectant des erreurs*

stocker leur état après chaque exécution sur un serveur de reprise sécurisé, résidant en l'occurrence chez la personne qui lance les calculs afin qu'elle ait confiance en cette machine. En parallèle, un vérifieur ré-exécute certaines tâches choisies de manière probabiliste pour vérifier si les résultats correspondent bien au résultat fourni au serveur de reprise. Si le vérifieur détecte une différence, le nœud est corrompu et les calculs doivent être relancés à partir des données du point de reprise avant l'exécution de la tâche corrompue et toutes les tâches qui dépendaient de cette tâches doivent être relancées (Jafar *et al.*, 2004; Krings *et al.*, 2005; Jafar *et al.*, 2005). Pour bien détecter les erreurs, en absence d'infrastructure sécurisée, le vérifieur doit aussi tourner chez la personne qui lance les calculs afin qu'elle ait confiance en cette machine.

Pour des raisons d'efficacité, on ne peut raisonnablement pas vérifier toutes les tâches et donc on ne peut prouver une exécution correcte qu'avec une certaine probabilité de non détection d'attaque qui dépend du taux de vérification.

Si nous avons des ressources sécurisées de type CRYPTOPAGE, certaines parties du système de vérifications vont pouvoir tourner à distance, puisque le support matériel garantira la correcte exécution du programme et la confidentialité de surcroît. Ce dernier point est important pour le vérifieur car le choix pseudo-aléatoire de la tâche à vérifier risquant d'être la sortie d'un générateur pseudo-aléatoire, s'il tourne en clair sur un ordinateur aux mains de l'attaquant, celui-ci pourra prédire les suites aléatoires futures et donc saura quelles tâches subiront un contrôle anti-fraude. Le fraudeur n'aura qu'à perturber les tâches non vérifiées pour falsifier tous les résultats sans que cela soit détecter.

Si nous avons suffisamment de ressources sécurisées performantes, on peut les utiliser pour exécuter aussi des tâches de manières sécurisées. Dans ce cas, elles seront écartées des vérifications et le vérifieur pourra se concentrer sur la vérification de autres tâches qui tourneront sur les nœuds vulnérables.

Notons que dans le cas où toutes les ressources ne sont pas sécurisées, nous ne garantissons plus la confidentialité des calculs et informations sur les ressources non

sécurisées. Par contre nous sommes encore capable de garantir avec un certain taux de probabilité que les résultats n'ont pas été falsifiés.

Ce système, sans la partie avec sécurisation matérielle, a été appliquée à une application médicale d'analyse de mammographies (Varrette *et al.*, 2006b). L'intérêt de rajouter de la sécurisation matérielle permettrait en plus de rajouter une partie confidentialité, si sensible dans le monde des données médicales.

De manière auto-réflexive, on pourrait utiliser ce type de grille de calcul sécurisée pour faire tourner la simulation de l'architecture étudiée dans le section 3.4.

5. Travaux similaires du domaine

5.1. *Architectures sécurisées*

Dans cet article, nous avons présenté une nouvelle architecture sécurisée permettant de combiner, à un coût raisonnable, le chiffrement mémoire, la protection de l'intégrité mémoire et la protection contre les fuites d'informations *via* le bus d'adresse. Ces trois mécanismes sont essentiels pour les architectures informatiques sécurisées afin de permettre de garantir la confidentialité et l'intégrité de processus.

Il existe principalement deux familles d'architectures permettant de garantir l'intégrité et la confidentialité de processus sécurisés.

5.1.1. *Approche à coprocesseur*

Dans la première famille, parfois appelée approche à *coprocesseurs*, tout l'environnement d'exécution des processus sécurisés (processeur, mémoire permanente et volatile, système d'exploitation) est inséré au sein d'une enceinte blindée résistante aux attaques matérielles et pouvant disposer de dispositifs d'autodestruction si jamais cette enceinte est percée. Les échanges entre les processus sécurisés s'exécutant dans cet environnement blindé et l'hôte (ordinateur, terminal de paiement, etc.) sont effectués *via* des canaux d'entrées-sorties. Les cartes à puce et l'IBM 4758 (IBM, 2007) (carte PCI intégrant un processeur $x86$, de la mémoire flash et de la mémoire vive au sein d'une enceinte blindée avec dispositif d'autodestruction) font partie de cette famille.

