RECHERCHE

CryptoPage : support matériel pour crypto-processus

Guillaume DUC — Ronan KERYELL — Cédric LAURADOUX

Département Informatique
École Nationale Supérieure des Télécommunications de Bretagne, CS *83818*
F-29238 PLOUZANÉ CEDEX

RÉSUMÉ. *Les ordinateurs actuels ne sont pas aussi sécurisés que leur développement ubiquitaire et leur interconnexion le nécessiteraient. En particulier, on ne peut même pas garantir l'exécution sécurisée et confidentielle d'un programme face à un attaquant logiciel (l'administrateur système) ou matériel (analyseur logique sur les bus, contrôle des ressources de l'ordinateur,...).*
Dans cet article est proposée une architecture utilisant du chiffrement fort avec coopération du système d'exploitation pour se rapprocher de ces objectifs en essayant de garder de bonnes performances.
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. Enfin, des implications sur le système d'exploitation et quelques applications du système sont décrites.*

ABSTRACT. *Computers are widely used and interconnected but are not as secure as we could expect. For example, a secure execution cannot even be achieved or proved against a software (the system administrator) or hardware attacker (a logical analyzer on the computer buses).*
In this article a strong cryptography-based architecture with an operating system support is presented to reach such security levels without reducing the performance.
A cache line cipher and a memory verifier based on MERKLE *tree hash function is added to the internal cache in order to resist to various attacks and even replay attacks.*
Then the impact on the operating system and some applications are described.

MOTS-CLÉS : *Sécurité informatique, cryptoprocesseur, cryptographie, arbre de* MERKLE*, attaques par rejeu.*

KEYWORDS: *Computer security, cryptoprocessor, cryptography,* MERKLE *tree, replay attack.*

1. Introduction

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

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

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

Dans le domaine du calcul distribué, le calcul sur la grille permet typiquement à un individu ou à une société de disposer d'un ensemble d'unités de calcul fournies par des tiers pour avoir des puissances de calcul impossibles à avoir par le passé mais nécessaires pour résoudre des problèmes très demandeurs en puissance tel que la cryptanalyse ou la bioinformatique. Or aujourd'hui, un utilisateur est forcé de faire totalement confiance au propriétaire d'un tel matériel pour pouvoir y exécuter son code.

La notion d'exécution sécurisée de processus remonte au moins aux années 1960 dans le but d'empêcher l'espionnage durant la guerre froide [WAR 70] et le piratage informatique par simple copie : un processeur est spécialisé avec une clé et ne peut exécuter qu'un programme en clair ou chiffré avec cette clé. Un voleur récupérant un exemplaire d'un programme pour ce processeur ne pourra rien en faire [BES 80].

Cette problématique est restée très éloignée des considérations du grand public, mais, avec le développement de processeurs comportant de plus en plus de transistors, il reste suffisamment de transistors pour intégrer désormais de tels mécanismes de sécurisation et on constate, depuis peu, une explosion du domaine. Et il est intéressant de remarquer que, si ces notions reviennent à la mode de nos jours [TCP03, ENG 03], c'est aussi pour les mêmes raisons que les motivations d'origine de [BES 80], à savoir la protection des contenus intellectuels.

Avec le développement de médias de forte capacité et peu chers tel que les CD-ROM et autres DVD-ROM, la diffusion de logiciels et de contenus artistiques tel que les films ou la musique est entrée dans une ère de diffusion de masse : le très faible coût de reproduction du contenu numérique, qui consiste à recopier une suite de 0 et de 1, échappe aux lois économiques classiques de la production dans la mesure où le coût est concentré dans la confection de l'original et non plus dans la production des copies vendues.

L'apparition de réseaux mondiaux à fort débit de type Internet apporte un autre type de distribution de logiciels ou de contenus artistiques où chacun peut acheter une copie et la télécharger directement sur son ordinateur.

Malheureusement, le développement de versions inscriptibles de ces médias et plus généralement le faible prix du giga-octet des disques durs[1] permettent aussi le développement d'utilisations allant bien au delà de l'usage dans un but de copie privée de sauvegarde [TRA 03]... Cet usage illégal est aussi aidé par la possibilité qu'offre Internet de disséminer mondialement tout type d'information numérique lorsque on utilise des techniques issues du domaine du calcul distribué à haute performance (systèmes de fichiers distribués, stockage pair à pair,...).

Dans cet article ayant pour but de décrire une protection des processeurs contre les attaques dont celles par rejeu, on abordera tout d'abord la problématique et certaines applications de bases (§ 2) pour motiver le lecteur puis le contexte : la problématique des attaques et les solutions existantes en § 3. Ensuite, après description du principe du système (§ 4) on prendra comme objectif leur implantation dans un processeur moderne (§ 5) basée sur une vérification globale avec un arbre de MERKLE pour éviter les attaques par rejeu. On continuera avec le rajout d'un système de chiffrement des adresses (§ 6) et un survol de la mise en œuvre logicielle en § 7. On terminera un tour des autres approches et des travaux futurs (§ 8).

2. Motivations et applications

Il s'agit donc de résoudre deux problèmes principaux :

– le premier est d'éviter le piratage logiciel, la copie du programme (code exécutable et de données afférentes). Il s'agit de garantir la confidentialité des informations ;

– le second est de garantir une exécution sécurisée, ou, tout au moins, de rendre impossible l'exécution et la fin normale d'un processus sécurisé en cas de modification malveillante.

2.1. *Spécialisation du processeur*

Le premier problème (piratage logiciel) peut être résolu en spécialisant chaque ordinateur au point qu'un programme ne puisse plus tourner que sur un seul ordinateur unique au monde. La technique utilisable est l'usage de la cryptographie pour chiffrer de manière unique les instructions à exécuter qui seront ensuite déchiffrées par le processeur avant l'exécution. Si l'algorithme de chiffrement est robuste, un programme chiffré de telle manière ne pourra être exécuté *que* sur *ce* processeur.

Si on utilise un chiffrement symétrique, une clé possédée par l'auteur d'un logiciel pour chiffrer le programme d'un processeur donné pourra aussi être utilisée pour

1. 0,40 €/Go au moment de la dernière relecture de cet article.

déchiffrer son programme, bien entendu, mais aussi tous les programmes destinés à ce processeur, ce qui enlève une bonne partie de la sécurité : un secret partagé de tous ne le reste guère longtemps... Il suffit de voir comment l'algorithme DeCSS de chiffrement des DVD, partagé par tous les lecteurs de DVD, a été mis à jour simplement par rétro-ingéniérie d'un logiciel de lecture de DVD.

Pour éviter cela, on peut utiliser un algorithme cryptographique à clé publique et clé secrète [COC 73, DIF 76, RIV 78] au sein même du microprocesseur, ou de manière plus réaliste une approche mixte où un algorithme à clé publique est utilisé pour chiffrer une clé de session utilisée pour chiffrer de manière symétrique. C'est la technique utilisée dans les cartes à puces. Mais dans le cas d'un ordinateur générique, le principe est plus difficile à mettre en œuvre car il y a des éléments périphériques qui sont espionnables (contrairement à une carte à puce qui est un ordinateur complet) : les bus sont espionnables et leur états sont modifiables, la mémoire est espionnable et modifiable, les entrées-sorties aussi, etc. Bien que compliquées, de telles attaques sont possibles [HUA 02] et, une fois le logiciel mis à nu, son contenu peut être répliqué par des pirates.

La suite de cet article présentera des techniques essayant de parer à ce genre d'attaques en chiffrant et signant électroniquement tous les flux d'information entrant et sortant du microprocesseur afin de résoudre le second problème.

2.2. *Communications et routeurs sécurisés*

Le fait de disposer d'un processus s'exécutant de manière chiffrée dépasse le seul domaine de la protection des programmes. Dans le domaine des télécommunications sécurisées, l'usage de la cryptographie permet déjà d'assurer une communication secrète de bout en bout. Dans la mesure où les routeurs du réseau sont espionnables, il est néanmoins possible de savoir s'il y a ou non du trafic sur tel ou tel lien de communication et donc de savoir quelles sont les entités communiquant entre elles en surveillant toutes les instructions exécutées par le routeur et les données manipulées. En effet, même si les paquets sont chiffrés, le routeur est obligé de déchiffrer au moins les adresses de destination pour pouvoir faire du routage et cette information sera capturée.

Si on regarde par exemple le protocole de communication sécurisé d'Internet IPsec [IPS03], on a deux modes de fonctionnement, le mode point à point et le mode tunnel :

– le mode point à point permet à n'importe quel couple de machines de communiquer de manière sécurisée. Les données des paquets IP sont mises dans des entêtes ESP (*Encapsulating Security Payload* [ESP98]) mais les entêtes IP sont laissés tels quels afin que les paquets puissent être routés de la source vers la destination à travers le réseau. On peut donc, à défaut d'avoir une information sur le contenu, savoir encore que 2 ordinateurs communiquent, quand ils communiquent, savoir la taille des paquets échangés. Autant d'informations gênantes en milieu sensible qui peuvent permettre des attaques statistiques, des attaques temporelles [KOC 96], etc.

– le mode tunnel reliant typiquement 2 routeurs qui chiffrent les paquets devant passer d'un routeur à l'autre en les encapsulant dans des entêtes ESP. On ne voit plus passer des informations sur des paquets individuels mais juste des paquets circulant entre les 2 routeurs. Moins d'information filtre, c'est plus simple à configurer au niveau des machines terminales (car elles ne font pas d'IPsec, seulement les 2 routeurs en font), mais il y a une baisse de la tolérance aux pannes (points faibles sur les routeurs). On voit encore que des paquets circulent, ce qui peut être un manque de furtivité.

