Portage d'un système GNU/Linux sur l'architecture sécurisée CryptoPage/x86

Guillaume Duc et Ronan Keryell*

13 décembre 2004

Résumé

Il existe de nombreuses solutions permettant de concevoir des applications informatiques sécurisées mais celles-ci supposent que le support matériel d'exécution est sécurisé. Or mis à part dans le cas des applications tournant sur des cartes à puce, qui ne disposent que de ressources très limitées, le support d'exécution (par exemple un ordinateur personnel) n'est absolument pas sécurisé vis à vis d'un administrateur malveillant, d'un système d'exploitation hostile ou même d'un adversaire pouvant réaliser des attaques au niveau physique (comme l'instrumentalisation du bus du processeur).

Depuis quelques années, des architectures matérielles sont en développement afin de répondre à ce besoin. Dans cet article, nous allons étudier les modifications à apporter à un système d'exploitation de type Unix, et plus particulièrement le système GNU/Linux, afin de lui permettre d'exploiter les fonctionnalités offertes par l'architecture CryptoPage/x86 et d'exécuter des processus sécurisés.

Mots clés : Sécurité informatique, informatique de confiance, cryptoprocesseurs, cryptographie, processus sécurisés, système d'exploitation.

1 Introduction

Jusqu'à récemment, il n'existait que peu d'architectures matérielles sécurisées permettant d'assurer la sécurité des applications qui s'exécutaient sur celles-ci. Si l'on voulait exécuter une application sécurisée contre des attaques physiques, il n'existait pratiquement que la solution des cartes à puce. Malheureusement, les ressources disponibles sur ce type de support sont très faibles.

Pourtant il y a un réel besoin d'une informatique de confiance. Par exemple, les entreprises et les centres de recherche s'intéressent de plus en plus aux grilles de calcul qui leur permettent d'accéder à des ressources de calcul très importantes mais très distribuées. Cependant, actuellement rien n'empêche le propriétaire d'un nœud de la grille d'accéder au code du programme de calcul (ce qui peut poser des problèmes de propriété intellectuelle ou de divulgation si les algorithmes mis en œuvre sont protégés ou secrets), aux résultats (ce qui pose également des problèmes de propriété intellectuelle) mais il peut également perturber le bon fonctionnement du programme afin de lui faire produire des résultats erronés. Si le support d'exécution de chacun des nœuds de la grille était sécurisé, la confidentialité et l'intégrité du code et des données de ces programmes seraient garanties.

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

D'abord confinées aux applications militaires sensibles, les architectures matérielles sécurisées se développent pour permettre une exécution secrète de programmes et un chiffrement des données afférentes. On peut notamment citer AEGIS [12], XOM [8] ou CRYPTOPAGE [5, 7, 3]. 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.

Les modifications à apporter au niveau logiciel pour supporter de telles architectures sont en partie décrites dans [9, 6]. Dans cet article, nous étendons ces idées pour l'architecture CRYPTOPAGE/x86 tout en proposant d'autres solutions, notamment au niveau du traitement des signaux UNIX.

Dans cet article, nous allons décrire les modifications à apporter à un système d'exploitation pour qu'il puisse fonctionner sur une architecture matérielle sécurisée. Dans la section 2 nous décrirons l'architecture CRYPTOPAGE/x86 qui nous servira de base pour l'analyse, dans la section 3, nous étudierons les modifications à apporter à la bibliothèque standard C, dans la section 4, celles à apporter à un noyau et enfin dans la section 5 nous décrirons les expériences de simulation de ces modifications.

2 L'architecture CryptoPage/x86

Nous allons tout d'abord décrire les architectures CRYPTOPAGE et CRYPTOPAGE/x86. CRYPTOPAGE/x86 correspond à l'implantation des mécanismes de l'architecture CRYPTOPAGE [5, 7] sur un processeur de type Intel x86 (tel que le Pentium).

2.1 Objectifs de l'architecture

L'architecture CRYPTOPAGE permet l'exécution de processus sécurisés, c'est-à-dire de processus pour lesquels les deux propriétés suivantes sont garanties :

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

Par hypothèse, nous supposerons que tout ce qui est à l'extérieur du processeur (et notamment la mémoire externe) est entièrement sous le contrôle d'un adversaire qui peut effectuer n'importe quelle action malveillante destinée à mettre en défaut une des propriétés énoncées ci-dessus.