Cette approche entraîne quelques inconvénients, notamment au niveau des performances et de l'évolutivité limitée.

5.1.2. *Architectures à chiffrement de bus*

La seconde famille, aussi appelée architectures à chiffrement de bus, à laquelle l'architecture proposée dans cet article fait partie, part du principe que seul le processeur sur lequel s'exécutent les processus sécurisés est sécurisé et inattaquable. Le reste de l'ordinateur (bus, mémoire, périphérique de stockage, système d'exploitation, etc.) n'est pas considéré comme digne de confiance et donc peut être sous le contrôle d'un

attaquant. Ces architectures sécurisées ont été introduites dans le monde civil par Best dans une série de brevets (Best, 1979; Best, 1980; Best, 1981; Best, 1984).

Cependant, ces premières architectures ne protégeaient que la confidentialité des programmes (en chiffrant tout ce qui sort du processeur) et pas leur intégrité. C'était notamment le cas de l'architecture TrustNo 1 proposée par (Kuhn, 1997) ou de la série de microcontrôleurs DS5000 commercialisés par la société *Dallas Semiconductor* (Dallas, 2006). Ces architectures étaient donc vulnérables car un attaquant pouvait modifier le contenu de la mémoire et ainsi perturber sa bonne exécution, malgré le chiffrement (le DS5002FP a notamment été cassé (Kuhn, 1998) de cette manière).

Les architectures récentes proposent quant à elles également la protection de l'intégrité des processus sécurisés.

5.1.2.1. Intégrité mémoire

Dans cet article, nous avons proposé un mécanisme permettant d'implémenter un vérificateur mémoire en ligne en utilisant l'infrastructure HIDE. L'objectif d'un vérificateur mémoire est de permettre au processeur de vérifier que la mémoire se comporte bien comme une mémoire normale, c'est-à-dire que la valeur lue à une adresse en mémoire doit être identique à celle stockée par le processeur en dernier à cette adresse. On peut ainsi distinguer trois attaques possibles contre la mémoire : injection (un attaquant injecte une valeur à une adresse particulière en mémoire), permutation spatiale (l'attaquant déplace une donnée stockée en mémoire vers une autre adresse) et permutation temporelle (aussi appelée attaque par rejeu où l'attaquant sauvegarde la valeur stockée à une adresse en mémoire et où il la replace à la même adresse plus tard dans le temps).

Les deux premières attaques sont relativement simples à contrer, par exemple en utilisant des MAC (*Message Authentication Codes*) calculés à l'aide d'une clé secrète sur les données et leur adresse. Ce mécanisme est utilisé notamment par XOM (Lie *et al.*, 2000) et par *Secret-Protected* (Lee *et al.*, 2005).

Les attaques par rejeu nécessitent des mécanismes plus complexes. Dans (Gassend *et al.*, 2003), un arbre de MERKLE est utilisé, calculé sur toute la mémoire, afin d'implémenter un tel vérificateur. Ils ont également proposé d'utiliser le cache du processeur afin d'améliorer la vitesse de la vérification d'intégrité en stockant certaines parties de l'arbre dans le cache supposé non attaquable. Cependant, même en utilisant le cache, ils obtiennent un surcoût en termes de performances de l'ordre de 20 %. Dans notre proposition, nous combinons un mécanisme simple de MAC et un arbre de MERKLE couvrant uniquement les informations sur les pages mémoire (et non pas toute la mémoire par elle-même) afin de construire un vérificateur mémoire, résistant aux attaques par rejeu, et avec de meilleures performances.

Un vérificateur mémoire hors ligne à également été proposé dans (Gassend *et al.*, 2003) qui utilise des fonctions de hachage incrémentales.

Un autre vérificateur mémoire hors ligne avec un surcoût constant en termes de bande passante mémoire a été proposé dans (Clarke *et al.*, 2005) en combinant des arbres de MERKLE et des fonctions de hachage incrémentales. Les vérificateurs mémoire hors-ligne sont plus rapides [12] mais ils doivent être appelés avant l'exécution de chaque fonction critique. Entre chaque vérification, les instructions sont exécutées sans vérification et ce comportement peut entraîner des problèmes de sécurité.

