Portage de l'architecture sécurisée CryptoPage sur un microprocesseur x86

Guillaume Duc et Ronan Keryell*

14 décembre 2004

Résumé

Les ordinateurs actuels ne sont pas aussi sécurisés que leur développement ubiquitaire et leur interconnexion le nécessiteraient. En particulier, on ne peut même pas garantir l'exécution sécurisée et confidentielle d'un programme face à un attaquant logiciel (l'administrateur système) ou matériel (analyseur logique sur les bus, contrôle des ressources de l'ordinateur,...).

Dans cet article on présente le rajout à un processeur standard $x86$ de PC les mécanismes de sécurité du processeur conceptuel CryptoPage consistant à rajouter du chiffrement fort sur les bus externes. Afin de prouver que les accès du processeur à sa mémoire extérieure sont globalement corrects et résister aux attaques par rejeu, on rajoute au niveau du cache, en plus d'un chiffrement des lignes, un vérificateur utilisant un arbre de hachage de Merkle stocké lui-même en cache afin d'en accélérer le calcul et d'impacter *a minima* les performances par rapport à un système non sécurisé.

Enfin, est abordée l'interaction avec un système d'exploitation non sécurisé et la gestion sécurisée des interruptions logicielles (signaux) pour permettre l'exécution la plus transparente possible et sécurisée de processus. Le tout est réalisé dans le simulateur *Bochs* pour faire tourner un Linux adapté.

Mots clés : Sécurité informatique, informatique de confiance, cryptoprocesseurs, cryptographie, processus sécurisés.

1 Introduction

Un certain nombre d'architectures matérielles permettant d'exécuter de façon sécurisée des processus ont été développées ces dernières années. On peut notamment citer Aegis [13], xom [9] ou CryptoPage [5, 8]. Dans toutes ces architectures, des mécanismes cryptographiques de chiffrement et de vérification d'intégrité sont intégrés directement à l'intérieur du processeur pour garantir la sécurité des applications.

Dans ce papier nous allons décrire comment modifier un processeur de la famille Intel x86 (comme le Pentium) afin de le transformer en processeur sécurisé de type CryptoPage [3]. Dans la section 2 nous décrirons les objectifs à atteindre pour l'architecture CryptoPage/x86, puis dans la section 3 les mécanismes à mettre en œuvre pour assurer la propriété de confidentialité et enfin dans la section 4 un aperçu des mécanismes nécessaires à la protection de l'intégrité d'un processus sécurisés.

*Laboratoire Informatique et Télécommunications, ENST Bretagne, CS 83818, 29238 Brest Cedex — France

2 Objectifs de l'architecture

Nous allons décrire les objectifs de sécurité pour l'architecture CryptoPage/x86 ainsi que les moyens dont va disposer un éventuel attaquant.

2.1 Objectifs

L'objectif de l'architecture CryptoPage/x86 est de permettre l'exécution de processus sécurisés, c'est-à-dire de processus pour lesquels les deux propriétés suivantes doivent être garanties :

- la *confidentialité* que l'on peut énoncer ainsi : un adversaire doit pouvoir obtenir le moins d'information possible sur le code du processus et sur les données manipulées par ce dernier ;
- la propriété d'*intégrité* : la bonne exécution d'un processus ne peut pas être altérée par une attaque. Si une attaque est détectée, le processeur doit arrêter le processus.

2.2 Moyens de l'attaquant

Pour la conception de l'architecture, on considère que l'attaquant a les moyens de contrôler tout ce qui se trouve à l'extérieur de la puce du processeur (en particulier le bus mémoire). L'attaquant n'a pas de moyen de sonder directement ou indirectement l'intérieur même du processeur. On considère que les attaques temporelles [6], DPA [7], etc. sont évitées par d'autres moyens sortant du cadre de cet article.

Nous ne considérerons également pas les attaques par déni de service étant donné que celles-ci sont inévitables (il suffit que l'attaquant n'alimente pas en électricité le processeur).

3 Confidentialité

Dans cette section, nous allons décrire progressivement les mécanismes mis en place dans CryptoPage/x86 afin de garantir la propriété de confidentialité durant les trois phases de la vie d'un processus sécurisé : son chargement, son exécution et ses interruptions (ce qui inclue les appels systèmes et la fin de vie du processus).