2.2 Confidentialité

Un certain nombre de mécanismes sont mis en place afin de garantir la propriété de confidentialité.

2.2.1 Chiffrement des données et des instructions

Tout d'abord, le code d'un processus sécurisé et les données manipulées par celui-ci sont chiffrés au niveau de chaque ligne de cache en utilisant le mode de chiffrement CBC (*Cipher Block Chaining* [11]) de l'algorithme symétrique AES [10]. Le mode CBC est utilisé afin

d'empêcher un adversaire d'identifier des motifs récurrents en mémoire malgré le chiffrement (comme par exemple une zone mémoire remplie de zéros). Pour cette opération, chaque processus dispose de deux clés symétriques (K_i et K_d) qui sont utilisées respectivement pour chiffrer les instructions et les données.

Le déchiffrement d'une ligne de cache s'effectue lorsqu'elle est chargée depuis la mémoire vers le niveau de cache le plus élevé intégré au processeur (on a supposé que tout ce qui était dans le processeur était physiquement sécurisé). Ainsi, lorsque le processeur manipule des données situées dans son cache, aucune pénalité en terme de temps d'accès n'est ajoutée. De même, lorsqu'une ligne de cache doit être écrite en mémoire, elle est préalablement chiffrée.

Avec ces mécanismes, la confidentialité des données et du code du processus est protégée durant son exécution.

2.2.2 Gestion des interruptions

Cependant, il est également nécessaire de protéger la confidentialité du processus lors d'une interruption ou d'une exception. Dans un processus normal, lors d'une interruption, le système d'exploitation a accès à l'état des registres du processus. Donc, afin d'empêcher cette fuite d'informations, le processeur sauvegarde le contexte matériel du processus (qui contient l'ensemble des registres généraux, des différents registres flottants, des sélecteurs de segment, du compteur de programme ainsi que des clés symétriques du processus et des registres spéciaux de CryptoPage/x86) au moment de l'interruption et le place dans un tampon spécial ajouté au processeur, appelé par la suite tampon de contexte courant (cb_{cur}). Ce tampon n'est accessible que par les mécanismes de sécurité du processeur. Ensuite, le processeur efface le contenu de tous ses registres avant d'exécuter la fonction de traitement pour cette interruption.

Pour relancer l'exécution de ce processus, le système d'exploitation utilise l'instruction spéciale `IRETS` (retour d'interruption sécurisée) qui restaure le contexte matériel situé dans le tampon courant et relance l'exécution à partir de ce dernier. Ainsi le système d'exploitation est incapable d'accéder à l'état du processus arrêté (ni même de l'altérer).

Néanmoins, pour permettre entre autre l'implémentation des appels systèmes (cf. section 3.1), un registre de contrôle spécialisé permet au processus de contrôler quels registres ne doivent pas être effacés et/ou restaurés depuis le contexte lors de la prochaine interruption logicielle.

De plus comme plusieurs processus sécurisés peuvent s'exécuter en concurrence sur le processeur, le système d'exploitation doit pouvoir échanger le contenu du tampon courant avec le contexte matériel d'un autre processus sécurisé pour pouvoir relancer l'exécution de ce dernier. Pour permettre une gestion efficace de ces transitions entre processus chiffrés, l'architecture CryptoPage/x86 dispose en réalité de plusieurs tampons pouvant contenir des contextes matériels (en plus du tampon courant). Chacun de ces tampons est constitué de deux parties : une (ecb_i) contenant le contexte matériel correspondant sous forme chiffrée et l'autre (cb_i) contenant le contexte matériel sous forme non chiffrée. Le système d'exploitation dispose d'instructions spécialisées (voir table 1) lui permettant de :

- copier le contenu d'un tampon vers un autre (et potentiellement vers ou depuis le tampon courant) ;
- stocker en mémoire une partie de la partie chiffrée d'un tampon (le système d'exploitation doit exécuter cette instruction plusieurs fois afin de stocker la totalité du contexte chiffré vers la mémoire du fait de la grande taille de ce dernier) ;

– charger une portion de la partie chiffrée d'un tampon de contexte depuis la mémoire.