5.1.2.2. Chiffrement mémoire

Le chiffrement mémoire est également un des points essentiels des architectures sécurisées. Les premières architectures proposées (TrustNo 1 (Kuhn, 1997) par exemple) utilisaient le mode d'opération ECB (*Electronic CodeBook*) d'un algorithme de chiffrement par bloc. Ce mode présente le désavantage de ne pas cacher certains motifs et ainsi de laisser fuir des informations vers un attaquant. Le mode CBC (*Cipher Bloc Chaining*) a ensuite été utilisé dans les architectures suivantes, comme par exemple pour *Secret-Protected* (Lee *et al.*, 2005), les premières versions d'AEGIS (Suh *et al.*, 2003b; Suh *et al.*, 2005) et les premières versions de CRYPTOPAGE (Lauradoux et Keryell, 2003; Keryell, 2000; Duc *et al.*, 2005; Duc, 2004).

Il a également été proposé d'utiliser le mode compteur afin de chiffrer les lignes de cache (Shi *et al.*, 2005). Cependant, ils ont besoin de stocker le compteur en mémoire et donc le calcul des masques est retardé jusqu'à la récupération du compteur. Afin de réduire ce problème, ils proposent d'utiliser une unité de prédiction associée à un cache spécialisé afin d'essayer de prédire les compteurs. Dans notre proposition, les compteurs sont déduits des données sur les pages utilisées. Comme ces informations sont toujours disponibles avant un accès mémoire, excepté dans le cas d'un défaut de TLB, les masques peuvent toujours être calculés en parallèle à l'accès mémoire, sans avoir besoin de recourir à une unité de prédiction ou un cache particulier.

Certaines architectures utilisent un mode de chiffrement permettant de combiner le chiffrement et l'intégrité. Par exemple, dans (Elbaz, 2006; Elbaz *et al.*, 2006a; Elbaz *et al.*, 2006b), Elbaz présente pour des cibles embarquées PE-ICE (*Parallelized Encryption and Integrity Checking Engine*) qui est un mécanisme permettant de chiffrer et de garantir l'intégrité d'une information en utilisant un seul algorithme de chiffrement. Le principe est d'associer l'information proprement dite, de la redondance et un aléa, et de chiffrer à l'aide d'un mode d'opération d'un algorithme de chiffrement par bloc ayant certaines propriétés (diffusion d'erreur). L'aléa est stocké de manière sécurisée à l'intérieur du processeur. Lors du déchiffrement, on vérifie que la redondance et l'aléa sont corrects, ce qui permet de vérifier l'intégrité. Elbaz montre également comment implémenter cet algorithme au sein d'un processeur embarqué et évalue ses performances. Afin de garantir la protection contre le rejeu mémoire, les aléas utilisés par PE-ICE doivent être stockés de façon sécurisée et leur intégrité doit être protégée, y compris contre des attaques par rejeu. A cette fin, Elbaz présente également PRV-Tree (*PE-ICE Protected Reference Random Values Tree*). Ce mécanisme est basé

12. Ils vérifient l'intégrité d'une série de transactions et non pas de chaque transaction comme le font les vérificateurs mémoire en ligne.

sur un arbre de hachage dans lequel chaque nœud est chiffré et protégé à l'aide de l'algorithme PE-ICE et où chaque nœud contient l'aléa ayant servi à chiffrer et à protéger les nœuds fils (contrairement à un arbre de hachage où chaque nœud contient le résumé cryptographique des nœuds fils). Ainsi, seul le nœud racine a besoin d'être stocké sur le processeur dans une mémoire sécurisée. L'arbre couvre donc toute la mémoire, contrairement à l'architecture que nous proposons dans cet article où l'arbre de hachage est plus petit puisqu'il ne couvre que les descripteurs de pages mémoire.

D'autres modes d'opération permettent de combiner le chiffrement et l'intégrité. L'utilisation du mode d'opération Galois/Compteur (GCM) est proposé afin de réduire la latence introduite par la vérification mémoire par (Yan *et al.*, 2006).