3.1 Durant l'exécution d'un processus

Tout d'abord, afin d'assurer la propriété de confidentialité durant l'exécution d'un processus, les instructions et les données manipulées par ce dernier vont être chiffrées. Sous l'hypothèse que tout ce qui se trouve sur la puce même du processeur est protégé contre des attaques, nous pouvons implémenter les mécanismes de chiffrement entre le bus mémoire et le cache de plus haut niveau interne au processeur. Les instructions et les données situées dans les différents caches du processeur ne sont pas chiffrées. Ainsi les opérations de chiffrement et de déchiffrement n'interviendront que lors d'un défaut de cache mais aucune pénalité n'est rajoutée lors des accès à des données déjà situées en cache.

3.1.1 Algorithme et mode de chiffrement

Comme l'unité de base de transfert entre la mémoire et le processeur est la ligne de cache, on applique également le chiffrement au niveau d'une ligne de cache. Pour chiffrer, on utilise

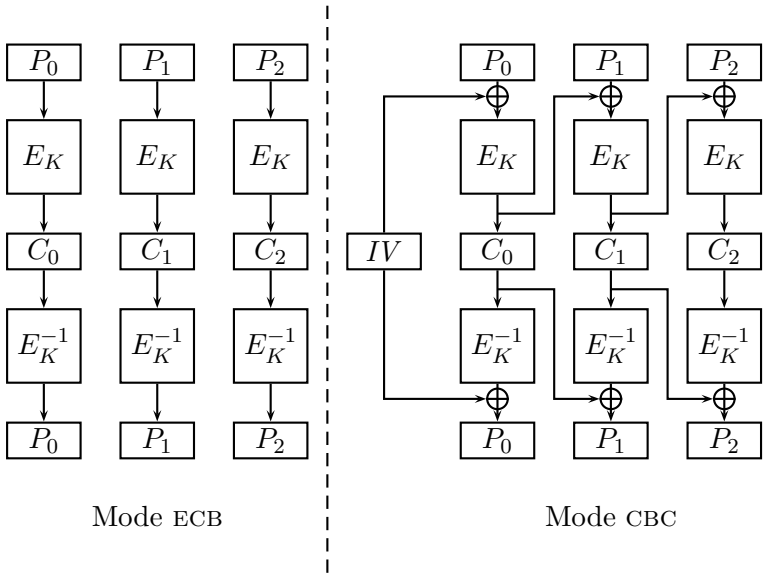

FIG. 1 – Modes de chiffrement ECB et CBC.

une algorithme de chiffrement symétrique tel qu'AES [10], qui réalise ses opérations sur des blocs de 16 octets en utilisant une clé de taille 128, 192 ou 256 bits, en mode *Cipher Block Chaining* (CBC) [11]. Dans ce mode (cf. figure 1), un bloc chiffré dépend non seulement du bloc en clair et de la clé mais également du bloc chiffré précédent (ou d'un *Vecteur d'Initialisation* (IV) choisi aléatoirement pour le premier bloc chiffré). L'avantage de ce mode sur le mode *Electronic CodeBook* (ECB) est que deux lignes de cache identiques seront chiffrées en deux lignes différentes (pour peu que les IV soient différents pour les deux lignes). Ainsi on diminue la possibilité de l'adversaire de découvrir des motifs récurrents en mémoire (par exemple les zones remplies de zéros).

3.1.2 Clés utilisées

Pour isoler chaque processus les uns des autres mais également pour empêcher qu'une erreur de programmation n'amène le programme à s'écrire lui-même sur une sortie non chiffrée, chaque processus dispose de deux clés symétriques, une dédiée au chiffrement des instructions (K_i) et une dédiée au chiffrement des données manipulées (K_d).

3.1.3 Stockage des iv

L'IV, qui n'a pas besoin d'être confidentiel et dont la taille est égale à celle de la taille d'un bloc de l'algorithme utilisé, doit être stocké en mémoire. Nous proposons de le stocker directement avec la ligne de cache correspondante et de modifier le mécanisme de calcul d'adresse (au sein de la *Memory Management Unit* (MMU)) afin que ce stockage soit transparent pour le système d'exploitation et pour l'application.

Pour réduire les modifications au niveau du système d'exploitation, la taille des pages mémoires manipulées par ce dernier ne doit pas être affectée par ces données supplémentaires. Au niveau de l'application, les adresses de deux lignes de cache contiguës doivent rester

contiguës, même si physiquement elles sont séparées du fait du stockage des IV utilisées par ces deux lignes.

Nous proposons donc la solution suivante. Nous modifions les fonctions d'allocation mémoire du système pour que lorsque l'application demande de la mémoire sécurisée, elles allouent assez de mémoire pour stocker les données ainsi que les IV correspondant. Ensuite, lorsque l'application accède à ces données de façon sécurisée, l'adresse virtuelle est modifiée par la MMU du processeur pour prendre en compte l'emplacement exact des informations.

