CryptoPage : une architecture efficace combinant chiffrement, intégrité mémoire et protection contre les fuites d'informations

Cyril Brulebois, Guillaume Duc, and Ronan Keryell

ENST Bretagne,
Technopôle Brest-Iroise,
CS 83818,
29200 Brest, France
{cyril.brulebois,guillaume.duc,ronan.keryell}@enst-bretagne.fr

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 terme 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 %).

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.

Durant ces dernières années, plusieurs architectures (comme XOM [14,15,16], AEGIS [21,22] et CRYPTOPAGE [13,6,5]) 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 [10] 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. Zhuang *et al.* ont montré dans [23] que la connaissance de ces motifs d'accès peut être suffisante pour identifier certains algorithmes utilisés, 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 [23]. 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 terme 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 terme de performances de ce système et la section 4 présente les autres travaux menés dans ce domaine.

2 Architecture

Dans cette section, nous présenterons tout d'abord les objectifs en terme de sécurité de notre architecture. Nous résumerons ensuite les concepts clés de l'infrastructure Hide dont nous aurons 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. Elle doit 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 doit être capable d'exécuter en parallèle des processus sécurisés et des processus normaux. Le système d'exploitation n'a pas besoin d'être sécurisé, ni même de confiance et peut être malicieux.

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. En particulier, nous ne considérerons pas les attaques temporelles [11], les attaques par mesure de la consommation électrique (dpa [12]), etc.

De plus, nous ne considérerons 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 systèmes 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.

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

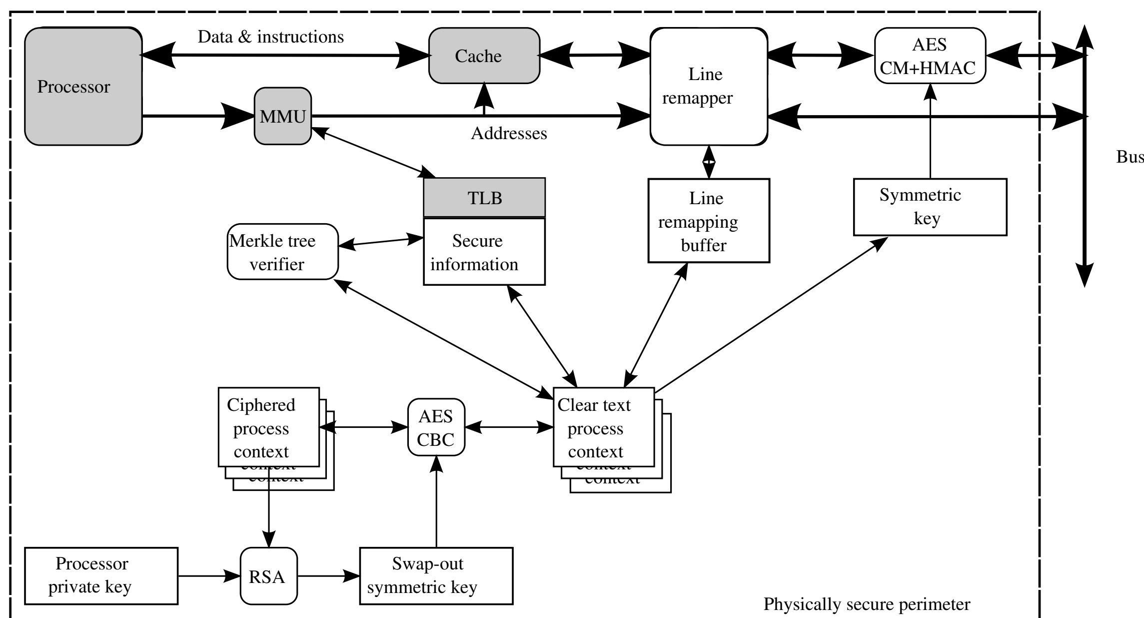


FIG. 1: Architecture globale résumée de CRYPTOPAGE.

2.2 L'infrastructure Hide

Notre proposition est partiellement basée sur l'infrastructure HIDE que nous décrirons brièvement ici. Cette infrastructure, présentée dans [23], 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 chiffrées à nouveau. 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, [23] propose 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 [23]).