Si en plus un attaquant a un accès physique au routeur, la sécurité s'effondre. On a donc besoin de protéger l'algorithme de routage, de garantir son exécution sécurisée et de protéger ses données avec en particulier les paquets réseaux échangés sur le réseau.

En utilisant un système de processus cryptographiques dans les routeurs, on ne peut même plus accéder à l'algorithme de routage et si même les données stockées en mémoire ou envoyées ou reçues depuis les dispositifs d'entrées-sorties sont chiffrées, on a enfin un routeur sécurisé. Le fait que de l'information transite entre tel et tel nœud peut être dissimulé en cachant par exemple le flux dans un pseudo-bruit continu sur chaque lien. On ne pourra distinguer l'absence de la présence de transmission puisque on ne saura pas déchiffrer le code générant ce flux au niveau des routeurs, ni déchiffrer les données et autres entrées-sorties du processus.

2.3. *Contenus artistiques numériques*

Outre la protection de programmes ou de réseaux, il est nécessaire de pouvoir empêcher la copie illégale de contenus artistiques numériques [ESK 03, TRA 03]. Le problème est que ces contenus doivent être concrétisés en sons et en images pour être perçus par leurs utilisateurs.

Donc, dès le moment où on a la transcription sous forme physique (c'est à dire analogique) du contenu numérique, ce contenu peut être copié et re-numérisé. Il y a certes une perte de qualité liée à cette double conversion numérique→analogique→numérique, mais cette perte de qualité n'est pas forcément (très) sensible, surtout si en plus on utilise déjà des algorithmes de compression avec pertes.

Mais si on peut accéder directement au contenu numérique, typiquement avant les convertisseurs numérique→analogique, on peut copier les données d'origines. On peut se protéger d'une telle attaque en intégrant ces convertisseurs sur le processeur ou en utilisant un canal chiffré entre le processeur et les convertisseurs.

Une méthode plus définitive serait de rajouter une partie logicielle au contenu artistique [KOC 03], une petite application multimédia présentant tel groupe de musique ou telle mise en scène d'un film, ou encore de sortir simplement du cadre monodimensionnel du classique CD de musique qu'il suffit d'écouter une fois du début à la fin pour pouvoir le recopier.

Une autre approche serait de pousser l'approche haute fidélité en rajoutant une partie logicielle gérant l'adaptation en temps réel du contenu sonore à l'acoustique de la salle à partir de retours avec des microphones, etc. Cette partie logicielle serait chiffrée ainsi que le contenu artistique pour éviter une extraction triviale du contenu. Un pirate pourrait certes enlever cette partie logicielle mais la copie résultante serait assez loin de l'original et facilement identifiable.

D'autres approches permettent de suivre ou de gérer les fuites, basées sur le marquage individuels des contenus, pour tracer les responsables des fuites [COX 01], voire ensuite les éliminer de la diffusion ultérieure [NAO 03]. Certaines méthodes motivent les utilisateurs à respecter leur contrat en les obligeants à utiliser des données très personnelles pour accéder aux contenus (par exemple son numéro de carte bancaire comme mot de passe). Et diffuser son contenu personnalisé revient à diffuser son numéro de carte bancaire à tout vent [DWO 96] !

Toutes ces méthodes reposent malheureusement sur l'hypothèse que l'exécution du logiciel par le matériel est sure, et c'est cette hypothèse que cet article va essayer d'assurer.

Il est clair que cette problématique, finalement sociale, est difficile et ne peut être résolue par la seule technologie même si certains considèrent qu'une solution partielle peut suffire à contenter les éditeurs [TRA 03].

3. Contexte

Les attaques physiques contre les systèmes informatiques restent relativement méconnues du grand public même si, depuis l'engouement des faussaires pour les cartes à puces et l'apparition de cryptoprocesseurs dans les consoles de jeux vidéos (X-BOX [HUA 02],...), le sujet devient de plus en plus en vogue.

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

– les mémoires (flash) peuvent imprimer définitivement les valeurs qu'elles sauvegardent ;

– on peut refroidir les composants jusqu'à atteindre la rémanence de la mémoire [SKO 02a] ;

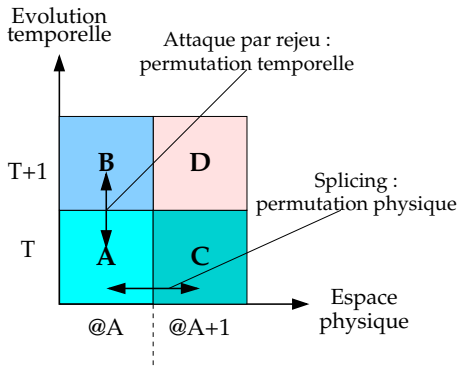
– l'exposition des composants aux radiations (fours à micro-ondes,. . .) impriment les données en mémoires [WEI 00] ou les altèrent ;

– dans le domaine des infrarouges, le silicium des composants devient transparent et on peut accéder au contenu mémoire par imagerie [SKO 02b].

Une technique de rétro-ingéniérie brutale mais classique consiste à retirer le boîtier du circuit intégré qui est alors susceptible d'être modifié. On peut ainsi affaiblir les algorithmes et provoquer des failles dans le système. Mais il y a des techniques moins intrusives telles que le parasitage des diodes utilisées comme générateur de bruit aléatoire par des sources électro-magnétiques peut provoquer la génération de clefs biaisées [WEI 00], les analyses de consommation [KOC 99], des analyses de temps de calcul [KOC 96], etc. qu'il faut aussi résoudre mais qui ne font pas l'objet de cet article.

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

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




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

Figure 1. *Les différentes attaques par permutations.*

En fait, le principal problème d'un ordinateur moderne est que toute la partie extérieure au boîtier du processeur de la hiérarchie mémoire est incontrôlée : périphériques mémoires, bus, contrôleurs ou bien unités de stockage,... C'est à cet aspect que s'intéresse cet article.

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

– génération : on cherche à effectuer des attaques avec texte choisi pour casser les algorithmes. Pour cela, l'attaquant doit pouvoir injecter ses propres données. C'est ainsi que le microprocesseur DALLAS DS5002FP a été cassé [KUH 98]. L'antidote consiste à rajouter à chaque mot mémoire un code d'authentification ou une signature électronique liée de manière unique au processeur [GIL 98, KER 00, LIE 03a] : si le mot n'est pas signé avec la signature du processeur l'exécution s'arrête. C'est ce problème qui sera traité en § 4.2 ;

– permutation spatiale ou temporelle : ce type d'attaque, si on ne peut pas faire les attaques précédentes, vise plutôt à modifer le comportement des programmes en modifiant l'organisation mémoire afin d'obtenir des privilèges ou d'effectuer des attaques par débordement. En effet, puisque le point précédent empêche l'attaquant de générer des données correctement authentifiées, il peut essayer de substituer des données générées par le processeur, donc correctement signées, avec d'autres (figure 1). Ce sont des attaques par rejeu car consistant à rejouer une vieille données à une adresse différente ou pas.

Dans un système à mémoire classique, la valeur récupérée lors d'un accès en lecture à l'adresse a correspond (heureusement !) à la valeur écrite lors du dernier accès en écriture effectué à cette adresse. Cela signifie donc qu'un accès en écriture révoque la donnée précédemment stockée. Cette méthode de révocation par écrasement physique est satisfaisante en temps normal, mais si on considère que tout le chemin de données n'est pas sûr, alors ce mode de révocation n'est plus suffisant : un attaquant pourrait très bien se contenter de bloquer certaines écritures en mémoire et si, comme sur l'exemple de la figure 1, c'est la zone qui contient le compteur de boucle `i`, faire itérer le processeur plus que nécessaire en gardant `i` constant et donc faire sortir des informations confidentielles par débordement de la zone pointée par `p`. Il s'agit donc d'un cas d'attaque par permutation triviale.

De façon similaire, en exécutant pas à pas un programme, on peut très bien aller modifier les mémoires en cours d'utilisation. Comme aucun mécanisme, mis à part l'effacement physique de la donnée, ne permet de révoquer celle-ci, cela signifie qu'un adversaire enregistrant les valeurs intermédiaires prises par une adresse physique, est capable d'écraser sa valeur finale par une valeur intermédiaire [GAS 03].

C'est la seule faille connue dans des systèmes tels que XOM[2] [LIE 03a] ou CRYPTOPAGE-1 [KER 00] qui utilisent de la cryptographie forte pour chiffrer et signer les données et les instructions. Comme il n'y a pas de protocole assurant la révocation des données, ces systèmes sont vulnérables au rejeu.

4. Principes du système CryptoPage-1

4.1. *Chiffrement*

Il y a eu plusieurs propositions de crypto-microprocesseurs exécutant des instructions chiffrées et chiffrant leurs données afin de compliquer la rétroingéniérie des programmes [BES 80, BES 81, BES 84] et il y a même plusieurs circuits commerciaux comme les Dallas Semiconductor DS5000FP [Dal99a] et DS5002FP [Dal99b] qui sont des micro-contrôleurs de type 8051 en version sécurisée.

Globalement l'approche est schématisée sur la figure 2. Le principe est de chiffrer les accès à la mémoire sur le bus d, à savoir les données et les instructions accédées

2. eXecute Only Memory.