Ce mode d'opération combiné permet, en plus de la confidentialité (assurée par un mode compteur normal) d'assurer l'authentification des données. L'avantage de ce mode par rapport à des fonctions de hachage ou des MAC courants est qu'il est beaucoup plus rapide. Une version de CRYPTOPAGE utilisant GCM à la place du mode compteur et du MAC est en cours de développement.

5.1.2.3. Fuites d'informations sur le bus d'adresse

Enfin, le problème des fuites d'informations sur le bus d'adresse a été étudié dans plusieurs articles. Une approche possible est de chiffrer le bus d'adresse, ce qui est équivalent à effectuer une permutation initiale de l'espace mémoire. Par exemple, ce mécanisme est utilisé dans les processeurs de type DS5000. Cependant, si le chiffrement utilisé ne varie pas au cours du temps, un attaquant peut toujours remarquer qu'une ligne est plus utilisée qu'une autre. Plusieurs approches ont été proposées par (Goldreich et Ostrovsky, 1996) pour garantir qu'aucune fuite d'informations ne peut se produire sur le bus d'adresse mais leurs algorithmes réduisent considérablement les performances.

5.2. *Informatique de confiance*

Pour terminer, il est nécessaire de citer les efforts menés par le *Trusted Computing Group* (TCG) (TCG, 2007) dans le domaine de l'informatique de confiance. Il est important de noter que ses objectifs, ainsi que les modèles d'attaques considérés, sont différents de ceux des architectures sécurisées : il s'agit initialement de protéger les données importantes d'un utilisateur contre des attaques principalement logicielles. Le TCG a notamment spécifié le fonctionnement du *Trusted Platform Module* (TPM), petite puce attachée physiquement à la carte mère, qui est présent maintenant dans de nombreux ordinateurs. Lors du démarrage, si elle est activée par l'utilisateur, cette puce va effectuer un certain nombre de mesures sur la configuration matérielle et logicielle de l'ordinateur. Ces mesures vont être stockées de façon sécurisée au sein du TPM. Le TPM va ensuite pouvoir, à la demande d'une entité externe, attester (à l'aide d'une signature numérique) de l'état matériel et logiciel d'une plate-forme. L'utilisateur va également pouvoir, à l'aide de fonctions dédiées, lier une donnée à une

configuration matérielle et/ou logicielle donnée. Ainsi, cette donnée ne pourra être révélée par le TPM ultérieurement que si la plate-forme est dans l'état spécifié lors de l'enregistrement. Cependant, contrairement aux architectures sécurisées présentées précédemment, il n'est pas possible, avec un TPM, de protéger l'intégrité et la confidentialité de programmes complets, et encore moins contre des attaques matérielles.

6. Conclusions et travaux futurs

Dans cet article, nous avons décrit une solution pour implémenter, à faible coût d'exécution, un mécanisme de chiffrement et de protection de l'intégrité de la mémoire sur l'infrastructure HIDE, qui elle-même empêche les fuites d'informations *via* le bus d'adresse. Nous avons également proposé un mécanisme permettant de déléguer une partie des opérations de vérification au système d'exploitation sans avoir besoin de lui faire confiance grâce à l'introduction de seulement 2 nouvelles instructions de sécurité.

L'impact sur les performances de ces mécanismes est de seulement 3 % à 8 %, par rapport à une architecture normale non sécurisée, ce qui est largement meilleur que les autres solutions proposées jusqu'à présent. Ce résultat est obtenu grâce à une combinaison d'arbres de MERKLE et de MAC.

En termes de complexité matérielle, c'est difficile à chiffrer sans réalisation concrète hors de portée d'une équipe académique. On estime néanmoins que cela représenterait de l'ordre du pour cent pour un processeur généraliste avec de l'ordre d'un milliard de transistors.

Nous pensons qu'il est donc désormais possible d'avoir des architectures performantes sécurisées à prix raisonnable. Cela permettrait d'avoir, sans rogner sur les performances, des ordinateurs capables de faire fonctionner des programmes en mode chiffré garantissant la confidentialité des données et programmes en mémoire centrale et résistant aux attaques logicielles *et* matérielles, ce que les approches de type TCG ne gèrent pas. Avec notre architecture, les approches telles que TCG sont caduques.