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 supposerons 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). Nous utiliserons également les notations suivantes :

- $\|$ représente la concaténation de deux chaînes de bits et $\oplus$ l'opération *ou exclusif* bit-à-bit (XOR) ;
- $L_{a,c} = L_{a,c}^{(0)}\| \dots \|L_{a,c}^{(l-1)}$: le contenu de la ligne de cache numéro a dans le bloc 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_i et K_d : les clés symétriques utilisées pour chiffrer respectivement le code et les données d'un processus sécurisé. Par souci de simplification, dans la suite de l'article, K_e désignera, suivant le cas (code ou données), K_i ou K_d.
- K_m : la clé symétrique utilisée pour calculer le code d'authentification de message (MAC, *Message Authentication Code*) utilisé pour authentifier le code et les données du processus.

Les clés K_i, K_d et K_m sont propres à un processus sécurisé donné et sont stockées de manière sécurisée dans le contexte matériel de ce dernier.

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.

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 du bloc et p le numéro de la permutation), et les stocke avec les autres informations liées au bloc (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 $C_{a,c}\|H_{a,c}$ avec :

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

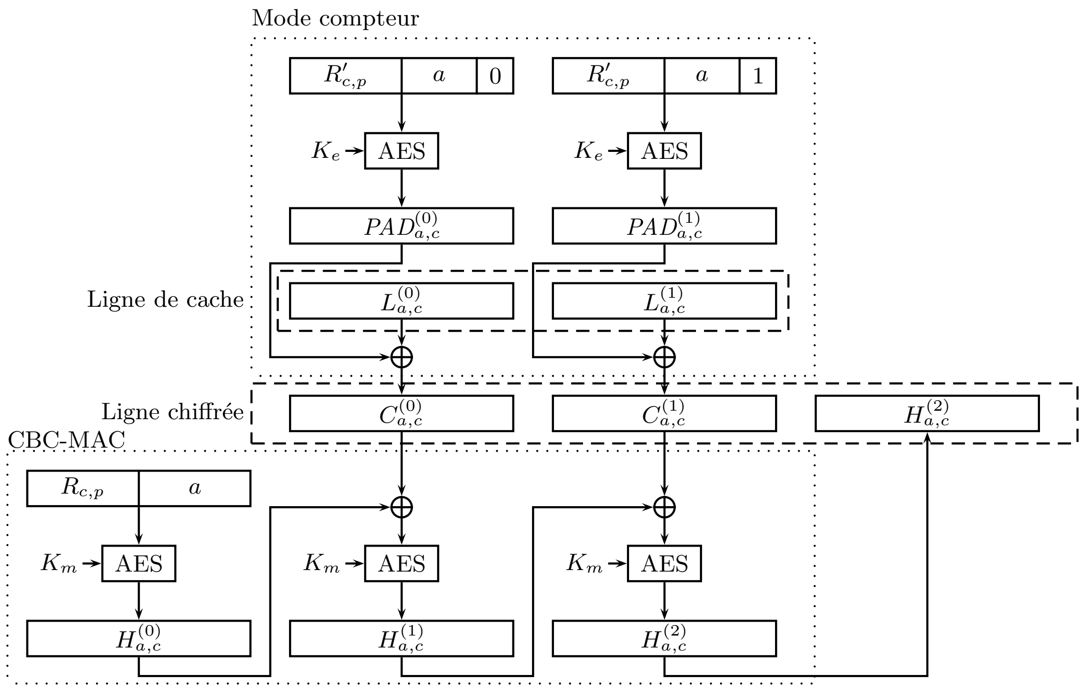
$$C_{a,c}^{(i)} = L_{a,c}^{(i)} \oplus PAD_{a,c}^{(i)} \tag{2}$$