Par exemple, si l'on considère qu'un IV a une taille S_{IV} (par exemple 16 octets pour AES), qu'une ligne de cache a une taille S_L (par exemple 32 octets pour x86) et que l'on stocke l'IV à la suite de la ligne de cache correspondante, l'adresse virtuelle A_v du début d'une ligne de cache en mémoire est obtenue simplement à partir de l'adresse virtuelle interne A_{vi} du début de la ligne de cache grâce à la formule suivante :

$$A_v = \frac{S_{IV} + S_L}{S_L} A_{vi}$$

Cette translation d'adresse est uniquement appliquée lorsque le processus tourne en mode sécurisé. Ainsi, le système d'exploitation a accès à la mémoire sans ce mécanisme et peux donc ainsi continuer à la gérer sans même savoir que les pages qu'il manipule contiennent des données et des IV.

3.1.4 Optimisations

Lorsqu'une ligne de cache est lue depuis la mémoire, elle doit être déchiffrée (et vérifiée) avant d'être disponible dans le cache du processeur. Cette opération ajoute donc une latence supplémentaire en cas de défaut de cache. Cependant, certaines optimisations sont possibles.

Tout d'abord, le déchiffrement en mode CBC peut être parallélisé (cf. figure 1). En effet, il suffit pour déchiffrer un bloc d'avoir le bloc chiffré et le bloc chiffré précédent (ou l'IV s'il s'agit du premier bloc). Les opérations de déchiffrement des différents blocs peuvent également être pipelinées sur un seul AES. Il existe des réalisations pipelinées de l'AES ne nécessitant que 11 cycles d'horloge [12], ce qui est relativement faible par rapport au coût de l'accès mémoire dans le cas des processeurs rapides actuels (proche de 150 cycles pour un processeur à 2 GHz par exemple).

Lors de l'écriture d'une ligne de cache vers la mémoire, elle doit tout d'abord être chiffrée. Cette étape ajoute également une latence supplémentaire. Cependant, contrairement au déchiffrement en mode CBC, le chiffrement ne peut pas être parallélisé à cause de la position de la rétro-action (cf. figure 1) qui rajoute une dépendance vraie. Néanmoins, on peut profiter d'un pipeline pour traiter plusieurs chiffrement de lignes de cache en parallèle. De plus dans ce cas, si une ligne de cache qui vient d'être effacée du cache est de nouveau nécessaire, elle peut être récupérée sans coût dans le pipeline avant qu'elle ne soit réellement écrite en mémoire, comme cela était fait dans le système mémoire de l'IBM 360/91 [14].

Des simulations sont en cours de réalisation afin de mesurer plus précisément l'impact de ces latences supplémentaires sur les performances globales de l'architecture faisant tourner un système d'exploitation et des applications.

3.2 Durant l'interruption d'un processus

Avec le chiffrement proposé précédemment, la confidentialité est assurée pendant que le processus sécurisé s'exécute. Intéressons-nous maintenant au moment où le processus est

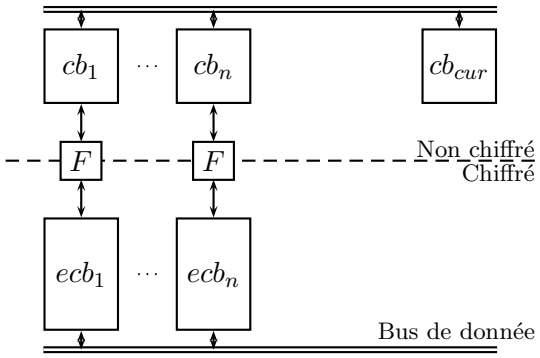

FIG. 2 – Les tampons de contextes matériels.

interrompu.

3.2.1 Interruptions et exceptions

Lorsque le processus sécurisé s'exécute, il peut être interrompu par une interruption ou une exception. Dans le cas normal, le processeur lance l'exécution d'un gestionnaire d'interruption, dont l'adresse est stockée dans une table de vecteurs d'interruption, et ensuite le processeur et/ou le système d'exploitation se charge de sauvegarder le contexte matériel du processus (l'ensemble des registres utilisés par le processus) afin de pouvoir restaurer son exécution plus tard. Cependant, afin de garantir la propriété de confidentialité, le système d'exploitation ne doit pas pouvoir accéder et/ou modifier le contexte matériel d'un processus sécurisé tout en étant quand même capable de sauvegarder ce dernier et de lancer le contexte d'un autre processus.