Par contre il faudrait s'intéresser à la preuve de sécurité de telles architectures sécurisées. Mais vue l'ampleur de la tâche, personne, à notre connaissance, ne s'est lancé dans cette voie pourtant nécessaire.

Une application passionnante serait de pouvoir construire des architectures distribuées permettant des exécutions sécurisées de la même manière mais à distance, ce qui n'est pas possible actuellement où on se contente de sécuriser localement des exécutions venant de l'extérieur. Ce concept de grilles de calcul sécurisées est en cours d'étude dans notre projet SAFESCALE à l'utilisation de tels systèmes pour faire du calcul distribué haute performance de confiance (Varrette *et al.*, 2006a).

Nous travaillons actuellement à étendre nos propositions à des processeurs multicœurs et à des systèmes multiprocesseurs ainsi qu'à intégrer ses mécanismes dans Linux. Avec la généralisation des machines virtuelles avec support matériel dans les

processeurs, nous allons aussi regarder comment il est possible de sécuriser matériellement cette virtualisation sans que le logiciel de paravirtualisation ait lui-même besoin d'être sécurisé.

Enfin, nous regardons la problématique d'authentification des processus chiffrés qui fonctionnent sur une machine. En effet, maintenant que le concept de processus opaques chiffrés existe, se pose le problème pour un propriétaire de vérifier que le programme exécuté est bien un programme non malicieux. On peut certes vérifier le hachage du contexte matériel chiffré par rapport à une liste de programmes autorisés, certifiés par exemple par un tiers de confiance, mais ce n'est pas adapté à du calcul distribué ouvert. En rajoutant un mécanisme d'authentification du programme en clair à l'intérieur du processus chiffré, il est possible de vérifier qu'un programme correspond bien à un code source donné, compilé dans un environnement donné (Duc et Keryell, 2008).

Remerciements

Ce travail est soutenu par une bourse de thèse de la Délégation Générale pour l'Armement (DGA), par le projet ANR ARA/SSIA BGPR SAFESCALE et par le projet de l'Institut TÉLÉCOM TCP (Trusted Computing Platform). Merci aux collègues de TÉLÉCOM ParisTech du projet TCP (en particulier Sylvain Guilley et Renaud Pacalet) et du projet SAFESCALE pour leur nombreuses interactions (particulièrement Sébastien Varette, Jean-Louis Roch et Christophe Cérin), à Jacques Stern et aux membres de l'équipe de cryptographie de l'ENS pour les discussions sur le projet. Merci à Loïc Plassart pour la relecture finale. Enfin, pour terminer, merci aux 4 rapporteurs de cet article ainsi qu'à Laurence Sourdillon pour leur excellent travail.

7. Bibliographie

Bellare, M., Desai, A., Jokipii, E. et Rogaway, P., octobre 1997. "A Concrete Security Treatment of Symmetric Encryption." *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (*FOCS*'97)*. IEEE Computer Society, p. 394–403.

Bellare, M. et Namprempre, C., 2000. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm." *Advances in Cryptology* - ASIACRYPT *2000 Proceedings*. Springer-Verlag, vol. 1976 de *Lecture Notes in Computer Science*.

Bellare, M. et Sahai, A., 2006. "Non-malleable encryption: equivalence between two notions, and an indistinguishability-based characterization." Cryptology ePrint Archive report 2006/228.

Best, R. M., septembre 1979. *Microprocessor for Executing Enciphered Programs*. rapport technique US4168396, United States Patent.

Best, R. M., février 1980. "Preventing Software Piracy with Crypto-Microprocessors." IEEE *Spring* COMPCON*'80*. IEEE Computer Society, p. 466–469.

Best, R. M., juillet 1981. *Crypto Microprocessor for Executing Enciphered Programs*. rapport technique US4278837, United States Patent.

Best, R. M., août 1984. *Crypto Microprocessor that Executes Enciphered Programs*. rapport technique US4465901, United States Patent.

Chaumette, S., Grange, P., Sauveron, D. et Vignéras, P., juillet 2003. "Computing with Java Cards." *International Conference on Computer, Communication and Control Technologies (CCCT'03)*. Orlando, FL, USA.

Chaumette, S., Karray, A. et Sauveron, D., octobre 2006a. "Extended Secure Memory for a Java Card in the Context of the Java Card Grid project." *11th IEEE Nordic Workshop on Secure IT-systems: NordSec 2006*. Linköping, Suède.