Dès que la partie chiffrée d'un tampon est complètement chargée, son déchiffrement et sa vérification sont lancés et de même, dès que la partie non chiffrée d'un tampon est chargée, son chiffrement est lancé. Le système d'exploitation n'a accès qu'aux versions chiffrées des contextes matériels et ne peut donc pas récupérer d'informations sur le processus.

2.2.3 Chargement d'un programme sécurisé

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. Lors du démarrage, le système d'exploitation charge un des tampons de contexte avec cette section spéciale et automatiquement, le processeur va récupérer les clés du processus en la déchiffrant à l'aide de sa clé privée. Il déchiffre également le contexte matériel initial du programme. Ainsi, le système d'exploitation ne peut obtenir d'informations sur l'état initial du processus.

2.3 Intégrité

Nous allons maintenant aborder le problème de la protection de l'intégrité du code et des données manipulées par le processus.

2.3.1 Intégrité mémoire

L'architecture CryptoPage/x86 intègre un mécanisme de protection de l'intégrité mémoire qui permet de vérifier que la mémoire n'est pas altérée, c'est-à-dire que la propriété suivante est assurée durant toute la vie d'un processus sécurisé : *la valeur lue depuis une adresse mémoire doit être celle la plus récemment stockée à cette même adresse par le processeur.*

Pour assurer cette propriété, 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 ses optimisations possibles dans [1, 4, 7], on construit un arbre couvrant la totalité de la mémoire. Les feuilles de l'arbre sont les données 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 altéré 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 niveau 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.

2.3.2 Intégrité des contextes matériels

Il est également nécessaire de protéger l'intégrité des contextes matériels chiffrés lorsque ceux-ci sont stockés en mémoire par le système d'exploitation. Dans l'architecture CryptoPage/x86, on leurs adjoint un MAC (*Message Authentication Code*) calculé grâce à une clé secrète propre au processeur. Ainsi, lorsque le système d'exploitation recharge le contexte matériel chiffré depuis la mémoire, le processeur vérifie que le MAC est correct. Comme il ne peux pas être calculé sans la clé, un adversaire ne peut donc pas modifier un contexte matériel chiffré sans être détecté.

3 Modifications apportées à la bibliothèque standard

Cette section décrit les modifications à apporter à la bibliothèque standard afin de supporter les mécanismes de sécurité offerts par l'architecture CryptoPage/x86 décrite précédemment.

3.1 Appels systèmes

Sur un système x86 faisant tourner Linux, la bibliothèque standard offre un certain nombre de fonctions qui encapsulent tout le déroulement d'un appel système. Ces fonctions placent les arguments (ou seulement les adresses de ceux-ci en fonction de leur taille) de l'appel système dans les registres généraux (`EAX`, `EBX`, etc.) du processeur pour qu'ils puissent être récupérés par le système d'exploitation. Ensuite, elles exécutent une instruction spéciale permettant le changement de contexte vers le noyau (grâce à l'instruction d'interruption logicielle `INT 0x80` ou, sur des processeurs récents, l'instruction spécialisée `SYSENTER`).

Une fois l'appel système en question terminé (lorsque le processus redémarre suite à l'exécution de l'instruction de retour d'interruption `IRET` ou la nouvelle instruction `SYSEXIT`), ces fonctions d'encapsulation récupèrent la valeur de retour de l'appel système (stockée par le système d'exploitation dans le premier registre du processeur, `EAX`) et la renvoie à la fonction appelante.

Cependant, dans l'architecture CryptoPage/x86, afin de garantir la propriété de confidentialité et ainsi afin d'empêcher le système d'exploitation d'accéder aux valeurs des registres manipulés par un processus sécurisé, le processeur efface le contenu de ces registres lors d'une interruption et les restaure depuis le contexte matériel du processus lors du retour d'interruption. Donc en l'état, les appels systèmes ne pourraient plus fonctionner, le système d'exploitation n'étant plus capable de récupérer leurs arguments. Afin de permettre à un processus sécurisé de communiquer tout de même avec le noyau, un registre de contrôle spécialisé permet à un processus sécurisé de définir quels sont les registres qui ne devront pas être effacés et/ou restaurés lors de la prochaine interruption logicielle.