Nous proposons la solution suivante. Lors d'une interruption ou d'une exception, le contexte matériel (qui contient les valeurs de tous les registres généraux, flottants, des sélecteurs de segments, des registres spécialisés, etc.) du processus courant est automatiquement copié dans un tampon sécurisé, appelé tampon de contexte courant (cb_{cur}), sur la puce du processeur, non accessible directement de façon logicielle. Les registres généraux du processeur sont effacés et la procédure d'interruption ou d'exception continue normalement. Ainsi, le système d'exploitation n'a pas accès au contenu des registres manipulés par le programme.

Pour relancer un processus à partir d'un contexte matériel stocké dans le tampon de contexte courant cb_{cur}, le système d'exploitation exécute une instruction spécialisée (cf. table 2) qui relance l'exécution du processus à partir du contexte matériel situé dans cb_{cur}. Ainsi, si le système d'exploitation ne peux pas voir ou modifier le contexte durant l'interruption (puisqu'il est stocké dans cb_{cur}), la propriété de confidentialité est garantie.

Cependant, si plusieurs processus sécurisés tournent en même temps, le système d'exploitation doit pouvoir sauvegarder et restaurer le contenu du tampon courant cb_{cur}. Afin de permettre de changer rapidement de processus, l'architecture CRYPTOPAGE possède un certain nombre d'autres tampons cb_i pouvant contenir les contextes matériels. Une instruction spécialisée permet de copier le contenu d'un tampon dans un autre (potentiellement le tampon courant). Chacun de ces tampons est constitué d'une partie non chiffrée et d'une partie chiffrée. Le système d'exploitation dispose de deux instructions permettant de sauvegarder le

contenu de la partie chiffrée d'un tampon vers la mémoire ou d'en charger le contenu depuis la mémoire. Il peut également contrôler le chiffrement du contenu de la partie non chiffrée de l'un des tampons vers la partie chiffrée et/ou le déchiffrement et la vérification du contenu de la partie chiffrée vers la partie non chiffrée à l'aide d'un registre de contrôle spécialisé.

Ainsi, le système d'exploitation peut gérer de façon efficace les changements de contexte entre les processus sécurisés :

- si un seul processus sécurisé s'exécute, le système d'exploitation peut laisser son contexte dans le tampon courant pour qu'il soit restauré immédiatement lors de sa reprise ;
- si un nombre limité de processus sécurisés s'exécutent, le système d'exploitation stocke leur contexte matériel dans les différents tampons et évite ainsi le coût du chiffrement et du déchiffrement de ceux-ci ;
- si beaucoup de processus sécurisés s'exécutent, il doit gérer la sauvegarde des contextes chiffrés en mémoire et leur restauration mais peut optimiser son ordonnanceur de tâche pour qu'il privilégie les tâches dont le contexte matériel est déjà déchiffré dans un tampon.

La partie chiffrée d'un tampon de contexte matériel est composée de :

- $IV_1 \| E_{K_p}(IV_1, K_d)$, c'est-à-dire la concaténation d'un IV aléatoire et de la clé de chiffrement des données K_d du processus chiffrée avec la clé symétrique K_p du processeur et l'IV ;
- $IV_2 \| E_{K_d}(IV_2, \textit{Contexte matériel})$, c'est-à-dire la concaténation d'un vecteur d'initialisation aléatoire (a priori différent du précédent) et du contexte matériel du processus chiffré avec la clé de chiffrement des données du processus K_d afin que le concepteur du programme puisse interpréter ces informations, et l'IV ;
- $MAC_{K_p}(\textit{Contexte matériel chiffré})$, c'est-à-dire le MAC (*Message Authentication Code*) calculé sur les données précédentes à l'aide de la clé symétrique K_p du processeur (cf. section 4.1).

3.2.2 Problème des appels systèmes

La solution précédente présente néanmoins un problème. Les arguments d'un appel système sont placés dans les registres généraux du processeur juste avant l'exécution de l'interruption logicielle. Or, avec l'effacement de ces registres lors du changement de contexte, le système d'exploitation ne pourra pas accéder aux informations relatives à l'appel système. De même, le code de retour de cet appel système est également placé dans un registre. Mais avec la restauration complète de ces registres à partir du contexte matériel contenu dans le tampon cb_{cur}, cette valeur ne peut pas être récupérée par le programme.