Chaumette, S., Karray, A. et Sauveron, D., mai 2006b. "Secure Collaborative and Distributed Services in the Java Card Grid Platform." *Workshop on Collaboration and Security (COLSEC'06)*. Las Vegas, Nevada, USA.

Clarke, D., Suh, G. E., Gassend, B., Sudan, A., van Dijk, M. et Devadas, S., mai 2005. "Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data." *Proceedings of the 2005* IEEE *Symposium on Security and Privacy*. IEEE Computer Society, p. 139–153.

Collberg, C. S. et Thomborson, C., août 2002. "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection." *IEEE Transactions on Software Engineering*. vol. 28, p. 735–746.

Dallas, juillet 2006. *DS5002FP Secure Microprocessor Chip*. Dallas Semiconductor. `http://datasheets.maxim-ic.com/en/ds/DS5002FP.pdf`.

Duc, G., juin 2004. CRYPTOPAGE — *an architecture to run secure processes*. Diplôme d'Études Approfondies, École Nationale Supérieure des Télécommunications de Bretagne, DEA de l'Université de Rennes 1. `http://enstb.org/~gduc/dea/rapport/rapport.pdf`.

Duc, G., 2007. *Support matériel, logiciel et cryptographique pour une exécution sécurisée de processus*. thèse, École Nationale Supérieure des Télécommunications de Bretagne. `http://enstb.org/~gduc/these/these.pdf`.

Duc, G. et Keryell, R., février 2008. « Support architectural pour identification de programmes chiffrés dans une architecture sécurisée sans système d'exploitation de confiance. » *Symposium en Architecture de Machines (*SYMPA*'2008)*.

Duc, G., Keryell, R. et Lauradoux, C.,« CRYPTOPAGE : Support matériel pour cryptoprocessus. » *Technique et Science Informatiques*, vol. 24, (2005), p. 667–701.

Elbaz, R., 2006. *Hardware Mechanisms for Secured Processor-Memory Transactions in Embedded Systems*. thèse, Université de Montpellier II.

Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P. et Bardouillet, M., avril 2006a. "PE-ICE: Parallelized Encryption and Integrity Checking Engine." *Proceedings of the 9th* IEEE *workshop on Design and Diagnostics of Electronic Circuits and Systems*. p. 141–142.

Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M. et Martinez, A., juillet 2006b. "A parallelized way to provide data encryption and integrity checking on a processor-memory bus." *Proceeding of the 43rd* ACM/IEEE *Design Automation Conference*. p. 506–509.

Gassend, B., Suh, G. E., Clarke, D., van Dijk, M. et Devadas, S., février 2003. "Caches and Hash Trees for Efficient Memory Integrity Verification." *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (*HPCA*'03)*. p. 295–306.

Goldreich, O. et Ostrovsky, R., "Software protection and simulation on oblivious RAMs." *Journal of the* ACM, vol. 43, n° 3, (1996), p. 431–473.

Goldwasser, S. et Micali, S., "Probabilistic Encryption." *Special issue of Journal of Computer and Systems Sciences*, vol. 28, n° 2, (1984), p. 270–299.

Henning, J. L., "SPEC CPU2000: measuring CPU performance in the New Millennium." IEEE *Computer*, vol. 33, n° 7, (2000), p. 28–35.

Huang, A., mai 2002. *Keeping Secrets in Hardware: the Microsoft XBox (*TM*) Case Study*. rapport technique AI Memo 2002-008, Massachusetts Institute of Technology.

IBM, mai 2007. "IBM PCI Cryptographic Coprocessor." `http://www-03.ibm.com/security/cryptocards/pcicc/overview.shtml`.

Jafar, S., Krings, A. W., Gautier, T. et Roch, J.-L., May 2005. "Theft-Induced Checkpointing for Reconfigurable Dataflow Applications." IEEE (éditeur), *IEEE Electro/Information Technology Conference , (EIT 2005)*. Lincoln, Nebraska. This paper received the EIT'05 Best Paper Award.