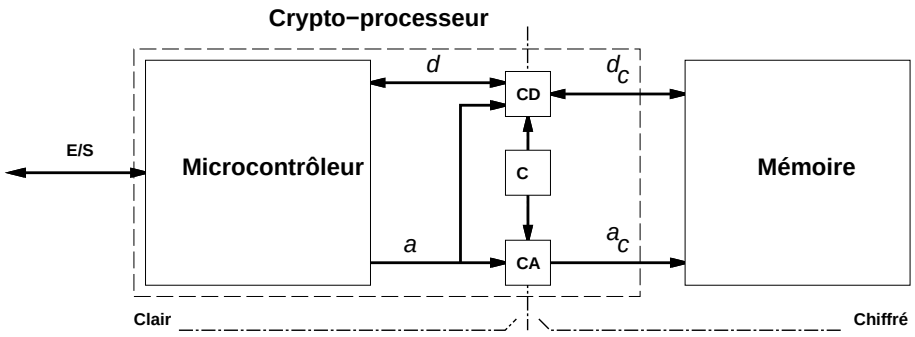


Figure 2. *Principe d'un processeur de type DS5002FP chiffrant sa mémoire.*

par le microprocesseur, avec une fonction de chiffrement des données C_D dépendant d'une clé secrète c_s unique au processeur. Ainsi, le seul moyen de pouvoir déchiffrer ce qui passe sur le bus de données chiffrées d_c ou en mémoire est de connaître la clé c_s ou d'avoir cassé la fonction C_D. En ayant une fonction cryptographique suffisamment sûre dans l'état actuel de la connaissance et une clé suffisamment longue pour épuiser toute attaque brute essayant toutes les clés possibles, le problème doit résister à un pirate [MEN 97]. Afin de compliquer la tâche d'un attaquant, la fonction C_D dépend aussi de l'adresse, rendant plus difficile des détections de zones mémoires uniformément remplies qui auraient permis des attaques à texte partiellement connu[3] et on stocke en fait $d_c = C_D(c_s, a, d)$.

Dans les systèmes de Robert M. BEST, les périphériques d'entrée-sortie sont sur le bus mémoire et incluent le même système de chiffrement afin de pouvoir communiquer de manière secrète avec le processeur. Le système utilise une architecture de type HARVARD avec un système de chiffrement différent sur le bus de données et le bus d'instructions afin d'éviter que le programme puisse se lire lui-même et le sortir en clair sur un port d'entrée-sortie lors d'une attaque.

Dans le cas des systèmes de Dallas Semiconductor, les entrées-sorties sont sur un bus séparé non chiffré qui simplifie l'utilisation du circuit pour le marché visé par ces micro-contrôleurs : terminaux de transactions financières ou de télévision payante, etc. Afin de mélanger un peu plus les cartes, le bus d'adresse est lui même mélangé via la fonction de chiffrement d'adresse C_A pour rendre plus difficile la compréhension de l'allocation mémoire. Comme il n'y a pas de mémoire cache ou de hiérarchie mémoire cela ne pose pas de problème. Le processeur est doté d'un mécanisme d'autodestruction en cas d'ouverture du boîtier, d'un mécanisme de détection fine de variations dans la tension d'alimentation, d'un système pouvant empêcher la reprogrammation. En sus, dès que le processeur n'accède pas à la mémoire, un système d'adressage génère de fausses lectures à la mémoire afin de troubler toute surveillance.

3. Typiquement les variables initialisées à 0 auraient été représentées de manière identique dans tout le programme, facilitant le travail de l'attaquant.

La programmation de la mémoire de programme non volatile d'un tel système est classique si ce n'est la phase de chargement initiale qui utilise un mode particulier du processeur permettant de chiffrer les instructions au fur et à mesure de leur stockage en mémoire programme.

Le DS5002FP est aussi doté d'un revêtement du silicium empêchant l'usage de micro-sonde (la version DS5002FPM), d'une patte d'autodestruction (en fait l'effacement de la clé) qu'on peut relier à un mécanisme externe de détection d'intrusion, d'un vrai générateur aléatoire de 64 bits pour tirer au hasard la clé qui est de 64 bits contre 40 bits pour le DS5000FP. L'intérêt est que même le programmeur du circuit n'a pas accès à la clé c_s et ne peut pas déchiffrer son propre programme et donc ne permet pas d'avoir une porte dérobée triviale pour accéder ensuite au système.

Le système semble robuste. Pourtant été cassé dans le cas du DS5002FP [KUH 98] ! L'attaque utilise d'abord le fait que la fonction C_D est bijective : toute valeur envoyée depuis la mémoire à l'entrée de C_D^{-1} est traduite en une instruction potentielle. Le fait que le jeu d'instruction soit publié aide ensuite : on va essayer d'envoyer toutes les instructions possibles séparées par des *reset* pour repartir dans un état cohérent à chaque fois. Comme le jeu d'instruction est sur 8 bits il n'y a que 256 essais à faire. L'idée étant de générer l'instruction commandant le port d'entrée-sortie pour faire sortir de l'information, il faut aussi faire la recherche sur l'adresse immédiate du port, ce qui multiplie le nombre d'essais à faire. De proche en proche on va construire un programme qui aura pour effet de sortir le contenu de la mémoire sur le port d'entrée-sortie en clair.

Afin d'éviter cette approche, les machines de Robert M. BEST interprètent toute instruction non définie, c'est à dire les valeurs ne représentant aucune instruction du jeu d'instructions valides, comme une instruction d'autodestruction, ce qui doit empêcher ce type d'attaque contrairement à ce qui est prétendu dans [KUH 98] qui semble ignorer ce détail. Mais dans le cas d'un ordinateur générique où on est amené à exécuter malheureusement n'importe quel code, du simple programme buggué (on écrase un pointeur de fonction...) au virus ou cheval de Troie le plus hostile, on ne peut pas se permettre qu'une simple exécution d'instruction illégale détruise le processeur. Néanmoins, si on n'exécute que des programmes prouvés comme étant sûrs, c'est une option à considérer et on peut imaginer que ce mode de fonctionnement puisse être choisi lors du démarrage du système.

4.2. *Authentification de la mémoire*

Afin d'éviter néanmoins ce type d'attaque, on peut soit chiffrer des blocs de données plus importants d'un coup, de type ligne de cache [KUH 98], soit rajouter une signature électronique ou code d'authentification [MEN 97]. Dans tous les cas le but est de devoir faire une recherche brute dans un espace beaucoup plus vaste et être concrètement impossible.

Si on se contente de chiffrer des blocs de données ou d'instructions, toute injection pirate d'autres valeurs sur le bus externe seront bijectivement traduites par des valeurs internes qui perturberont l'exécution du programme.

C'est donc une méthode avec signature électronique ou plus simplement avec une authentification que nous préférons. L'idée est d'associer à toute valeur d une valeur hachée par une fonction H utilisée comme MAC (Message Authentication Code [MEN 97]) ayant comme propriété que H^{-1} est très difficile (en pratique impossible) à calculer avec les moyens actuels. Au lieu de stocker en mémoire $C_D(c_s, a, d)$, on va stocker la valeur chiffrée et la concaténation de d et de son hachage, c'est à dire la donnée authentifiée et chiffrée $d_{sa} = C_D(c_s, a, d)\|H(d)$, voire $d_{sa} = C_D(c_s, a, d)\|H(d, a)$ pour que l'authentification dépende aussi de l'adresse et donc évite de fournir de l'information sur des plages mémoire de contenu identique.

Lors du déchiffrement d'une information chiffrée d_{sa} on extrait (d'', h''). Si le principe ci-dessus a été utilisé pour le chiffrement, on doit retrouver le fait que $H(d'') = h''$, ce qu'il suffit de calculer pour vérifier. Si la propriété n'est pas vérifiée, c'est qu'on essaye de déchiffrer des données qui n'ont pas été chiffrées de cette manière. Si la taille du résultat de h est de b_a bits, un pirate a potentiellement une chance sur 2^{b_a} d'injecter une valeur qui passera la vérification. Si, par exemple, $b_a = 128$, cela fait une chance sur $3{,}4.10^{38}$, ce qui semble raisonnable pour éviter des attaques, même intensives, dans l'état actuel de la technologie.

En fait, comme on chiffre la donnée de manière forte, on peut très bien se contenter d'avoir $d_{sa} = C_D(c_s, a, d\|H(d))$ où H est une fonction très simple de type CRC, voire même la fonction constante telle que 128 bits de 0 par exemple. Là aussi, lors du déchiffrement il faudra vérifier qu'on a bien $H(d'') = h''$. Si la fonction de chiffrement C_D est faible, le fait que H soit simple à calculer peut faire qu'on sera amené à chiffrer du texte avec une certaine corrélation, ce qui peut affaiblir la robustesse du système. Dans ce cas on peut avoir intérêt à choisir une fonction H de hachage cryptographique mais, évidemment, cela risque d'augmenter la longueur du chemin critique de vérification et de diminuer les performances globales du système. Évidemment il ne faut pas perdre de vue qu'il faut choisir une réalisation de C_D qui soit compatible avec l'usage d'authentification qu'on veut en faire. Par exemple, utiliser du chiffrement AES en mode ECB serait inutilisable car on chiffrerait toujours de la même manière 128 bits de 0 sensés rajouter l'authentification [DWO 01].

Le problème, avec tous les systèmes d'authentification, est que cela rajoute de la redondance : si les informations sont chiffrées par blocs de b bits et que l'authentification rajoute une taille de b_a bits en plus, une quantité de n bits non chiffrés en interne nécessitera d'avoir au moins $\lceil \frac{n}{b} \rceil (b + b_a)$ bits de mémoire externe sous forme chiffrée. Concrètement, si on chiffre au niveau d'une ligne de cache de 32 octets (soit 256 bits) et qu'on a un résultat de hachage de 128 bits, l'occupation mémoire du programme augmente de 50 %. C'est raisonnable car cela ne touche qu'un programme chiffré, d'autres programmes pouvant tourner en mode non chiffré dans l'ordinateur simultanément) et dans un ordinateur généraliste la contrainte de mémoire est plus

faible vus les prix actuels. Mais cela a aussi un impact en terme de vitesse dans le cas d'applications limitées par le débit mémoire : elles seront ralenties d'autant.