Pour résoudre ce problème, on laisse la possibilité au programme de choisir les registres qui devront être effacés ou non lors du déclenchement de l'interruption (et donc ceux qui seront visibles par le système d'exploitation ou non) et les registres qui devront être restaurés ou non à partir du contexte matériel lors du retour d'interruption. Cette possibilité n'est offerte que lors d'une interruption volontairement déclenchée par le processus (à l'aide de l'instruction `INT` par exemple dans l'architecture x86) afin d'éviter une fuite involontaire d'informations lors d'une interruption extérieure non générée ou non prévue par le processus.

À cette fin, un registre de contrôle spécial, $cpcr_1$, est ajouté. Ce dernier contient sous forme d'un champ de bits les registres qui ne doivent pas être effacés et ceux qui ne doivent pas être restaurés lors d'une interruption. Ce registre de contrôle ne peut être manipulé que par un processus qui s'exécute en mode sécurisé et est sauvegardé dans son contexte matériel

lorsqu'il ne s'exécute pas. Une fois l'interruption terminée, ce registre est réinitialisé à une valeur par défaut (tous les registres sont effacés et restaurés).

3.2.3 Problème des signaux UNIX

Sous UNIX, les signaux permettent au système d'exploitation d'informer un processus d'un événement particulier. Ils sont asynchrones et peuvent donc intervenir à n'importe quel moment de l'exécution de ce dernier. Lors d'un retour d'interruption, le système d'exploitation vérifie si un signal doit être délivré au processus et dans ce cas, si le processus a explicitement demandé à être informé de ce signal, il modifie le compteur de programme indiquant l'adresse de la prochaine instruction en le remplaçant par l'adresse de la première instruction de la fonction de traitement du signal donnée par le processus lors de l'enregistrement du signal. Ainsi lors de la réactivation du processus, la fonction en question s'exécute et traite le signal.

Dans l'architecture telle que décrite jusqu'à présent, cette modification du compteur de programme par le système d'exploitation est normalement considérée comme une tentative d'attaque contre le programme car le système d'exploitation ne doit pas être capable de modifier arbitrairement le flot normal d'exécution d'un processus en mode sécurisé.

SIG Nous devons donc permettre au système d'exploitation de demander au processeur de reprendre l'exécution d'un processus en se branchant à une adresse particulière spécifiée auparavant par ce dernier. À cette fin, l'architecture CryptoPage dispose d'un deuxième registre de contrôle ($cpcr_2$) que le processus sécurisé peut charger avec l'adresse d'une fonction responsable du traitement des signaux.

Lorsqu'un signal doit être délivré au processus dont l'exécution doit reprendre, le système d'exploitation doit tout d'abord effectuer une copie du contexte matériel du processus dans un autre tampon de contexte (afin de pouvoir restaurer ce dernier à la fin du traitement du signal). Ensuite, au lieu d'exécuter l'instruction `IRETS` de retour d'interruption sécurisée, il exécute l'instruction spécialisée `SIG` qui réalise les mêmes opérations sauf qu'elle remplace le compteur de programme contenu dans le tampon courant cb_{cur} par le contenu du registre $cpcr_2$ contenu dans le même tampon. Ainsi l'exécution du programme reprendra à l'adresse de la fonction de traitement des signaux.

De plus, afin d'identifier qu'un contexte matériel créé durant le traitement du signal est en relation avec le contexte matériel créé lors de l'interruption qui a précédé l'instruction `SIG`[1], cette dernière copie la valeur de la racine de l'arbre de Merkle R_{Merkle} (voir la section 4.2 sur la protection mémoire) vers un autre registre appelé R_{Old}.

SIGEND Une fois que la fonction de traitement du signal est terminée, le système d'exploitation peut restaurer l'état initial du processus grâce à l'instruction `SIGEND`. Cette instruction prend en argument le numéro du tampon qui contient le contexte matériel sauvegardé avant l'appel à `SIG`. Elle vérifie tout d'abord que les tampons correspondent bien en vérifiant que la valeur de la racine de l'arbre de Merkle de l'ancien tampon est bien égale à celle stockée dans R_{Old} dans le tampon courant. Si les deux valeurs correspondent, elle restaure l'ancien contexte matériel en utilisant la valeur de la racine de l'arbre de Merkle du tampon courant

[1]Sinon un système d'exploitation attaquant pourrait, lors du retour d'interruption, rejouer un ancien contexte matériel mais avec les nouvelles données en mémoire du fait de la mise à jour de la valeur de l'arbre de Merkle lors de la fin du signal.