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

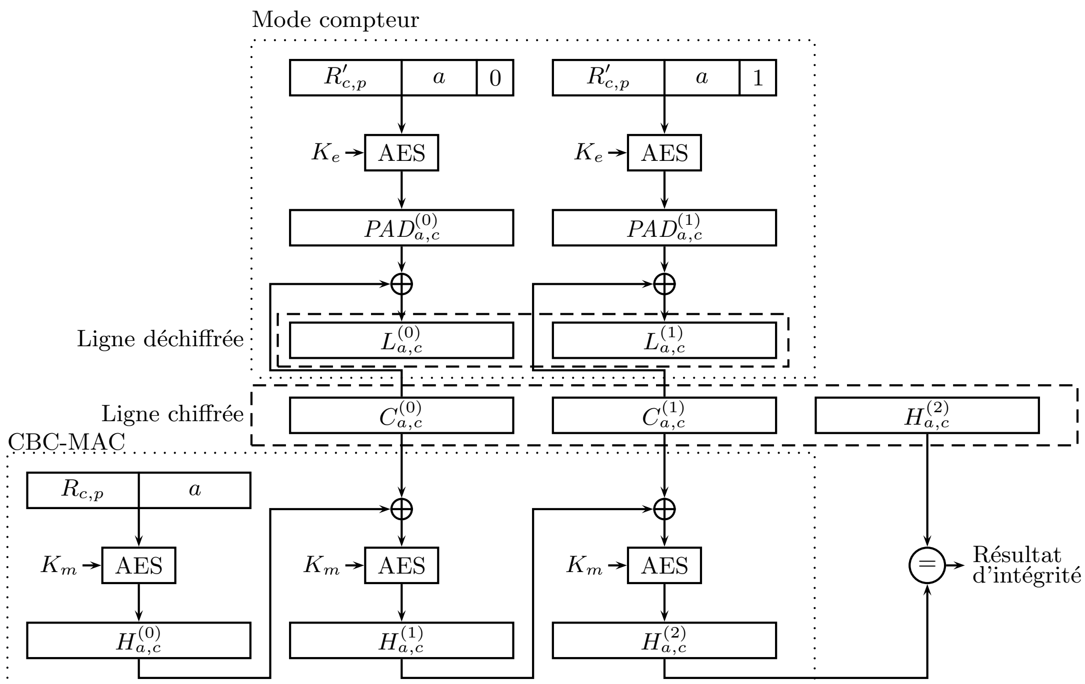
$$H_{a,c} = H_{a,c}^{(l)} \tag{4}$$

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

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



(a) Chiffrement



(b) Déchiffrement

Fig. 2: Les opérations de chiffrement et de déchiffrement d'une ligne de cache.

Une ligne de cache est chiffrée en utilisant le mode compteur [19] (équation 2). Les masques (*pads*) utilisés dépendent de $R'_{c,p}$ et de a (équation 3). Les équations de 4 à 6 définissent un CBC-MAC [18][1] 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.

Le mode compteur est sûr [2] à condition que le compteur ne soit utilisé qu'une seule fois avec la même clé [2]. Entre deux permutations, la ligne de cache a n'est chiffrée qu'une seule fois [3] 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 [4], 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. Bellare *et al.* ont montré dans [3] 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.

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 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, 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

[1] 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.

[2] 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.

[3] En effet, avec Hide, entre deux permutations, une ligne est lue et écrite au plus une fois en mémoire.

[4] 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.

ligne. Enfin, si un attaquant ne peut pas rejouer $R_{c,p}$, il n'est pas en mesure de monter une attaque par rejeu [5] car le MAC dépend de $R_{c,p}$ dont la valeur change à chaque permutation.

Protection des informations sur les blocs 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 différents blocs ($R_{c,p}$, $R'_{c,p}$, la table de permutation, etc.) à l'aide d'un arbre de MERKLE [17].

Le principe est de construire un arbre de hachage dont les feuilles sont les 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 (*Translation Lookaside Buffer*) par exemple), il vérifie le contenu des nœuds jusqu'à la racine.

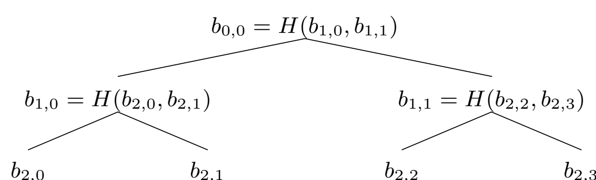


FIG. 3: Arbre de MERKLE.

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 tables 1a et 1b [13,6], 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 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 forcément correct.