Ainsi, les fonctions d'encapsulation de la bibliothèque standard doivent être modifiées afin de définir la bonne valeur dans ce registre de contrôle pour ne pas effacer les registres qui vont contenir les arguments de l'appel système lors de l'interruption logicielle et de ne pas restaurer le registre qui contiendra la valeur de retour de l'appel système lors du retour d'interruption. La définition de la valeur de ce registre de contrôle s'effectue via une instruction spécialisée définie dans l'architecture CryptoPage/x86 (cf. table 1).

De plus, si besoin est, les fonctions d'encapsulation devront au préalable déchiffrer les arguments des appels systèmes si ces derniers doivent être stockés en mémoire plutôt que

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 cb_i 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 stockée 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. 1 – Résumé des nouvelles instructions de l'architecture CRYPTOPAGE/x86.

dans un registre, du fait de leur taille. Par exemple, le premier argument de l'appel système `stat` qui contient le nom du fichier à examiner, doit être au préalable déchiffré afin que le noyau puisse le comprendre. Afin de stocker temporairement les arguments sous forme déchiffrée, ces fonctions devront allouer temporairement de la mémoire. Une fois l'appel système terminé, et si le système d'exploitation est susceptible d'avoir modifié le contenu d'un argument stocké en mémoire, les fonctions d'encapsulation devront le chiffrer avant de se terminer. C'est par exemple le cas du second argument de l'appel système `stat` qui est un pointeur vers une structure de données qui contient les informations sur le fichier en question et qui, pour être interprétées par le programme, devront être chiffrées.

Enfin certaines fonctions d'encapsulation devront exister en deux versions : une traitant directement des données chiffrées et l'autre des données non chiffrées. C'est par exemple le cas pour l'appel système `write` qui devra exister en deux versions : la première qui passe au noyau les données à écrire sans déchiffrement préalable (et donc le système d'exploitation va écrire des données chiffrées) et une seconde qui déchiffrera au préalable les données à écrire.

3.2 Allocation mémoire

Pour permettre l'utilisation du mode de chiffrement CBC afin d'empêcher un adversaire d'identifier des motifs identiques en mémoire, le processeur doit pouvoir stocker en mémoire des vecteurs d'initialisation (IV), dont la taille est identique à celle d'un bloc pour l'algorithme de chiffrement utilisé (ex. 128 bits pour AES). Afin que ce stockage supplémentaire soit transparent pour le processus sécurisé, un mécanisme de transformation des adresses virtuelles est ajouté au processeur qui permet, lorsque le processus en cours s'exécute en mode sécurisé, de dilater les adresses. Par contre, quand le processus accède à de la mémoire non sécurisée ou que le processeur ne tourne pas en mode sécurisé (par exemple quand le noyau accède à la mémoire du processus), ce mécanisme est désactivé. Ainsi, au niveau du noyau, rien ne change en ce qui concerne la gestion de la mémoire du processus. Cependant, il manipulera sans le savoir aussi bien des données chiffrées que des IV.

Ainsi, les fonctions d'allocation mémoire de la bibliothèque standard (telles que `malloc`) doivent être modifiées afin d'allouer plus de mémoire que nécessaire pour que le processeur puisse y stocker les IV utilisés.

3.3 Bibliothèque CryptoPage

Enfin, la bibliothèque standard peut être modifiée afin d'inclure quelques fonctions utiles aux programmes s'exécutant sur l'architecture CryptoPage/x86 comme des fonctions permettant de chiffrer ou de déchiffrer des portions de la mémoire.

4 Modifications apportées au noyau Linux

Afin d'exploiter toutes les caractéristiques de l'architecture CryptoPage/x86, le noyau du système d'exploitation doit également être modifié. Nous allons dans cette section prendre l'exemple du noyau Linux.

4.1 Appel système `execve`

Le déroulement de l'appel système `execve`, qui permet de remplacer le code et les données du processus en cours d'exécution par ceux d'un autre programme dont le fichier exécutable est passé en paramètre, doit être modifié pour prendre en compte les exécutables sécurisés.

Lorsque cet appel système détecte que le programme qui va être lancé est chiffré (par exemple à l'aide d'un drapeau particulier dans les entêtes dans le cas d'un fichier ELF (*Executable and Linkable Format*)), il doit charger le contenu d'une section particulière du fichier dans un tampon spécialisé du processeur CryptoPage/x86. 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) ;
- $E_{K_d}\{\textit{contexte initial}\}$ 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).