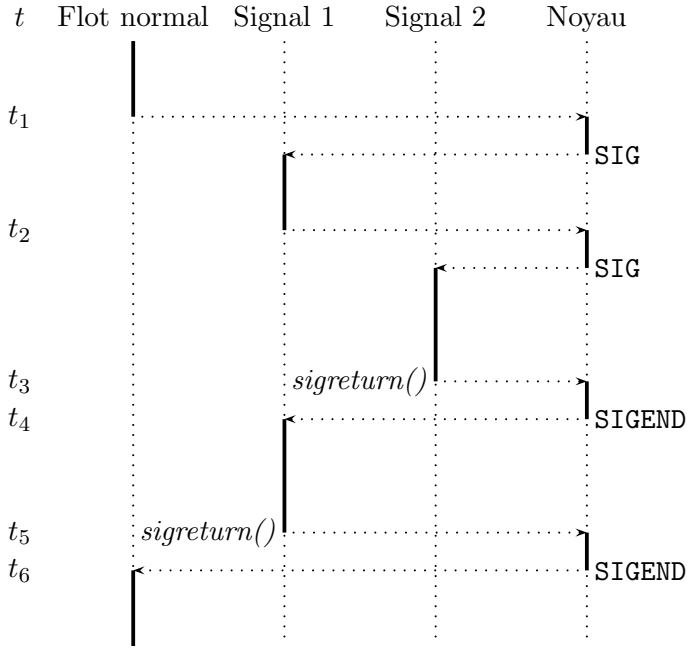

FIG. 3 – Le traitement des signaux.

Temps	t_1	t_2	t_3	t_4	t_5	t_6
Registres	Reg_1	Reg_2	Reg_3	Reg_2	Reg_4	Reg_1
R_{Merkle}	R_{M_1}	R_{M_2}	R_{M_3}	R_{M_3}	R_{M_4}	R_{M_4}
R_{Old}	$\perp$	R_{M_1}	R_{M_2}	R_{M_1}	R_{M_1}	$\perp$

TAB. 1 – Le contexte matériel courant à différents moments durant le traitement des signaux de la figure 3.

afin de prendre en compte les modifications apportées par la fonction de traitement du signal au contenu de la mémoire. La figure 3 et le tableau 1 résument ce mécanisme.

3.3 Durant le chargement d'un processus

Il nous reste à décrire l'aspect confidentialité lors du démarrage du programme. Lors du chargement d'un programme sécurisé, le code et les données initiales de ce dernier sont chiffrés. Le processeur doit donc en premier lieu récupérer les clés symétriques du programme afin de pouvoir commencer à l'exécuter. Pour permettre cette opération, le processeur dispose d'une paire de clés asymétriques (SK_p et PK_p). Lors de la création du programme, les clés symétriques de ce dernier sont chiffrées à l'aide de la clé publique du processeur et stockées dans une section spéciale du programme. Cette section contient :

– $E_{PK_p}\{K_d\}$, c'est-à-dire la clé de chiffrement des données propre au programme chiffrée à l'aide de la clé publique du processeur (en utilisant un algorithme de chiffrement

asymétrique comme RSA) ;
- $E_{K_d}\{$*contexte initial*$\}$, c'est-à-dire le contexte matériel initial du processus (qui contient l'état de l'ensemble des registres, ainsi que les clés de chiffrement du processus), chiffré à l'aide de la clé de chiffrement des données du programme (en utilisant un algorithme de chiffrement symétrique comme AES).

Lors du démarrage du processus, le système d'exploitation charge un des tampons de contexte avec cette section spéciale pour que le processeur puisse récupérer les clés du processus en le déchiffrant à l'aide de sa clé privée. Il déchiffre également le contexte matériel initial du programme. Le système d'exploitation copie ensuite le contexte déchiffré vers le tampon courant, avec une instruction spécifique, et l'exécution du programme peut démarrer.

4 Intégrité

Jusqu'à maintenant, nous nous sommes concentrés sur la propriété de confidentialité. Cependant, cette propriété seule n'est pas suffisante pour garantir le bon déroulement d'un processus. On doit également garantir l'intégrité de ce dernier.

4.1 Intégrité des contextes matériels

Étant donné que le système d'exploitation a la possibilité de manipuler les contextes matériels chiffrés durant l'interruption d'un processus sécurisé, le processeur doit également protéger leur intégrité. Pour garantir l'intégrité des contextes matériels, en plus de chiffrer ce dernier, le processeur calcul un MAC à l'aide de sa clé symétrique K_p. Cette clé étant uniquement connue du processeur, le système d'exploitation (ou un attaquant) ne peut pas modifier le contexte matériel chiffré lorsqu'il est stocké en mémoire sans que cette modification ne soit détectée par le processeur lors de l'opération de déchiffrement/vérification.