Jafar, S., Varrette, S. et Roch, J.-L., September 2004. "Using Data-Flow Analysis for Resilence and Result Checking in Peer to Peer Computations." IEEE (éditeur), *IEEE DEXA'2004 - Workshop GLOBE'04: Grid and Peer-to-Peer Computing Impacts on Large Scale Heterogeneous Distributed Database Systems*. Zaragoza, Spain, p. 512–516.

Keryell, R., juin 2000. « CRYPTOPAGE-1 : vers la fin du piratage informatique ? » *Symposium d'Architecture (*SYMPA*'6)*. Besançon, p. 35–44.

Keryell, R. et Paris, N., août 1993. "Activity Counter : New Optimization for the Dynamic Scheduling of SIMD Control Flow." *1993 International Conference on Parallel Processing — Volume II : Software*. Saint Charles, Ohio, USA, p. II–184–II–187.

Kocher, P. C., août 1996. "Timing Attacks on Implementations of DIFFIE-HELLMAN, RSA, DSS, and Other Systems." *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (*CRYPTO*'96)*. Springer-Verlag, 1109, p. 104–113.

Kocher, P. C., Jaffe, J. et Jun, B., août 1999. "Differential Power Analysis." *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (*CRYPTO*'99)*. Springer-Verlag, 1666, p. 388–397.

Krings, A., Roch, J.-L., Jafar, S. et Varrette, S., February 14–16 2005. "A Probabilistic Approach for Task and Result Certification of Large-scale Distributed Applications in Hostile Environments." S. Verlag (éditeur), *Proceedings of the European Grid Conference (EGC2005)*. LNCS, Springer Verlag, Amsterdam, Netherlands, LNCS 3470.

Kuhn, M., avril 1997. *The* TRUSTNO1 *cryptoprocessor concept*. rapport technique CS555, Purdue University.

Kuhn, M. G., octobre 1998. "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP." IEEE *Transaction on Computers*. IEEE Computer Society, vol. 47, p. 1153–1157.

Lauradoux, C. et Keryell, R., octobre 2003. « CRYPTOPAGE-2 : un processeur sécurisé contre le rejeu. » *Symposium en Architecture et Adéquation Algorithme Architecture (*SYMPAAA*'2003)*. La Colle sur Loup, France, p. 314–321.

Lee, R. B., Kwan, P. C. S., McGregor, J. P., Dwoskin, J. et Wang, Z., juin 2005. "Architecture for Protecting Critical Secrets in Microprocessors." *Proceedings of the 32nd Annual International Symposium on Computer Architecture (*ISCA*'05)*. IEEE Computer Society, p. 2–13.

Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J. et Horowitz, M., octobre 2000. "Architectural support for copy and tamper resistant software." *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (*ASPLOS IX*)*. p. 168–177.

Lie, D., Trekkath, C. A. et Horowitz, M., octobre 2003. "Implementing an Untrusted Operating System on Trusted Hardware." *Proceedings of the 9th* ACM *Symposium on Operating Systems Principles (*SOSP*'03)*. p. 178–192.

Lie, D. J., 2004. *Architectural support for copy and tamper-resistant software*. thèse, Stanford University.

Loureiro, S., Bussard, L. et Roudier, Y., 2002. "Extending Tamper-Proof Hardware Security to Untrusted Execution Environments." *CARDIS*. p. 111–124.

Merkle, R. C., 1989. "A Certified Digital Signature." *Proceedings on Advanced in Cryptology (*CRYPTO*'89)*. Springer-Verlag New York, Inc., vol. 435, p. 218–238.

Molnar, D., Piotrowski, M., Schultz, D. et Wagner, D., 2005. "The program counter security model: Automatic detection and removal of control-flow side channel attacks (full version)." IACR eprint archive report 2005/368.

Mutka, M. W. et Livny, M., décembre 1987. "Profiling Workstations' Available Capacity for Remote Execution." *Performance '87, Proceedings of the 12th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*. p. 529–544.

NIST, mai 1985. "Computer Data Authentication." Federal Information Processing Standards Publication 113.

NIST, décembre 2001. "Recommendation for Block Cipher Modes of Operation." Special Publication 800-38A.

SAFESCALE, 2005. "SAFESCALE project home page: Security And Fault-tolerance to Exploit Safety ambient Computing in lArge scaLe Environments." `https://www-lipn.univ-paris13.fr/safescale/`.