Le chargement de cette structure s'effectue en utilisant une instruction spécialisée qui permet de charger une partie d'un des tampons spécialisés de CryptoPage/x86 avec un mot lu depuis la mémoire ou depuis un registre. Le noyau doit donc exécuter un certain nombre de fois cette instruction afin de charger la totalité de ce tampon. Une fois cette opération effectuée, une unité spécialisée du processeur commence immédiatement le déchiffrement de ce tampon. Cependant, étant donnée qu'une partie de cette structure de données utilise un algorithme de chiffrement asymétrique, cette étape est relativement longue. Donc en attendant la fin du déchiffrement, le système d'exploitation peut choisir d'exécuter un autre processus.

Une fois le déchiffrement terminé (indiqué par un drapeau dans un des registres de contrôle spécialisés de CryptoPage/x86), le noyau peut demander au processeur de copier le contexte matériel initial issu du déchiffrement du tampon vers le tampon de contexte courant et lancer son exécution à l'aide de l'instruction de retour d'interruption sécurisée (`IRETS`).

4.2 Retour d'interruption

Lorsque le système d'exploitation désire relancer un processus, il doit tout d'abord vérifier si ce dernier s'exécutait en mode sécurisé ou non. Si oui, le noyau doit au préalable s'assurer que le contexte matériel du processus en question se situe bien dans le tampon courant, et si ce n'est pas le cas, il doit l'y charger. Ensuite, au lieu d'effectuer un retour d'interruption classique (via l'instruction `IRET`), le noyau doit exécuter l'instruction spécialisée `IRETS` qui signale au processeur que le contexte matériel du processus à relancer n'est pas stocké dans la pile du noyau mais dans le tampon courant.

4.3 Gestion des tampons de contexte

Afin de gérer le contenu des différents tampons pouvant contenir des contextes matériels disponibles sur l'architecture CryptoPage/x86 (cf. section 2.2.2), le système d'exploitation doit être capable de savoir à quels processus appartiennent les contextes situés dans les différents tampons et de savoir où est situé le contexte d'un processus sécurisé en particulier (chiffré en mémoire, stocké dans l'un des tampons, stocké dans le tampon courant). Avec ces informations, il peut gérer les différents tampons afin de relancer l'exécution du bon processus sécurisé.

4.4 Ordonnanceur

En fonction du lieu de stockage du contexte matériel d'un processus donné, son redémarrage peut être plus ou moins long. Plusieurs cas sont possibles :

- lorsque le contexte matériel est stocké dans le tampon courant, il peut être relancé instantanément grâce à l'instruction `IRETS`. Dans ce cas, le coût de l'opération est presque nul ;
- lorsque le contexte matériel est stocké sous forme déchiffrée dans l'un des tampons, son rechargement nécessite sa copie vers le tampon courant puis l'exécution de l'instruction `IRETS`. Donc le coût de l'opération est relativement faible ;
- enfin, si le contexte matériel est stocké sous forme chiffrée en mémoire, le noyau doit tout d'abord le copier dans la partie chiffrée de l'un des tampons, demander au processeur de le déchiffrer, attendre la fin de ce déchiffrement et copier le contexte matériel déchiffré vers le tampon courant avant d'exécuter l'instruction de reprise `IRETS`. Dans ce cas, le coût de l'opération est élevé.

Afin d'améliorer les performances des processus sécurisés, l'ordonnanceur peut être modifié afin de tenir compte de ces coûts de relance pour choisir le processus à exécuter (un peu comme la pagination est prise en compte par les systèmes d'exploitation). Les processus sécurisés dont le contexte est déjà présent sous forme déchiffrée dans un tampon seront ainsi privilégiés.

4.5 Signaux

Le traitement des signaux demande quant à lui plus de modifications. En effet, lorsqu'un signal doit être délivré à un processus, le système d'exploitation doit modifier le flot normal d'exécution du processus afin d'exécuter la fonction de traitement du signal en question. Cependant, cette modification peut être perçue par l'architecture CryptoPage/x86 comme un tentative d'attaque.

Aussi, pour permettre d'implanter tout de même les signaux, l'architecture CryptoPage/x86 fournit le mécanisme suivant. Lors de son exécution, un processus peut, en définissant la valeur d'un registre de contrôle particulier, indiquer une adresse d'exécution dite alternative vers laquelle le système d'exploitation pourra demander son déroutement en cas de survenue d'un évènement particulier, par exemple lorsqu'un signal doit être délivré au processus.