En choisissant des blocs plus gros on peut diminuer la part de l'authentification mais plus les blocs sont gros, plus la latence mémoire sera importante car avant d'être capable de savoir si une lecture est correcte il faudra lire tout le bloc, le déchiffrer et le vérifier. En utilisant de l'exécution spéculative, on peut néanmoins limiter la latence de vérification.

Reste encore le problème qu'un bloc identique à une adresse identique sera toujours représenté de la même manière en mémoire, ce qui permet potentiellement à un attaquant de détecter par exemple les réinitialisations successives de tableaux, etc. Pour éviter ce genre de détections, on rajoute des bits aléatoires r qui vont être stockés en plus comme sel pour éviter ce genre de détection, soit $d_s = C_D\Big(c_s, a, r, \big(d, H(d)\big)\Big)$. Concrètement, cela peut être réalisé sous forme de chiffrement par bloc de type CBC avec un vecteur d'initialisation aléatoire [DWO 01].

On peut imaginer que tous les paramètres précédents soient propres à un programme et donc que le programmeur puisse choisir son propre compromis, en terme de sécurité, latence ou occupation mémoire.

Le corollaire est que de toute manière, il va falloir trouver un moyen de ranger ces b_a bits supplémentaires en mémoire. On peut choisir de stocker les blocs chiffrés de manière contiguë ou encore de stocker les données par blocs de b bits aux adresses virtuelles internes a et de rajouter les b_a bits supplémentaires dans une autre zone virtuelle à une adresse $A_S + a$ par exemple. Les deux méthodes sont représentées sur la figure 3.

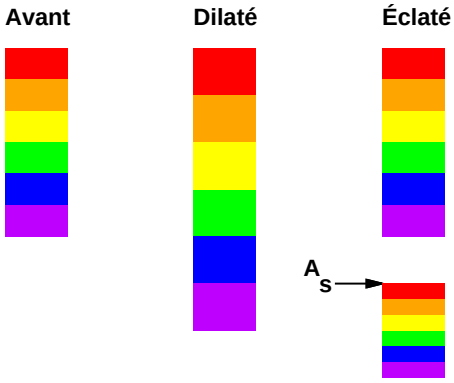


Figure 3. *Quelques possibilités de rangement des données en mémoire avec leur authentifi cations.*

Dans le premier cas « Dilaté » un octet chiffré à l'adresse virtuelle interne a se retrouvera en mémoire dans le bloc d'adresse virtuelle $[\lfloor \frac{a}{b} \rfloor \frac{b+b_a}{b}, \lfloor \frac{a}{b} + 1 \rfloor \frac{b+b_a}{b} - 1]$. C'est la traduction physique a_{pl} de cette adresse virtuelle linéarisée a_l qui sortira sur le bus mémoire lors des accès chiffrés au lieu de l'adresse physique a_p classique. Cette conversion peut être intégrée à la MMU [4] du système. Cela nécessite de rajouter un multiplieur sur le chemin d'adresse.

4. *Memory Management Unit* : système matériel gérant la mémoire virtuelle : traduction d'adresse virtuelle vers adresse physique, gestion des droits d'accès à la mémoire,...

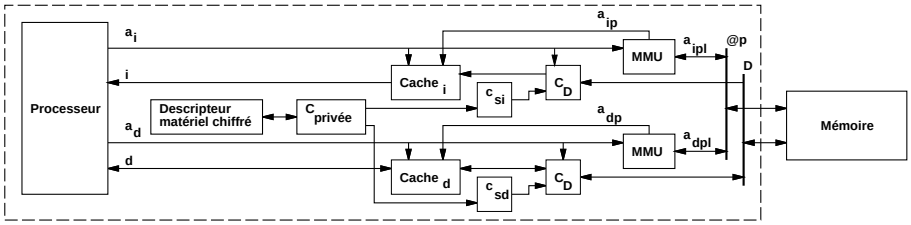


Figure 4. *Principe d'un processeur* CRYPTOPAGE*-1*.

Dans le second cas « Éclaté », l'octet se retrouvera dans deux zones $[\lfloor \frac{a}{b} \rfloor b, \lfloor \frac{a}{b} \rfloor b + b - 1]$ et $[A_S + \lfloor \frac{a}{b} \rfloor b_a, A_S + \lfloor \frac{a}{b} \rfloor b_a + b_a - 1]$. Dans ce dernier cas on va perdre en localité, ce qui peut être pénalisant au niveau des temps d'accès à de la mémoire dynamique ainsi qu'au niveau de la mémoire d'échange sur disque.

Dans les deux cas il faudra faire attention aux conflits d'adresse lorsqu'un programme manipulera à la fois des valeurs chiffrées et des valeurs en clair dans la mémoire.

Une autre solution proposée dans XOM [LIE 00] est d'utiliser la mémoire supplémentaire réservée aux codes correcteurs d'erreur mémoire (ECC) si on a un tel système et que le bus permet d'accéder à ces bits. Dans ce cas on peut choisir la taille des blocs et le nombre de bits supplémentaires pour qu'ils soient intégralement stockés dans la zone d'ECC. L'avantage est qu'on n'a plus de modification de l'espace d'adressage mais on n'a plus de correction d'un bit d'erreur mémoire disponible. L'authentification détectera par contre l'erreur et le programme s'arrêtera. On pourrait néanmoins imaginer augmenter la taille des blocs afin d'avoir suffisamment de bits de correction pour rajouter à la fois ce mécanisme de code correcteur mémoire et d'authentification dans CryptoPage. Mais un autre problème est que, si on veut sauvegarder une zone de mémoire chiffrée sur disque, il faudra rajouter un mécanisme matériel au niveau de la mémoire et du bus permettant d'accéder à ces bits supplémentaires.

Dans la suite on préférera la première approche car c'est la plus simple si on veut exporter une zone mémoire chiffrée sur disque (il n'y a qu'une seule zone mémoire à sauvegarder) et la plus performante en terme de localité mémoire. C'est celle schématisée sur la figure 4 et nommée « Dilaté » sur la figure 3.

On raisonne en adresse mémoire virtuelle car on bénéficie ainsi du travail de gestion mémoire fourni par un système d'exploitation classique. Il gérera de la même manière un processus chiffré ou non. Cela signifie aussi que si on a un processeur avec un cache en adresse physique, il faut stocker aussi l'adresse virtuelle dans le cache afin de permettre un chiffrement dépendant de l'adresse virtuelle. Cela empêche aussi le système d'exploitation de faire une attaque en modifiant la valeur d'une ligne de cache d'un processus chiffré simplement en jouant avec la table des pages.

4.3. *Sécurisation et mise en œuvre*

Une fois les bases de la sécurité établie et le schéma global défini sur la figure 4, il s'agit de ne plus laisser filtrer la moindre information d'un processus chiffré vers un autre processus. Afin d'éviter qu'un bug du programme permette de lire et de décoder les instructions, les bus de données et d'instructions sont chiffrés avec 2 clés privées différentes, respectivement c_{sd} et c_{si}. Un processus ne peut pas accéder aux clés, un processus chiffré est la seule entité pouvant accéder en clair à une de ses informations chiffrées. En particulier, même le noyau en mode superviseur ne peut accéder au contenu des registres ou du cache d'un processus chiffré.

Le fait d'utiliser du chiffrement symétrique à ce niveau a pour avantage de pouvoir partager le code et/ou les données entre plusieurs processeurs différents, à condition de partager ces clés.

Pour éviter les attaques à texte connu (reconnaître par exemple des mers de 0), on a considéré faire dépendre le chiffrement de l'adresse. Mais cela pose d'autres problèmes :

– si on écrit plusieurs fois une même valeur au même endroit, on aura la même valeur chiffrée ;

– si on accède via des mécanismes de mémoire virtuelle à une même case mémoire par différentes adresses, seule une permettra de relire la valeur correcte.

Pour éviter le premier problème on peut utiliser un chiffrement qui dépendra aussi d'un vecteur d'initialisation (IV) aléatoire comme dans le mode CBC [DWO 01]. C'est vers cette méthode que nous nous orientons [DUC 05b], au prix certes d'un autre surcoût mémoire : pour chaque bloc chiffré, on va par exemple, dans le cas de l'AES, rajouter 128 bits.

Par contre, dans le cas de segments de mémoire en lecture seule (typiquement un programme exécutable), on n'a pas ce besoin. On peut donc rajouter un mode au niveau du système de chiffrement permettant de choisir pour chaque page quel mode de chiffrement on a besoin pour éviter de gâcher trop de mémoire et de débit mémoire.

On a vu dans la section 4.3 qu'on rajoutait des bits d'authentification qui changeaient le mode d'adressage par dilatation de l'espace. Si on rajoute des IV pour le mode CBC on dilatera encore plus. On verra en section 5.1, qu'avec un mécanisme d'authentification globale, on peut économiser cette authentification

Enfin, en utilisant le mode CBC, on a la possibilité de pouvoir assez facilement utiliser les techniques actuelles de remplissage de cache à partir d'une adresse donnée, si on veut accéder le plus tôt possible à une donnée qui ne se trouve pas en début de ligne [DUC 05b]. Mais le problème dans ce cas est qu'on laisse sortir sur le bus d'adresse des informations qu'on peut par ailleurs vouloir cacher (voir en particulier la section 6).