4.2 Intégrité mémoire

L'intégrité des données manipulées par le programme doit être garantie quand celles-ci sont stockées en dehors du processeur et, notamment, lorsqu'elles sont en mémoire vive. Plus précisément, le processeur doit s'assurer que la mémoire se comporte comme une mémoire correcte, c'est-à-dire vérifiant la propriété suivante : *la valeur lue à une adresse mémoire doit être la dernière valeur stockée par le processeur à cette adresse.*

Utiliser uniquement des valeurs d'authentification locales (par exemple calculer des MAC) est vulnérable à des attaques par rejeu où l'attaquant rejoue un bloc de données et sa valeur d'authentification. En conséquence, il est nécessaire d'utiliser un vérificateur sur l'ensemble de la mémoire afin de contrer cette attaque. Dans CryptoPage, on utilise une variante des arbres de Merkle afin de créer un résumé cryptographique (*hash*) sur toute la mémoire. Avec cette technique, décrite ainsi que des optimisations possibles dans [1, 4, 8], on construit un arbre couvrant la totalité de la mémoire. Les feuilles de l'arbre sont les données mémoire à protéger. Chaque nœud de l'arbre contient un résumé cryptographique du contenu de ses nœuds fils. La valeur du nœud racine est stockée dans un espace mémoire sécurisé au sein du processeur et ne peut donc pas être altérée par un adversaire.

À chaque lecture depuis la mémoire, le processeur vérifie le contenu des nœuds parents jusqu'à la racine et à chaque écriture en mémoire, il met à jour le chemin jusqu'à la racine. Un adversaire ne peut pas modifier le contenu de la mémoire sans être détecté puisqu'il lui

faudrait modifier également le chemin jusqu'à la racine, or il ne peut pas mettre à jour cette dernière.

Les algorithmes de lecture et d'écriture sécurisés peuvent être optimisés en utilisant les différents niveaux de cache du processeur en partant du fait que si un nœud est déjà présent dans le cache (et que le cache ne peut pas être altéré par un adversaire), son contenu est correct et donc qu'il n'y a pas besoin de poursuivre la vérification en remontant aux ancêtres.

5 Conclusion

Dans cet article, nous avons décrit les modifications à apporter à un processeur classique afin de lui permettre d'offrir des propriétés de confidentialité et d'intégrité aux processus qu'il exécute. Elles ont été ajoutées au simulateur de PC *Bochs* [2]. Grâce à ce simulateur, nous sommes capables de faire tourner un Linux adapté ainsi que des processus sécurisés sur une plate-forme CryptoPage/x86 virtuelle. Pour le moment, la propriété d'intégrité n'a pas encore été explorée complètement, notamment au niveau des optimisations pour le calcul et le stockage de l'arbre de Merkle. Cette étude, ainsi qu'un calcul précis des coûts, aussi bien en performances qu'en surface sur la puce, sont en cours.

Les performances fines restent à mesurer après instrumentalisation de *Bochs*. Néanmoins, les mécanismes de pipelines et de cache laissent espérer un surcoût relativement faible.

Remerciements

Nous remercions Mathieu Chevrier, élève de 3ème année en 1999–2000, pour sa première étude bibliographique sur la question ainsi que Cédric Lauradoux, stagiaire de dea en 2002–2003. Merci aux membres du projet incitatif OpenSmartCard du get pour leur discussions passionnantes et tout particulièrement à Sylvain Guilley pour ses commentaires constructifs sur le projet. Enfin ce projet a reçu un financement de thèse de la part de la dga.

Références

[1] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, et Moni Naor. « Checking the correctness of memories ». Dans *Proceedings of the 32nd annual symposium on Foundations of computer science*, pages 90–99. IEEE Computer Society Press, octobre 1991.

[2] « Bochs : The Open Source IA-32 Emulation Project », novembre 2004. `http://bochs.sourceforge.net/`.

[3] Guillaume Duc. « 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, juin 2004. `http://www.lit.enstb.org/~gduc/dea/rapport/rapport.pdf`.

[4] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, et Srinivas Devadas. « Caches and Hash Trees for Efficient Memory Integrity Verification ». Dans *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, pages 295–306, février 2003.