L'architecture CryptoPage/x86 offre au système d'exploitation l'instruction `SIG`, similaire à `IRETS` à l'exception notable près qu'au lieu de relancer l'exécution du processus à partir du compteur de programme contenu dans le contexte matériel de ce dernier, elle utilise l'adresse d'exécution alternative spécifiée au préalable par le processus. Avant d'appeler cette instruction, le système d'exploitation devra dupliquer le contexte matériel du processus afin de pouvoir restaurer l'état du processus à la fin du traitement du signal.

Lorsque la fonction de traitement du signal se termine, le processus réalise un appel système pour le signaler et alors le noyau peut relancer le flot d'exécution normal du programme à l'aide de l'instruction `SIGEND`. Cette instruction prend comme paramètres deux tampons (celui sauvegardé avant l'exécution du signal et celui sauvegardé lors de l'appel système de fin de signal), vérifie qu'ils concordent bien (en utilisant des informations enregistrés lors du `SIG`) et, si tout est correct, elle restaure l'exécution du processus à partir du contexte matériel initial en modifiant les informations relatives à l'intégrité mémoire (racine de l'arbre de Merkle) afin de prendre en compte les éventuelles modifications mémoires

effectuées par la fonction de traitement du signal.

5 Expérimentations

Dans cette section nous allons décrire l'avancement de nos expériences de portage du système Linux sur l'architecture CryptoPage/x86.

5.1 Bochs

Tout d'abord, afin de simuler l'architecture CryptoPage/x86, nous utilisons le simulateur de PC *Bochs* [2]. Bochs est un logiciel libre qui simule le fonctionnement complet d'un PC classique (processeur, périphériques, etc.). Nous avons modifié le modèle du processeur afin de simuler une véritable architecture CryptoPage/x86. Pour le moment, seule la partie protection de la confidentialité des processus est implantée et l'aspect vérification de l'intégrité est en cours de réalisation.

5.2 Dietlibc

Les modifications décrites dans la section 3 ont été implantés dans la *Dietlibc*, une version allégée de la bibliothèque standard (plus simple à modifier que la *glibc*).

5.3 Chaîne de compilation

La chaîne de compilation a également été modifiée afin de permettre la génération d'exécutables chiffrés. L'éditeur de liens *ld* a été paramétré afin d'aligner correctement le code et les données à chiffrer sur des frontières de lignes de cache et une étape supplémentaire est rajoutée afin de chiffrer l'exécutable et d'y ajouter la section qui contient le contexte matériel initial et les clés de chiffrement utilisées.

5.4 Linux 2.6

Enfin, les modifications décrites dans la section 4 sont en partie implantées [3] (seul le nouveau fonctionnement des signaux n'est pas encore réalisé) dans la dernière version du noyau Linux (2.6.9 lors de l'écriture de cet article). Les modifications sont localisées principalement dans les fonctions de chargement d'un exécutable au format ELF (pour permettre le chargement dans le processeur des entêtes chiffrés du programme), dans les structures contenant les informations sur les processus en cours d'exécution (pour distinguer les processus sécurisés et identifier le lieu de stockage de leurs contextes matériels) ainsi que dans les fonctions bas niveau gérant les interruptions et les retours d'interruption.

6 Conclusion

Dans cet article, nous avons décrit les modifications à apporter à un système d'exploitation classique afin d'exploiter les fonctionnalités de sécurité offertes par un processeur sécurisé. Ces modifications ont été en grande partie validées par la simulation sur une architecture CryptoPage/x86 virtuelle. Nos travaux futurs vont porter principalement sur l'extension de l'architecture CryptoPage/x86 sur des systèmes multiprocesseurs. L'instrumentation du

système mémoire et cache de Bochs est en cours afin d'avoir une idée plus précise du surcoût d'exécution apporté par le système.

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.

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

[6] M. Kuhn. « The TrustNo1 cryptoprocessor concept ». Rapport Technique CS555, Purdue University, avril 1997.

[7] 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.

[8] 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.

[9] David Lie, Chandramohan A. Trekkath, et Mark Horowitz. « Implementing an Untrusted Operating System on Trusted Hardware ». Dans *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*, pages 178–192, octobre 2003.

[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] 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.