Les structures de données contenant les informations sur les pages mémoire doivent également être chiffrées afin d'empêcher un attaquant d'accéder à la table de permutation, ce qui rendrait

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

```
R_V(i, j) :
  b_{i,j} = R(i, j)
  tant que i > 0 :
    f = j ⊕ 1 ; p = ⌊j/2⌋
    b_{i,f} = R(i, f)            { Lecture du frère }
    b_{i−1,p} = R(i − 1, p)      { Lecture du père }
    si b_{i−1,p} ≠ H(b_{i,min(j,f)}, b_{i,max(j,f)}) :
      erreur
    i = i − 1 ; j = p            { On remonte }
  renvoie b_{i,j}
```

(a) Lecture vérifiée

```
W_V(b_{i,j}, i, j) :
  si i > 0 :
    f = j ⊕ 1 ; p = ⌊j/2⌋
    b_{i,f} = R_V(i, f)          { Lecture et vérification du frère }
    b_{i−1,p} = H(b_{i,min(j,f)}, b_{i,max(j,f)})
    W_V(b_{i−1,p}, i − 1, p)     { Écriture et vérification du père }
  W(b_{i,j}, i, j)               { Écriture du nœud }
```

(b) Écriture vérifiée

TAB. 1: Algorithmes de lecture et d'écriture vérifiées.

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.

2.4 Gestion des données d'authentification

Comme nous l'avons vu dans la section 2.3, pour chaque ligne de cache ($C_{a,c}$) stockée en mémoire, une valeur d'authentification ($H_{a,c}$) est également stockée.

Afin de gérer le stockage supplémentaire de ces informations de façon transparente pour le système d'exploitation et pour les applications, l'unité de gestion de la mémoire du processeur (MMU, *Memory Management Unit*) 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 [6].

2.5 Gestion déléguée de l'arbre de Merkle

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 $n = \log_2(\textit{nombre maximum de pages})$).

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 ne sont pas déjà disponibles 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 de la table 2. 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). L'algorithme de la table 2 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.

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 $d > 0$:
    {*L'*OS *trouve l'adresse du nœud de l'arbre selon son bon plaisir :*}
    $A_{d,q} = getNodeAddress(d, q)$
    { *Demande au processeur de charger un nœud dans le* VB *:* }
    `LoadNode` $A_{d,q}, d, q$
    si $b_{d,q}$ est déjà dans le cache :
      {*On a trouvé un nœud déjà certifié en cache : on a cerné* }
      break
    {*Remonte dans l'arbre :*}
    $d = d - 1; q = \lfloor \frac{q}{2} \rfloor$
  {*Redescend sur le premier nœud qui manque :* }
  $d = d + 1$
  { *Vérifie en descendant l'éventuelle branche manquante :* }
  tant que $d \leq n$:
    {*Demande processeur vérifier nœud dans* VB *et si correct bascule dans cache certifié :* }
    `HashCheck` d
    {*Descend sur le nœud suivant qui manque :* }
    $d = d + 1$
  {*Reprend l'exécution. Si attaque, le* TLB *manquera : double faute et déni de service détecté* }
  `ReturnFromInterrupt`

Tab. 2: Vérification déléguée à l'OS des données d'une page mémoire en utilisant 2 nouvelles instructions sécurisées.

2.6 Autres aspects liés à la sécurité

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 articles [21,15,14,5,23].

2.7 Applications

Un certain nombre d'applications peuvent bénéficier des propriétés de sécurité offertes par l'architecture CryptoPage.

Grilles de calcul Dans le domaine des grilles de calcul, jusqu'à présent, de nombreux travaux ont été menés pour protéger les nœuds de la grille contre une application malveillante (grâce à des techniques de bac à sable ou de virtualisation) et pour mettre en place des mécanismes d'authentification et d'autorisation pour les utilisateurs. Cependant, peu de travaux ont concerné la sécurisation d'une application distribuée contre des nœuds malveillants.

Les algorithmes utilisés et les résultats produits par une application s'exécutant sur la grille peuvent être de grande valeur et peuvent nécessiter de rester confidentiels, ce qui est gênant puisque l'on veut distribuer l'application sur un nombre important de nœuds que l'on ne contrôle pas directement. Jusqu'à maintenant, la confidentialité d'une application ou de ses résultats est principalement réalisée via des techniques d'obscurcissement de code. Cependant, aussi complexes soient-elles, ces techniques ne peuvent pas résister à un attaquant réellement motivé. Si chaque nœud de la grille dispose d'un processeur de type CryptoPage, l'application peut être chiffrée et ainsi, le processeur sécurisé va garantir sa confidentialité.