Reste le problème qu'il va falloir distribuer ces clés pour permettre aux programmeurs d'écrire du code spécifique à chaque processeur. En effet, si un attaquant se fait

passer pour un programmeur, il n'a qu'à demander les clés secrètes d'un processeur pour déchiffrer tous les programmes écrits pour ce processeur, ce qui rend caduque tout le système. Pour contourner ce problème on va classiquement utiliser des algorithmes à clé publique [COC 73, DIF 76, RIV 78] pour distribuer les clés secrètes de session qui ne seront utilisées que pour chiffrer/déchiffrer un programme donnée. En effet, on ne peut utiliser les algorithmes à clé publique pour chiffrer/déchiffrer tout dans le processeur car ils sont beaucoup moins performants que les algorithmes symétriques. En plus l'usage de clés de sessions secrètes permet éventuellement de partager des informations entre processus ou processeurs et permettre de la programmation parallèle distribuée.

Désormais, si un programmeur laisse échapper une clé secrète de session qu'il aurait oublié de détruire, un attaquant ne peut que déchiffrer le programme concerné et non plus tous les programmes destinés à ce processeur. Mais on pourrait imaginer qu'un attaquant demande à tous les programmeurs de chiffrer leur programme pour un processeur hypothétique dont l'attaquant fournirait une clé publique. Dans ce cas, l'attaquant aurait tout le loisir de déchiffrer les clés de session, puis les programmes avec leur données. Pour éviter ceci, il faut mettre en place un mécanisme de certification des processeurs dûment fabriqués.

Le choix de l'algorithme à clé publique n'est pas critique d'un point de vue vitesse, dans la mesure où il ne sert qu'à décoder un descripteur d'exécutable chiffré pour en extraire les deux clés symétriques qui servent à chiffrer et déchiffrer respectivement le programme et le contexte matériel d'exécution (registres,...), d'une part, et ses données, d'autre part. Changer de contexte nécessite donc de chiffrer l'ancien contexte pour le sauvegarder et déchiffrer le nouveau. Pour accélérer des changements de contextes entre différents processus chiffrés, on peut ajouter dans le processeur un cache de descripteurs de contexte matériel déchiffrés. C'est seulement lorsque ce cache est rempli qu'on est obligé de chiffrer/déchiffrer des contexte lorsqu'ils vont sortir du processeur.

Par contre l'algorithme à clé publique doit évidemment être sûr car, s'il est cassé, il suffit à l'attaquant d'acheter tous les logiciels avec la clé publique de son processeur : il pourra déchiffrer les logiciels et donc en faire des copies qui ne seront plus chiffrées.

Le choix de l'algorithme symétrique est plus problématique puisque c'est lui qui est utilisé pour chiffrer et déchiffrer les données entre les caches et la mémoire. Même pour des algorithmes standard tels que l'AES [Nat01] il existe des réalisations matérielles rapides comme [ALI 04] qui fonctionne à 12,5 Go/s avec une finesse de gravure standard de 0,18 μm. On peut de toute manière pipeliner encore plus mais cela pose déjà des problèmes avec les chaînages à la CBC [DWO 01]. L'autre solution est de répliquer ces algorithmes pour faire du parallélisme de données jusqu'à obtenir le débit suffisant mais dans ce cas on peut être amené à utiliser plusieurs vecteurs d'initialisation (et donc stocker en plus leur image en mémoire) si on veut éviter d'avoir plusieurs CBC en parallèle partageant le même vecteur d'initialisation ou mieux, en générant plusieurs vecteurs d'initialisation différents à partir d'un seul. Évidemment, cela rajouterait une corrélation affaiblissant plus ou moins le chiffrement (mais on pourrait

avoir une approche hybride se rapprochant d'un mode CTR). Il est clair que surtout dans ce dernier cas cela coûtera en transistors, en consommation, voire en mémoire. C'est donc à mettre en face du gain en performance.

Comme le système doit pouvoir (re)démarrer un processus en mode chiffré, on rajoute une instruction `rtiec` permettant de démarrer un processus à partir d'un descripteur de matériel chiffré avec la clé publique du processeur. De même, lorsqu'un processus chiffré est interrompu, il écrit le contexte matériel d'exécution dans une zone spéciale sous forme chiffrée avec la clé symétrique des données afin qu'il puisse éventuellement être inspecté ou modifié par le programme lui-même avec des instructions d'accès à la mémoire.

Enfin, un processus chiffré doit aussi être capable, seulement lorsqu'il le désire, d'écrire et de lire des données en clair pour faire des entrées-sorties ou communiquer avec des processus non chiffrés, à commencer par exemple avec le système d'exploitation ou l'utilisateur. On rajoute donc au processeur des instructions `stnc` et `ldnc` permettant d'accéder à la mémoire sans passer par les systèmes de chiffrement.

Le mode d'intrusion JTAG présent dans la majorité des processeurs et, plus généralement, tout mode de test ou de mise au point ne doivent pas pouvoir être utilisés en mode chiffré. Comme il faut bien mettre au point puis tester le processeur aussi en mode chiffré, il faut un système capable de supprimer cette possibilité une fois le processeur testé juste après fabrication et il ne faut pas que cette suppression ne soit contrôlée, par exemple, que par un simple bit qu'un cutter laser intrusif pourrait rétablir.

Afin de mieux résister à une attaque tentant de modifier de manière interne le microprocesseur, outre les techniques classiques, on peut rajouter des procédures de tests en faisant suivre tout chiffreur par le déchiffreur [AND 98] en vérifiant qu'on retrouve bien le même résultat. De manière générale on rajoutera des systèmes de vérification de préconditions et postconditions. Une incohérence provoquera l'arrêt du programme, voire, dans une approche plus extrémiste, le déclenchement d'un système de destruction pure et simple du processeur.

On peut aussi étendre le système en faisant voter plusieurs systèmes de chiffrement/déchiffrement. Ainsi, si le processeur est soumis à des attaques de type *glitch*, des variations de fréquence d'horloge interdites ou de tension d'alimentation, des rayonnements ionisants, des variations de température, etc. il sera fort probable qu'il y aura des comportements différents entre les chiffreurs/déchiffreurs redondants. Dans la mesure où le système vise les processeurs modernes de plus de 10^7 ou 10^8 transistors, le coût de la redondance peut être acceptable.

Pour les même raisons, la clé privée du processeur doit exister en plusieurs exemplaires à différents endroits du processeur, afin de compliquer des essais de modification.

Pour résumer cette section 4, on chiffre donc chaque bloc de mémoire en utilisant son adresse, une clé secrète et un vecteur d'initialisation aléatoire, avec, en plus, un

code d'authentification pour éviter de pouvoir injecter des blocs valides. Les adresses virtuelles sont expansées avant de passer dans la MMU pour prendre en compte le fait qu'on a rajouté des bits supplémentaires pour chaque bloc. Enfin, pour rendre pratique la distribution de programmes chiffrés et de leur clés secrètes de chiffrement, on regroupe ces informations dans un descripteur de programme chiffré qui est chiffré par la clé publique du processeur et sera déchiffré avant l'exécution avec la clé secrète du processeur. Cette dernière ne doit plus exister que dans le processeur pour des raisons de confidentialité.

5. Réalisation de CryptoPage-2

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

5.1. *Authentification et arbres de Merkle*

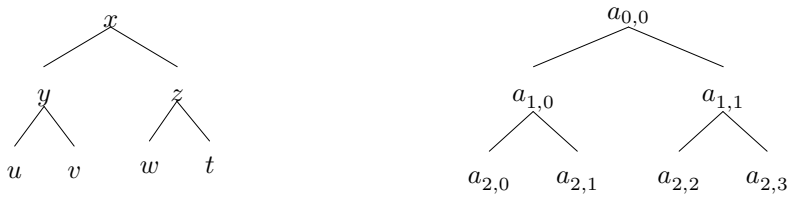
Mais en utilisant des fonctions de signature plus hiérarchiques dans ce contexte, [BLU 91, BLU 89] ont proposé d'employer une variante des arbres de MERKLE comme méthode pour effectuer des vérifications sur les mémoires exploitant la hiérarchie pour limiter les calculs.

Un arbre de MERKLE est une structure hiérarchique récursive capable de certifier et de révoquer des données. C'est une structure très employée dans les systèmes de fichiers distribués sécurisés (SFS [FU 02]) ou les réseaux pair à pair pour vérifier des données. Dans notre cas (figure 5(a)), on munit les données (nœuds à la base de l'arbre) d'une fonction de hachage à sens unique et d'une structure arborescente ainsi que d'une zone mémoire protégée qui va nous permettre de conserver le nœud racine x comme dans [SUH 03].

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

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

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



(a) Concept

(b) Arbre des données

Figure 5. *Arbres de* MERKLE.

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

Attaquer un tel système consiste à modifier les données (u, v, w, t) à image constante x. Comme on utilise une fonction de hachage cryptographique à sens unique résistant à la $2^{\text{ème}}$ pré-image, connaissant w et t (figure 5(a)), le pirate ne sait pas trouver t' associé au w' qu'il veut injecter qui donnera la même valeur z pour passer inaperçu. En plus, dans CRYPTOPAGE-2, on rend plus difficile une attaque sur H connaissant de nombreux textes chiffrés (blocs identiques de processus différents d'une part et blocs identiques à différentes adresses ayant par conséquent le même hachage) en faisant intervenir une clef secrète et l'adresse dans le calcul de H, vue alors comme une fonction d'authentification à clé secrète.

On peut noter que si H ne commute pas, on ne peut pas tenter une attaque par permutation spatiale (inverser les u et v précédents dans la mémoire). On pourrait donc se passer de faire intervenir l'adresse dans le calcul de la clé de chiffrement contrairement à [KER 00]. En fait on garde ce calcul pour éviter des attaques à texte connu (typiquement une zone de 0 en mémoire serait trop visible), dans le cas où pour accélérer le calcul de l'arbre on partait des données en clair.