Sherwood, T., Perelman, E., Hamerly, G. et Calder, B., octobre 2002. "Automatically characterizing large scale program behavior." *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (*ASPLOS-X*)*. ACM Press, p. 45–57.

Shi, W., Lee, H.-H. S., Ghosh, M., Lu, C. et Boldyreva, A., juin 2005. "High Efficiency Counter Mode Security Architecture via Prediction and Precomputation." *Proceedings of the 32nd Annual International Symposium on Computer Architecture (*ISCA*'05)*. IEEE Computer Society, p. 14–24.

SPEC, avril 2007. "Standard Performance Evaluation Corporation." `http://www.spec.org`.

Suh, G. E., Clarke, D., Gassend, B., van Dijk, M. et Devadas, S., décembre 2003a. "Efficient Memory Integrity Verification and Encryption for Secure Processors." *Proceedings of the 36th Annual* IEEE/ACM *International Symposium on Microarchitecture*.

Suh, G. E., Clarke, D., Gassend, B., van Dijk, M. et Devadas, S., juin 2003b. "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing." *Proceedings of the 17th International Conference on Supercomputing (*ICS*'03)*. p. 160–171.

Suh, G. E., O'Donnell, C. W., Sachdev, I. et Devadas, S., juin 2005. "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions." *Pro-*

*ceedings of the 32nd Annual International Symposium on Computer Architecture (*ISCA*'05)*. IEEE Computer Society, p. 25–36.

TCG, février 2007. "Trusted Computing Group." `http://www.trustedcomputinggroup.org`.

The Condor Team, mars 2007. "Condor Project Homepage." `http://www.cs.wisc.edu/condor/`.

Varrette, S., Duc, G., Keryell, R., Roch, J.-L. et Leprevost, F., juin 2006a. *Building Secure Resources to Ensure Safe Computations in Distributed and Potentially Corrupted Environments*. rapport technique, ENST Bretagne. `http://enstb.org/~keryell/publications/rapports/2006/SAFESCALE`.

Varrette, S., Roch, J.-L., Montagnat, J., Seitz, L., Pierson, J.-M. et Leprévost, F., june 2006b. "Safe Distributed Architecture for Image-based Computer Assisted Diagnosis." *IEEE 1st International Workshop on Health Pervasive Systems (HPS'06)*. Lyon, France.

Yan, C., Englender, D., Prvulovic, M., Rogers, B. et Solihin, Y., juin 2006. "Improving Cost, Performance, and Security of Memory Encryption and Authentication." *Proceedings of the 33rd International Symposium on Computer Architecture (*ISCA*'06)*. IEEE Computer Society, p. 179–190.

Zhuang, X., Zhang, T. et Pande, S., octobre 2004. "HIDE: an infrastructure for efficiently protecting information leakage on the address bus." *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (*ASPLOS-XI*)*. ACM Press, p. 72–84.

Article reçu le 7 mars 2007
Accepté après révisions le 17 avril 2008

Guillaume Duc *est titulaire d'un diplôme d'ingénieur de TÉLÉCOM Bretagne (2004), d'un diplôme d'études approfondies de l'Université de Rennes 1 (2004) et d'une thèse en Informatique de Rennes 1 à TÉLÉCOM Bretagne (2007). Il est actuellement en séjour post-doctoral dans les Orange Labs à Caen. Ses travaux de recherche concernent les architectures sécurisées matérielles et logicielles.*

Ronan Keryell *est un ancien élève de l'*ENS *(École Normale Supérieure) de la rue d'Ulm (1986) où il a effectué sa thèse en informatique (1992) avec l'Université Paris Sud (Paris XI). Il a travaillé comme enseignant-chercheur au Centre de Recherche en Informatique de l'École des Mines ParisTech, comme consultant et est au Département Informatique de TÉLÉCOM Bretagne depuis 1999. Il a aussi participé à la création de 3 jeunes-pousses en informatique, dont la dernière,* HPC PROJECT*, le finance actuellement. Ses centres d'intérêt tournent autour de l'informatique et des architectures à haute performance, la compilation, les systèmes d'exploitation et la sécurité des systèmes haute performance.*