Il est également important de garantir l'intégrité des applications s'exécutant sur la grille. Il est possible d'exécuter plusieurs fois l'application sur différents nœuds et de comparer les résultats en espérant que tous les nœuds ne sont pas corrompus. Cependant, cette technique est pénalisante d'un point de vue performance. L'idéal serait que chaque nœud de la grille soit équipé d'un processeur de type CryptoPage qui garantirait la bonne exécution de l'application.

Anti-virus Les processus sécurisés sont bien protégés contre une modification par un virus. Tout d'abord, le code et les données d'un processus sécurisé sont chiffrés avec deux clés symétriques différentes (K_d et K_i) ce qui rend l'injection de code (via un débordement de tampon par exemple) difficile (le code injecté serait écrit en mémoire comme une donnée et donc chiffré avec K_d et relu en tant que code et donc déchiffré à l'aide de la clé K_i). De plus, une application (ou le système d'exploitation) ne peut pas corrompre l'exécutable d'une application sécurisée car ce dernier est chiffré avec des clés inconnues de l'attaquant et dispose d'un mécanisme de protection d'intégrité.

Ainsi, un logiciel anti-virus, s'il s'exécute en tant que processus sécurisé ne pourra pas être corrompu par une attaque.

Cependant, on peut noter que si le virus est présent dans le code d'une application sécurisée avant son chiffrement, ou que l'application sécurisée elle-même est un virus ou un cheval de Troie, le virus s'exécutera en tant que processus sécurisé et ne pourra ainsi ni être corrompu, ni même être observé (excepté via ses effets de bord : ouvertures de sockets réseaux par exemple), ce qui rend la tâche d'un programme anti-virus plus compliquée...

3 Évaluation et résultats

Dans cette section, nous utiliserons les mêmes paramètres architecturaux que dans [23] pour évaluer l'impact de notre proposition. Ces paramètres sont résumés dans la table 3.

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

Tab. 3: Paramètres architecturaux.

3.1 Analyse théorique

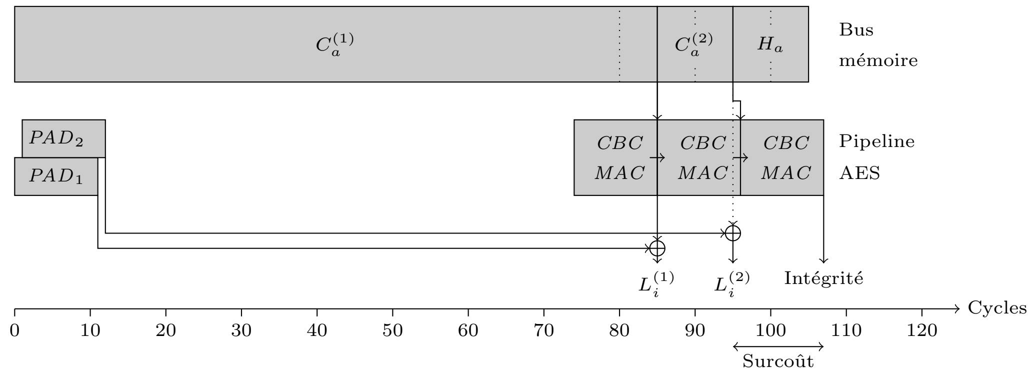


Fig. 4: Exemple du chargement d'une ligne de cache depuis la mémoire.

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 en 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 terme 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[6]. 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 terme 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 %.

3.2 Évaluation

Afin d'évaluer nos propositions, nous avons exécuté plusieurs programmes étalons de la suite SPEC2000int [9] avec le simulateur de micro-architecture SimpleScalar [1], modifié pour simuler notre architecture. Afin de réduire le temps nécessaire pour réaliser ces simulations, on réalise une simulation détaillée sur seulement 200 millions d'instructions après avoir passé rapidement le premier milliard et demi d'instructions.

Sur la figure 5a, on compare le nombre d'instructions exécutées par cycle d'horloge (IPC) pour chacun des programmes étalons sur trois implémentations différentes :

- notre propre implémentation de l'infrastructure HIDE [23] (blocs de 8 ko, tous les blocs protégés, pas d'optimisation de placement) ;
- une implémentation basique de notre architecture (sans le cache d'arbre de MERKLE, les instructions devant être vérifiées avant de pouvoir être exécutées) ;
- une implémentation avancée de notre architecture (avec un cache d'arbre de MERKLE pleinement associatif de 512 entrées et une exécution spéculative des instructions pendant la vérification d'intégrité).