5.2. *Réalisation sans cache*

On considère qu'on veut protéger une zone de n éléments mémoire (qui auront par exemple la taille des lignes de cache dans une réalisation efficace) qui seront les éléments terminaux de l'arbre représentés $a_{\log_2 n,0}$ à $a_{\log_2 n,n-1}$ sur la figure 5(b). Par mesure de simplification on considère n comme étant une puissance entière de 2. Les nœuds de l'arbre $a_{i,j}$ sont numérotés comme sur la figure 5(b), $i \in [0, \log_2 n]$ étant la profondeur dans l'arbre.

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

Tableau 1. *Algorithmes de vérifi cation et de mise à jour de la mémoire.*

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

H fonction de hachage à sens unique ;

$\mathcal{R}(i,j)$ la fonction de lecture dans la mémoire qui renvoie la valeur du nœud $a_{i,j}$ sauf pour $a_{0,0}$, la racine de l'arbre du processus en train de tourner, qui est elle stockée de manière sûre dans le processeur et sauvegardée dans le descripteur de processus chiffré au sens de [KER 00] ;

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

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

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

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

On constate que pour chaque accès vérifié il faut remonter toute une branche de l'arbre, soit dans le cas de la lecture une complexité en $\mathcal{O}(\log n)$. Pour l'écriture, comme on a à chaque étage une lecture vérifiée du nœud voisin cela fait une complexité $\mathcal{O}(\log^2 n)$ mais en supprimant la récursion de l'algorithme on peut descendre aussi en $\mathcal{O}(\log n)$ voire mieux [BER 04].

5.3. *Réalisation avec cache*

Ce type de vérificateur est coûteux en temps si on considère qu'il faut parcourir toute une branche de l'arbre pour chaque accès. Aussi, pour augmenter les performances, on peut pipeliner le mécanisme et le paralléliser. On peut aussi effectuer une exécution et une utilisation spéculative des instructions et des données à vérifier, le vérificateur générant une interruption qui bloquera le processeur et les effets de bord en cas d'erreur.

Tableau 2. *Algorithmes de vérifi cation et de mise à jour de la mémoire utilisant un cache de valeurs vérifi ées.*

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

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

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

CRYPTOPAGE a une architecture de type HARVARD pour des raisons de performance et on utilise un chiffrement du bus de données et d'instructions avec des clefs différentes pour empêcher que les instructions ne puissent s'extraire par erreur sous forme de données comme lors de l'attaque de [KUH 98]. On peut donc dupliquer les vérificateurs, avec une simplification pour la partie instructions qui n'a pas besoin de gérer les écritures.

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

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

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

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

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

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

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

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

Pour conclure cette section, on a donc rajouté en plus du chiffrement (§ 4) un mécanisme de vérification globale de l'intégrité. Ce dernier rend finalement redondant l'authentification de la section 4.2. Néanmoins il garde un intérêt si on considère que la programme aura besoin d'envoyer des données chiffrées et authentifiées à d'autre processus ou stocker des choses de manière chiffrée sur disque.

6. Chiffrement des adresses

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

C'est pour ces raisons que le chiffrement des adresses a été étudié d'un point de vue théorique [GOL 96] ou réalisé [Dal99b]. Le problème est que ces systèmes doivent être compatibles avec l'usage courant d'un processeur général : efficaces, être utilisables par des systèmes d'exploitation classiques et des systèmes de gestion mémoire (MMU).

Comme la gestion mémoire est l'apanage des systèmes d'exploitation et qu'on ne veut pas forcément faire tourner un système par cryptoprocessus, on a intérêt à avoir un système de chiffrement des adresses compatible avec les systèmes d'exploitation classiques. Concrètement le système va voir les adresses dans les TLB [5], donc il faut qu'elles soient chiffrées avant d'arriver dans la MMU. En outre, on ne veut pas casser la hiérarchie du système et on a besoin d'avoir des zones chiffrées et des zones non chiffrées [KER 00], par exemple pour les entrées-sorties en clair.

Dans un système paginé classique, une adresse est de type $A = p\|l\|d$, c'est à dire hiérarchisée avec la concaténation des bits respectifs du n° de page, de ligne de cache

5. Translation Lookaside Buffer : entrée du cache de traduction d'addresse virtuelle vers physique utilisé pour accélérer la MMU.

et de déplacement dans cette dernière. Pour limiter les problèmes, on va coller à cette organisation et trouver un compromis entre opacité et usage.

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

Ensuite, au niveau de la plage d'adresses chiffrées, l'adresse est chiffrée au niveau de p et de l séparément. Cela signifie que la localité dans la page est respectée mais que, par contre, cela fournira plus d'information à l'attaquant. Au sein de la ligne de cache, c'est le système de chiffrement par bloc standard de CRYPTOPAGE qui brouille les bits de données et donc l'adresse ne sort pas au niveau du bus, tout se gère au niveau du cache. Évidemment, si on utilise un chiffrement style CBC et un chargement au plus tôt, on est amené à chiffrer aussi l. On pourrait imaginer étendre ce mélange des données au niveau d'une page mais cela nécessiterait d'avoir une page entière dans le processeur avant de pouvoir lire ou écrire ne serait-ce qu'un bit dans une page.

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

Il est certain que l'analyse macroscopique des flux de pages anonymisées fournira encore de l'information à un attaquant mais beaucoup moins qu'avec l'usage d'adresses non chiffrées. On aura tout intérêt à coupler aux techniques matérielles des techniques logicielles d'opacification de code (générateur de programmes spaghetti avec des branchements inutiles dans tous les sens,...) [NAU 03]. Il est clair que les performances pourront en pâtir mais les techniques classiques de cache et plus récemment de cache de trace devraient les compenser en grande partie. On peut rendre ce chiffrement des adresses optionnel pour fournir au programmeur le choix de pouvoir exécuter plus rapidement son programme au contrepartie d'un mode d'adressage plus clair pour l'attaquant.

7. Mise en œuvre logicielle

Après avoir défini la réalisation matérielle il faut s'intéresser à l'aspect logiciel et ici plus particulièrement dans un contexte UNIX [VAH 96], système d'exploitation dont de nombreux sources sont disponibles, facilitant la mise en place d'un tel système, et qui a en plus l'avantage de bien fonctionner et d'être robuste.

Un système fonctionnel sera constitué du noyau UNIX permettant l'exécution coexistante de processus utilisateurs ou superviseur (« `root` »), chacun pouvant être chiffré ou non.

La contrainte est que ni un processus superviseur, ni même le noyau ne peuvent surveiller ce qui se passe dans un processus chiffré, accéder à ses données, son code ou des registres. Par contre un processus chiffré avec un clé c_s peut partager de l'information directement de manière chiffrée avec un autre processus chiffré avec la même clé via des fichiers, des *sockets*, des segments de mémoire partagée, etc. Dans le cas de segments de mémoire partagée chiffrée, il faut qu'ils soient à la même adresse virtuelle dans chaque processus puisque le chiffrement dépend de l'adresse virtuelle. Avec la généralisation des espaces d'adressages en 64 bits, cette contrainte est facilement surmontable.

Cela signifie qu'un processus chiffré n'est pas traçable, ni espionnable par un autre processus qui n'a pas la clé c_s car, si le système d'exploitation est bien capable d'arrêter un processus chiffré et de récupérer son contexte matériel, il ne sait ni l'interpréter ni le modifier sans la clé.

Comme la mise au point des programmes est difficile, un programmeur peut ponctuellement chiffrer un déboggueur (ou tout au moins un petit mandataire qui s'interfacera avec le vrai déboggueur) avec la clé publique d'un client combinée à la clé symétrique (aléatoire) de son produit. Il aura ainsi un déboggueur (ou son mandataire tout au moins) qui ne tournera que sur le poste client et qui ne sera capable que de tracer l'exécution de sa copie spécifique du logiciel.

C'est une technique similaire qui permet de faire tourner un programme sur des multiprocesseurs ou des grilles de calcul. On a autant de programmes identiques chiffrés avec les mêmes clés symétriques pour le partage de données voire de programmes et dont seuls les descripteurs chiffrés changent, car ils sont chacun encodés avec la clé publique de chaque processeur.

Pour clarifier la suite, on rappelle quelques hypothèses matérielles du système CRYPTOPAGE :

– le processeur peut être en mode chiffré ou non ;

– le processeur peut être en mode superviseur ou non ;

– en mode chiffré, on ne peut influencer l'exécution depuis l'extérieur si ce n'est en interrompant l'exécution ;

– il n'y a aucun moyen d'accéder aux registres lors du mode chiffré (en dehors du processus en cours d'exécution bien sûr), que ce soit par des moyens matériels, des registres internes, des remises à 0, des interruptions logicielles ou matérielles,...

– le seul moyen de démarrer une exécution chiffrée est de partir d'un descripteur de contexte d'exécution chiffrée qui est alors déchiffré via la clé secrète du processeur ;

– un contexte d'exécution chiffrée contient une clé de chiffrement symétrique du programme, une clé de chiffrement symétrique des données, des contenus de registres, la valeur de la racine de l'arbre de vérification ;

– si un programme chiffré est interrompu, son état (registres,...) est stocké dans un nouveau descripteur de contexte chiffré avec la clé secrète du processeur. Ainsi le programme peut-être continué plus tard sans fuite d'information.

7.1. *Appels au système*

Un processus chiffré doit pouvoir utiliser les services du noyau en utilisant classiquement les appels systèmes. Afin que le noyau comprenne les services demandés, le processus chiffré doit déchiffrer les arguments passés avant de faire un *trap* dans le noyau.