Instruction	Description
`MOV` src,$ecb_{i,j}$	Charge une partie d'un contexte matériel chiffré depuis l'adresse mémoire donnée vers le tampon de contexte ecb_i. Nécessite les privilèges du système d'exploitation.
`MOV` $ecb_{i,j}$,$dest$	Sauvegarde une partie d'un contexte matériel chiffré depuis le tampon de contexte ecb_i vers l'adresse mémoire donnée. Nécessite les privilèges du système d'exploitation.
`MOV` cb_i,cb_j	Déplace le contexte matériel non chiffré contenu dans le tampon de contexte cbi vers le tampon de contexte cb_j. Nécessite les privilèges du système d'exploitation.
`IRETS`	Effectue un retour d'interruption en restaurant le processus sécurisé à partir de son contexte matériel contenu dans le tampon de contexte courant cb_{cur}. Nécessite les privilèges du système d'exploitation.
`SIG`	Similaire à l'instruction `IRETS` mais au lieu de reprendre l'exécution à la prochaine instruction, elle branche à l'adresse d'exécution alternative spécifiée par le processus et stocké dans son contexte matériel. Cette instruction permet d'émuler les signaux UNIX et nécessite les privilèges du système d'exploitation.
`SIGEND` cb_i	Permet de reprendre l'exécution normale d'un processus après le traitement d'un signal. Le contexte matériel initial à restaurer est stocké dans le tampon de contexte cb_i et le dernier contexte matériel durant le traitement du signal dans le tampon de contexte courant. Cette instruction vérifie la concordance des deux contextes et restaure le contexte initial en mettant à jour les informations de vérification mémoire contenues dans celui-ci pour prendre en compte les modifications effectuées durant le traitement du signal. Nécessite les privilèges du système d'exploitation.
`MOV` reg,$cpcr_i$	Charge le registre de contrôle $cpcr_i$ avec le contenu du registre général reg. Ne peut être exécutée que par un processus tournant en mode sécurisé.
`MOV` $cpcr_i$,reg	Charge le registre général reg avec le contenu du registre de contrôle $cpcr_i$. Ne peut être exécutée que par un processus tournant en mode sécurisé.
`MOVNE` reg,$dest$	Stocke le contenu du registre général reg à l'adresse mémoire donnée mais sans le chiffrer et sans protéger son intégrité. Ne peut être exécutée que par un processus tournant en mode sécurisé.
`MOVNE` src,reg	Charge le registre général reg avec le contenu de l'adresse mémoire donnée mais sans le déchiffrer ni vérifier son intégrité. Ne peut être exécutée que par un processus tournant en mode sécurisé.

TAB. 2 – Résumé des nouvelles instructions de l'architecture CRYPTOPAGE/x86.

[5] Ronan KERYELL. « CRYPTOPAGE-1 : vers la fin du piratage informatique ? ». Dans *Symposium d'Architecture (SympA'6)*, pages 35–44, Besançon, juin 2000.

[6] Paul C. KOCHER. « Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems ». Dans *CRYPTO '96 : Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, numéro 1109, pages 104–113. Springer-Verlag, août 1996.

[7] Paul C. KOCHER, Joshua JAFFE, et Benjamin JUN. « Differential Power Analysis ». Dans *CRYPTO '99 : Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, numéro 1666, pages 388–397. Springer-Verlag, août 1999.

[8] Cédric LAURADOUX et Ronan KERYELL. « CRYPTOPAGE-2 : un processeur sécurisé contre le rejeu ». Dans *Symposium en Architecture et Adéquation Algorithme Architecture (*SYMPAAA*'2003)*, pages 314–321, La Colle sur Loup, France, octobre 2003.

[9] David LIE, Chandramohan THEKKATH, Mark MITCHELL, Patrick LINCOLN, Dan BONEH, John MITCHELL, et Mark HOROWITZ. « Architectural support for copy and tamper resistant software ». Dans *Proceedings of the ninth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 168–177, octobre 2000.

[10] NIST. « Advanced Encryption Standard (AES) », novembre 2001. Federal Information Processing Standards Publication 197.

[11] NIST. « Recommendation for Block Cipher Modes of Operation », décembre 2001. Special Publication 800-38A.

[12] N. SKLAVOS et O. KOUFOPAVLOU. « Architectures and VLSI Implementations of the AES-Proposal Rijndael ». *IEEE Transactions on Computers*, 51(12) :1454–1459, décembre 2002.

[13] G. Edward SUH, Dwaine CLARKE, Blaise GASSEND, Marten van DIJK, et Srinivas DEVADAS. « AEGIS : Architecture for Tamper-Evident and Tamper-Resistant Processing ». Dans *Proceedings of the 17 International Conference on Supercomputing (ICS 2003)*, pages 160–171, juin 2003.

[14] R. M. TOMASULO. « An Efficient Algorithm for Exploiting Multiple Arithmetic Units ». *IBM Journal of Research and Development*, 11(1) :25–33, janvier 1967.