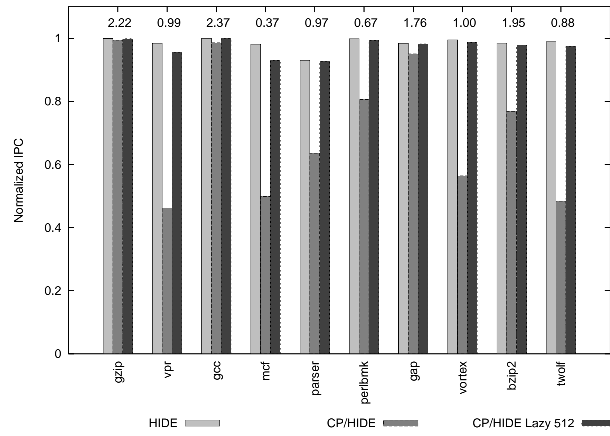
Tous les résultats sont normalisés à leur valeur originale donnée par le simulateur sur une architecture normale non sécurisée.

Notre implémentation basique donne de mauvais résultats (jusqu'à 50 % de ralentissement sur certains programmes étalons). Ceci est en partie lié au coût élevé de la vérification complète de l'arbre de MERKLE à chaque défaut de TLB. La version avancée de notre architecture donne quant à elle de bons résultats. Le ralentissement moyen n'est que de 3 % et le plus mauvais est de 7,4 %.

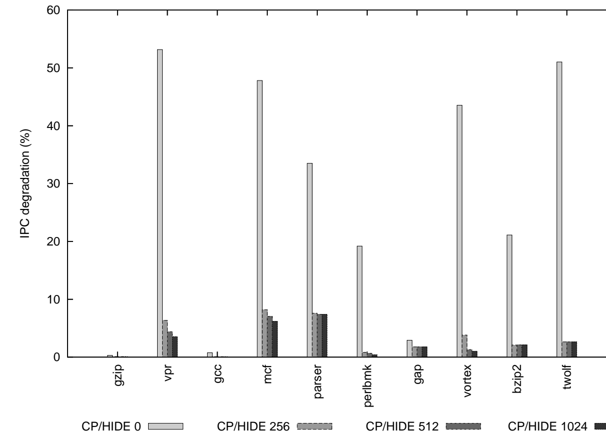
La figure 5a compare l'effet de différentes tailles du cache d'arbre de MERKLE : sans cache et un cache de 256, 512 ou 1024 entrées. Nous constatons que l'introduction d'un cache, même de petite taille, a un impact très positif sur les performances. Cependant, l'augmentation de la taille du cache a seulement un impact mineur à cause du grand nombre de pages utilisées par ces applications, qui est largement supérieur au nombre d'entrées dans le cache. La solution idéale serait un cache suffisamment grand pour stocker l'ensemble des informations sur les pages ainsi que tous les nœuds de l'arbre de MERKLE.

Le stockage des MAC augmente l'empreinte mémoire d'un processus sécurisé de 50 % avec nos paramètres. Afin de la réduire, les MAC peuvent être calculés sur plusieurs lignes de cache au lieu d'une seule, au détriment du temps nécessaire pour les vérifier et de l'utilisation du bus mémoire. La

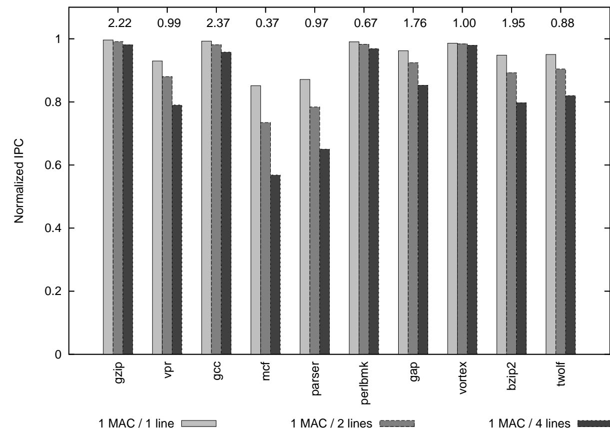
[6] Le calcul d'un résumé en utilisant l'algorithme SHA-1 prend 80 cycles.