Pour ce faire, il faut modifier la bibliothèque d'interfaçage système UNIX pour qu'elle déchiffre les arguments dans une zone tampon intermédiaire avant de faire le *trap*. Comme la partie de chiffrement dépend du programme considéré, elle ne peut pas faire partie d'une bibliothèque dynamique globale comme c'est la cas normalement. Si l'interface système au niveau *trap* ne change pas, cela permet de garantir la compatibilité ascendante et donc qu'un programme chiffré marchera encore sur des versions ultérieures du système[6], même si la bibliothèque système est liée statiquement à la compilation.

Certains appels systèmes doivent être proposés en version chiffrée mais aussi en version non chiffrée. Par défaut, pour des raisons de sécurité, les appels systèmes font des transferts chiffrés. Par exemple `write()` permettra d'écrire dans un fichier[7] et les données écrites le seront de manière chiffrée et donc illisibles par un attaquant. Afin de garder la sémantique UNIX des appels systèmes (si on veut écrire n octets physiquement en mémoire, qu'ils soient chiffrés ou non, c'est bien n qu'on passe en paramètre) les contraintes de format des données chiffrées devront être gérées : traitement des données par bloc et donc données de taille multiple d'une taille de bloc.

Mais il faut aussi pouvoir avoir des entrées-sorties non chiffrées pour communiquer avec l'utilisateur ou d'autres programmes. La bibliothèque système et éventuellement les autres bibliothèques proposeront des fonctions suffixées de `_nc` qui (dé)chiffreront à la volée les données transférées via des tampons allouées pour l'occasion. Les performances seront donc plus faibles que les appels systèmes communiquant directement en mode chiffré. La fonction `write_nc()` permettra par exemple d'écrire des données non chiffrées (en clair) dans un fichier.

On retrouvera cette distinction au niveau des bibliothèques standard. Typiquement les fonctions d'entrées-sorties tamponnées de la bibliothèque standard du C existeront en version capable d'accéder à des fichiers chiffrés et en version pour des fichiers non chiffrés.

Les fonctions d'allocation mémoire de type `malloc()`, `calloc()` appelées dans un processus chiffré doivent demander au système d'exploitation (via les appels `sbrk()` ou `mmap()`) de fournir la zone pour permettre en plus le stockage de l'authentification et du vecteur d'initialisation et tenir compte d'une taille de blocs entiers.

---

6. C'est par un mécanisme similaire qu'on arrive par exemple sur Sun à faire marcher des programmes SunOS 4.x en SunOS 5.x : en gérant une sémantique de *trap* SunOS 4.x au dessus de SunOS 5.x.
7. Au sens général d'UNIX, c'est à dire une entité représentable par un descripteur de fi chier.

Notons que du point de vue du programmeur, c'est transparent et il peut continuer de faire de l'arithmétique de pointeur comme d'habitude.

Comme le chiffrement dépend des adresses, cela pose un problème si on écrit à une adresse virtuelles des données et qu'ont les relit à d'autres adresses, par exemple si on partage un segment de mémoire entre 2 processus à différentes adresses ou qu'on lit un fichier via un tampon à une autre adresse que celle utilisée lors de l'écriture. Pour gérer ces problèmes on peut rajouter une instruction de lecture et d'écriture acceptant en plus une adresse spécifiquement utilisée pour le chiffrement.

7.2. *Création de processus chiffrés*

La création d'un nouveau processus en UNIX se fait par clonage via un appel à une fonction de type `fork()`. S'il s'agit d'un processus chiffré, `fork()` créera un nouveau processus chiffré.

L'exécution d'un autre programme dans un processus UNIX est réalisée par un appel de type `exec()`. Si cet appel système rencontre un fichier exécutable de type chiffré il démarrera le programme en exécutant une instruction de type `rtiec` sur un descripteur de contexte matériel présent dans le format exécutable après l'avoir mis en mémoire via un `mmap()`. Dans la réalisation actuelle, le code contient au début un code de *bootstrap* qui a juste pour effet d'exécuter l'instruction `rtiec` qui continue alors en mode chiffré. La gestion des initialisations mémoire est faite par le programme et non le système d'exploitation car ce dernier ne contrôle pas le contenu de la mémoire du processus [DUC 04]. De même que pour l'exécution d'un programme non chiffré, le processus résultant hérite des droits, des descripteurs de fichiers, des variables d'environnement et du statut de traitement des signaux [VAH 96].

Les variables d'environnement doivent être chiffrées au démarrage du programme si le processus qui lance l'exécution d'un programme chiffré n'était pas un processus chiffré. C'est rajouté dans le code de *bootstrap* par le compilateur ou l'éditeur de lien lorsqu'est construit le programme chiffré. Cela veut dire aussi qu'afin de transférer ces variables d'un processus chiffré à un autre processus chiffré lors d'un `exec()` il faut un structure de donnée plus relogeable en mémoire que l'actuel `char **environ` puisqu'une fois chiffrée, le système d'exploitation n'est plus capable de retrouver la structure des ces chaînes de caractères.

7.3. *Signaux*

Les signaux en UNIX sont des interruptions logicielles qui se traduisent par des appels de fonctions spécifiques d'un processus par le noyau lorsque le processus reçoit un signal [VAH 96]. Concrètement cela signifie que le noyau, qui fonctionne a priori en mode non chiffré, doit appeler une fonction (*a priori* chiffrée) du processus chiffré avec en paramètre le numéro de signal, des informations supplémentaires sur le signal et le contexte d'exécution au moment où a eu lieu le signal.

Ces deux dernières informations contiennent une partie provenant du système d'exploitation qui est donc non chiffrée ainsi qu'une partie représentant le contexte matériel du processus qui est en format chiffré. Il faut donc que l'édition de lien rajoute une sur-couche chiffrant les éléments non chiffrés avant exécution de la fonction gérant le signal dans le programme chiffré.

La réincarnation du processus chiffré dans le contexte du signal est semblable au démarrage d'un processus chiffré via un appel système de type `exec_c()` et nécessiterait donc une instruction de type `rtiec` prenant en paramètre un contexte matériel chiffré. Mais, par rapport à un démarrage de processus, la mémoire ne doit pas être réinitialisée dans un état conforme à celui résumé dans le contexte matériel chiffré. Il faut donc rajouter un nouveau mécanisme matériel permettant de démarrer un flot d'exécution à un endroit prédéterminé du programme avec possibilité au système d'exploitation de passer des paramètres, par exemple dans les registres. On peut imaginer avoir une syntaxe de type `sig` *`desc`* où *`desc`* est l'adresse d'un descripteur en mémoire. Cette partie est plus détaillée dans [DUC 05a].

L'enregistrement d'une fonction auprès du système d'exploitation est fait par l'appel système `sigaction_c()` (au lieu du classique `sigaction()`) qui prendra en paramètre un descripteur de contexte matériel chiffré initialisé pour exécuter la fonction (au lieu de prendre classiquement un pointeur de fonction). Contrairement à ce qui est fait dans UNIX normalement, ce sera du ressort du programme de gérer ensuite l'appel des bonnes fonctions lorsque le système d'exploitation lui rendra la main suite à une instruction `sig`.

7.4. *Compilation*

La compilation d'un source pour produire un binaire exécutable en mode chiffré est assez standard [VAH 96] mais le format des exécutables doit néanmoins être étendu pour comporter : un entête indiquant qu'on a affaire à un exécutable chiffré ; un descripteur chiffré avec la clé publique du processeur qui contient la clé symétrique de chiffrement de l'exécutable et le contexte matériel chiffré permettant de lancer l'exécution du programme, à la place de la simple adresse de démarrage classique ; une zone *text* comportant le code exécutable chiffré ; une zone *initialized data* contenant sous forme chiffrée les objets initialisés du programme ; une zone *uninitialized data* (ou BSS) contenant les variables non initialisées qui seront remplies par des 0 (chiffrés) lors du démarrage du programme ; une éventuelle table des symboles chiffrée pour un déverminage par un débogueur chiffré.

La phase de compilation en elle-même est inchangée : on génère du binaire non chiffré. Par contre l'édition de liens est modifiée pour générer dans la phase finale des instructions et des données chiffrées. Dans notre réalisation actuelle [DUC 04] on a utilisé le fait que l'éditeur de lien GNU `ld` possède un langage de programmation qui permet de faire de nombreuses manipulations de sections et de symboles plus simplement.

La table des symboles est étendue avec des indications de chiffrement des paramètres d'entrée ou de sortie des fonctions afin qu'une fonction puisse éventuellement être appelée par une fonction extérieure au programme non chiffrée (typiquement une fonction du système d'exploitation voulant exécuter une fonction associée à un signal). À partir de cette description « à la IDL » de CORBA, l'éditeur de lien peut emballer la fonction avec une fonction qui chiffrera avant et déchiffrera après l'appel à la fonction du programme.

Si un programme s'intéresse finement à son allocation mémoire, typiquement pour externaliser ses données internes en mode chiffré, il faudra qu'il tienne compte du fait que les données en mémoire virtuelle externe au cache prennent plus de place à cause de la signature des blocs.

Il faudra fournir au programmeur des fonctions permettant d'avoir simplement cette information.

8. Autres travaux du domaine et travaux futurs

Les recherches sur le matériel dans le domaine se sont tournées vers des petits systèmes [BES 80, BES 81, BES 84, Dal99a, Dal99b] et de manière plus générale sur les cartes à puce plutôt que de viser un processeur généraliste et un système d'exploitation standard devant travailler avec différents niveaux de sécurité permettant de concilier puissance de calcul et sécurité. Le problème des attaques par rejeu était ignoré.