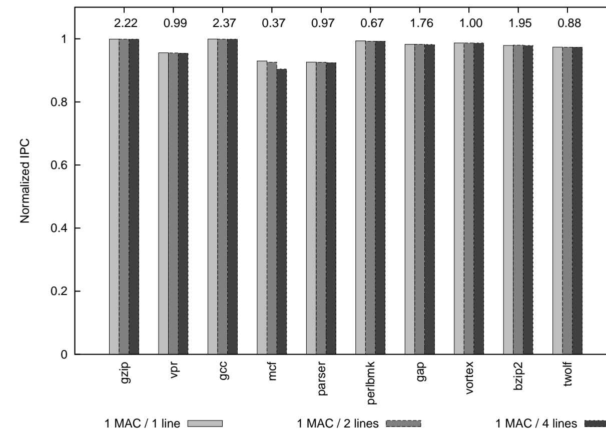
(a) Comparaison de trois architectures.



(b) Comparaison pour différentes tailles de cache d'arbre de Merkle.



(c) Performances pour divers ratios lignes de cache/MAC.



(d) Performances pour divers ratios lignes de cache/MAC (exécution spéculative).

Fig. 5: Résultats des simulations.

figure 5c montre l'impact du calcul des MAC sur une, deux et quatre lignes de cache, sans exécution spéculative des instructions avant leur vérification (basé sur une architecture CryptoPage avec un cache d'arbre de Merkle à 512 entrées). La dégradation moyenne des performances est de l'ordre de 4,4 % pour un calcul sur deux lignes et 11,7 % pour quatre lignes. Si les instructions sont exécutées en parallèle à leur vérification, la dégradation des performances est très faible (moins de 1 %, comme le montre la figure 5d).

4 Travaux similaires du domaine

Dans cet article, nous avons proposé un mécanisme permettant d'implémenter un vérificateur mémoire en ligne en utilisant l'infrastructure Hide. Dans [7], Gassend *et al.* ont utilisé un arbre de Merkle, 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 terme 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 l'intégralité de la mémoire) afin de construire un vérificateur mémoire, résistant aux attaques par rejeu, et avec de meilleures performances.

Gassend *et al.*, dans [7] ont également proposé un vérificateur mémoire hors ligne utilisant des fonctions de hachage incrémentales. Dans [4], Clarke *et al.* ont proposé un autre vérificateur mémoire hors ligne avec un surcoût en terme de bande passante mémoire constant en combinant des arbres de Merkle et des fonctions de hachage incrémentales. Les vérificateurs mémoire hors-ligne sont plus rapides [7] 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.

Le chiffrement mémoire a également été étudié dans plusieurs architectures comme Xom [14,15,16], Aegis [21,22] et CryptoPage [13,6,5]. Dans [20], Shi *et al.* ont également proposé d'utiliser le mode compteur afin de chiffrer les lignes de cache. 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.

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. Goldreich *et al.*, dans [8], ont proposé plusieurs approches pour garantir qu'aucune fuite d'informations ne peut se produire sur le bus d'adresse mais leurs algorithmes réduisent considérablement les performances.

---

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

5 Conclusions et travaux futurs

Dans cet article, nous avons décrit une solution pour implémenter, à faible coût, 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 deux nouvelles instructions de sécurité.

L'impact sur les performances de ces mécanismes est de seulement 3 %, 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.

Nous travaillons actuellement à étendre nos propositions à des processeurs multicœurs et à des systèmes multiprocesseurs ainsi qu'à intégrer ses mécanismes dans Linux. D'autre part, nous travaillons dans le cadre du projet safescale à l'utilisation de tels systèmes pour effectuer du calcul distribué haute performance de confiance.

6 Remerciements

Ce travail est soutenu par une bourse de thèse de la Délégation Générale pour l'Armement (dga), par le projet anr ara/ssia safescale et par le projet du get (Groupement des Écoles de Télécommunications) tcp (Trusted Computing Platform). Merci à Jacques Stern pour les discussions sur le projet et cette thèse, aux collègues de l'enst de tcp (en particulier Sylvain Guilley et Renaud Pacalet) et du projet safescale pour leurs nombreuses interactions.

Références

1. Austin, T., Larson, E., Ernst, D. : SimpleScalar : An Infrastructure for Computer System Modeling. *Computer*, 35 (2), IEEE Computer Society Press, pp. 59–67 (2002).
2. Bellare, M., Desai, A., Jokipii, E., Rogaway, P. : A Concrete Security Treatment of Symmetric Encryption. In Proceedings of the 38th Annual Symposium on Foundations of Computer Science (Focs'97), IEEE Computer Society, pp. 394–403 (1997).
3. Bellare, M., Namprempre, C. : Authenticated Encryption : Relations among Notions and Analysis of the Generic Composition Paradigm. In Advances in Cryptology - Asiacrypt 2000 Proceedings, LNCS 1976, Springer-Verlag (2000).
4. Clarke, D., Suh, G. E., Gassend, B., Sudan, A., van Dijk, M., Devadas, S. : Towards Constant Bandwidth Overhead Integrity Checking of Untrusted Data. In Proceedings of the 2005 IEEE Symposium on Security and Privacy, IEEE Computer Society, pp. 139–153 (2005).
5. Duc, G. : 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` (2004).
6. Duc, G., Keryell, R., Lauradoux, C. : CryptoPage : Support matériel pour cryptoprocessus, *Technique et Science Informatiques*, 24, pp. 667–701 (2005).
7. Gassend, B., Suh, G. E., Clarke, D., van Dijk, M., Devadas, S. : Caches and Hash Trees for Efficient Memory Integrity Verification. In Proceedings of the 9th International Symposium on High-Performance Computer Architecture (Hpca'03), pp. 295–306 (2003).

8. Goldreich, O., Ostrovsky, R. : Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43 (3), ACM Press, pp. 431–473 (1996).
9. Henning, J. L. : SPEC CPU2000 : measuring CPU performance in the New Millennium. *IEEE Computer*, 33 (7), pp. 28–35 (2000).
10. Huang, A. : Keeping Secrets in Hardware : the Microsoft XBox (TM) Case Study. Technical Report AI Memo 2002-008, MIT (2002).
11. Kocher, P. C. : Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, LNCS 1109, pp. 104–113, Springer Verlag (1996).
12. Kocher, P. C., Jaffe, J., Jun. B. : Differential Power Analysis. In Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (Crypto'99), LNCS 1666, pp. 388–397, Springer-Verlag (1999).
13. Lauradoux, C., Keryell, R. : CryptoPage-2 : un processeur sécurisé contre le rejeu. In Symposium en Architecture et Adéquation Algorithme Architecture (SympAAA'2003), La Colle sur Loup, France, pp. 314–321 (2003).
14. Lie, D., Trekkath, C. A., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M. : Architectural support for copy and tamper resistant software. In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (Asplos IX), pp. 168–177 (2000).
15. Lie, D., Trekkath, C. A., Horowitz, M. : Implementing an Untrusted Operating System on Trusted Hardware. In Proceedings of the 9th ACM Symposium on Operating Systems Principles (Sosp'03), pp. 178–192 (2003).
16. Lie, D. : *Architectural support for copy and tamper-resistant software*. PhD Thesis, Stanford University (2004).
17. Merkle, R. C. : A Certified Digital Signature. In Proceedings on Advanced in Cryptology (Crypto'89), LNCS 435, pp. 218–238, Springer-Verlag (1989).
18. NIST : Computer Data Authentication. Federal Information Processing Standards Publication 113, (1985).
19. NIST : Recommendation for Block Cipher Modes of Operation, Special Publication 800-38A (2001).
20. Shi, W., Lee, H. S., Lu, C., Ghosh, M., Lu, C., Boldyreva, A. : High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In Proceedings of the 32nd Annual International Symposium on Computer Architecture (Isca'05), IEEE Computer Society, pp. 14–24 (2005).
21. Suh, G. E., Clarke D., Gassend, B., van Dijk, M., Devadas, S. : Aegis : Architecture for Tamper-Evident and Tamper-Resistant Processing. In Proceedings of the 17th International Conference on Supercomputing (Ics'03), pp. 160–171 (2003).
22. Suh, G. E., O'Donnell, C. W., Sachdev, I., Devadas, S. : Design and Implementation of the Aegis Single-Chip Secure Processor Using Physical Random Functions. In Proceedings of the 32nd Annual International Symposium on Computer Architecture (Isca'05), IEEE Computer Society, pp. 25–36 (2005).
23. Zhuang, X., Zhang, T., Pande, P. : Hide : an infrastructure for efficiently protecting information leakage on the address bus. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (Asplos-XI), Acm Press, pp. 72–84 (2004).