Récemment, un groupe d'industriels a proposé une solution d'informatique de confiance (projet TCPA [TCP03, ENG 03]) reposant sur une puce fournissant des primitives cryptographiques à un micro-noyau digne de confiance et à des applications ainsi que des attestations garantissant l'état de la machine (matériel, système d'exploitation, applications). Cette solution ne prend malheuresement pas en compte les attaques physiques, notamment contre les bus du processeur.

Jusqu'à présent, il y a eu relativement peu d'études dans le domaine des systèmes d'exploitation classiques pour processeurs incluant une exécution chiffrée. On peut néanmoins citer [LIE 03b] qui présente un système d'exploitation non digne de confiance destiné à s'exécuter sur l'architecture sécurisée XOM [LIE 03a].

Il existe d'autres approches telle que la location de procédures par exemple mais cela nécessite d'interroger un serveur distant régulièrement et cette technique ne résiste pas au traçage du programme.

En ce qui concerne CRYPTOPAGE, on est encore loin d'avoir un système fonctionnel, tant au niveau architectural que logiciel, car les domaines impliqués sont large et il reste certainement plein de problèmes à résoudre qui sont à peine entrevus, tel que le partitionnement efficace du cache entre plusieurs processus différents, chiffrés ou pas, empêchant toute fuite d'information.

D'un point de vue matériel, il reste à modéliser plus finement le système à l'aide d'un simulateur de processeur afin de voir l'implication sur les performances en fonction des différentes tailles de cache et du surcoût mémoire acceptable. C'est un travail qui a été commencé en utilisant le simulateur SimpleScalar [LAU 03] et qui continue de manière plus ambitieuse [DUC 04] avec le simulateur d'un ordinateur PC complet BOCHS [BOC04] dans lequel on a rajouté les instructions CRYPTOPAGE supplémentaires au Pentium 4 virtuel, en mettant plus l'effort sur les fonctionnalités que sur les mesures de performance. Actuellement seul le mode de chiffrement sans authentification fonctionne. Les signaux ne sont pas encore gérés en mode chiffré (pas d'instruction `sig`).

On peut s'attendre néanmoins à ce que les performances restent bonnes car le système est vu comme un ralentissement entre le cache intérieur au processeur le plus externe et le monde extérieur. De plus, la partie chiffrement peut être parallélisée et pipelinée jusqu'à obtenir les performances requises, car le nombre de transistors n'est plus un problème même si c'est à prendre en considération en terme de coût et de consommation électrique. Néanmoins il y a une part de latence incompressible à laquelle il faudra essayer de s'adapter avec les techniques actuelles déjà utilisées dans les processeurs pour rendre plus tolérable la latence mémoire, par exemple en utilisant des modèles d'exécution très multithreadés ou SMT. Une partie du débit mémoire va aussi servir à transférer des vecteurs d'initialisation CBC et à gérer l'arbre de MERKLE. Du coup, on aura aussi un surcoût en terme de débit mémoire qui va ralentir l'application.

Au niveau logiciel, de manière plus prospective, il faut continuer l'étude de l'adaptation d'un système d'exploitation pour utiliser toutes les primitives de base décrites en § 7, comment gérer les zones non utilisées liées au caractère très creux de la mémoire virtuelle, etc. Actuellement nous faisons tourner sur notre BOCHS modifié un LINUX 2.6.10 adapté pour gérer les processus chiffrés [DUC 04]. Il reste un gros travail à faire dans le domaine des mesures de performances car contrairement à d'autres projets du domaine on a préféré mettre en avant une simulation fonctionnelle complète plutôt que des mesures de performances locales utilisant des traces d'exécution.

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

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

De manière plus générale, il faudrait aussi développer un modèle de niveaux de sécurité et compléter le système avec une vérification globale de cohérence automatisée pour vérifier la sécurité de tout le système.

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

En parallèle à une première version du système qui fait l'objet d'une thèse qui débute, certains concepts sont aussi utilisés dans le cadre du projet OPENSMARTCARD en collaboration avec l'ENST Paris. Il s'agit de concevoir dans un premier temps une carte à puce libre basée, d'une part, sur une architecture CRYPTOPAGE simplifiée pour la protection logique du système et, d'autre part, sur une microarchitecture utilisant des portes logiques symétrisées sans syndrome électrique et de la logique asynchrone pour résister aux attaques d'analyse différentielle de consommation électrique style DPA [KOC 99]. En plus, dans la suite de ce projet, une version de Leon (SPARC V8) sur FPGA est en cours d'adaptation pour intégrer des mécanisme de CRYPTOPAGE, ce qui nous permettra d'avoir des mesures de performances plus directe.

9. Conclusion

Nous avons présenté une architecture minimale au sens du RISC permettant de sécuriser l'exécution de processus au sein d'un processeur en préservant la confidentialité des informations échangées avec la mémoire, instructions et données, même si le système d'exploitation est corrompu. La mémoire étant signée globalement, il n'est pas possible de modifier discrètement des parties de celle-ci afin de modifier le comportement d'un processus et d'en extraire de l'information. De nombreux compromis sécurité-performance sont possibles pour ne pas trop pénaliser chaque partie du processus.

Une première implémentation simplifiée a été faite avec le simulateur de PC BOCHS et permet de faire tourner un LINUX modifié pour faire tourner aussi des processus chiffrés. Le développement cohérent du système d'exploitation, du matériel et des outils logiciels nous semble le seul moyen d'arriver à avoir une solution complète sans oublier certains aspects. Malheureusement cela explique qu'on n'ait aucune mesure précise des performances pour l'instant.

Le déploiement de ces techniques, que ce soit CRYPTOPAGE ou d'autres [SUH 03, LIE 03a, LIE 00], dans tout processeur est amené à augmenter la sécurité dans de nombreux domaines de l'informatique et à faire émerger de nouvelles applications sécurisées : album de cartes à puce virtuelles, stockage sécurisé de clefs privées, code mobile à exécution garantie, grilles de calcul et ordinateurs parallèles très répartis enfin sécurisés, des services *web* de confiance, des routeurs sécurisés, des systèmes de vérification des droits d'usage, des antivols, etc.

Les techniques présentées doivent permettre de rendre quasi-impossible le piratage logiciel par des moyens standard [AND 98]. Mais pour atteindre ce but il faut que tous les maillons de la chaîne soient consolidés. Au niveau matériel, il faut mettre en œuvre le plus de techniques de blindages pour résister aux attaques intrusives ou passives (analyse de consommation [KOC 99], de temps d'exécution [KOC 96],...). Ensuite, si

le programme exécuté possède des failles, ces failles seront exécutées de manière... sécurisées. Il faut donc aussi mettre l'accent sur les techniques de preuve, les analyses de programme, la compilation.

Le déploiement de ces techniques de chiffrement et de preuve de mémoire se traduit aussi par une augmentation de la latence mémoire qui est le problème majeur en architecture des ordinateurs. Toutes les techniques développées dans le but de mieux tolérer cette latence seront donc bienvenues.

Il est assez surprenant de constater qu'actuellement les constructeurs grand public sont plus motivés par les problèmes de piratage de contenus que par les problèmes de sécurité informatique en soi alors que la sécurité d'exécution devrait être un aspect primordial même si cela ne rajoute pas de fonctionnalité à la résolution d'un problème : dans le cas général, un programme sécurisé ou pas fournira le même résultat.

Le contenu numérique artistique, même chiffré, pose un problème dans la mesure où il est toujours copiable au moment où il est rematérialisé sous forme d'images ou de son. On peut donc en faire des copies, mêmes si elle ne sont pas faites en numérique et sont donc des copies de moindre qualité. Le développement de contenus plus interactifs pourrait freiner cette facilité de copie [KOC 03].

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

Comme toute technologie, on peut aussi imaginer d'autres utilisations moins nobles : processeurs vérouillés sur un système d'exploitation, crypto-mouchards indéchiffrables, virus exécutés de manière sécurisée par des logiciels troués, etc. Le tout garanti par le fabricant du processeur. Pourra-t-on lui faire confiance ? Qu'est-ce qui prouvera que les services secrets d'une grande puissance n'auront pas installé une porte dérobée dans le matériel ? Peut-être sera-ce l'occasion de développer des processeurs libres performant au même titre que les logiciels libres dans le contexte des logiciels de sécurité qui autorisent un contrôle plus facile du contenu par des pairs. Les prix de développement de processeurs performants sont exorbitants mais ceux des systèmes d'exploitation aussi et pourtant il a été montrée qu'avec une communauté suffisante c'est possible.

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

Remerciements

Nous remercions Mathieu CHEVRIER, élève de 3ème année en 1999–2000, pour sa première étude bibliographique sur la question. Merci aux membres du projet incitatif OPENSMARTCARD du GET pour leur discussions passionnantes et tout particulièrement à Sylvain GUILLEY et Renaud PACALET pour leurs commentaires constructifs sur le projet. Le projet a bénéficié d'un financement dans le cadre de l'action incitative OPENSMARTCARD du Groupement des Écoles des Télécommunications et surtout d'une bourse de thèse de la DGA. Enfin, grand merci aux relecteurs pour leur attentive inspection et en particulier celui qui a su retrouver l'algorithme d'écriture vérifiée qui avait disparu suite à un couper-coller malencontreux.

10. Bibliographie

[ALI 04] ALIREZA HODJAT I. V., « High-Throughput Programmable Cryptocoprocessor », *IEEE Micro*, vol. 24, n° 3, 2004, p. 34–45, `http://www.ee.ucla.edu/~ahodjat/papers/alireza_ieee_micro.pdf`.

[AND 98] ANDERSON R. J., KUHN M., « Low Cost Attacks on Tamper Resistant Devices », CHRISTIANSON B., Ed., *Security protocols : 5th international workshop, Paris, France, April 7–9, 1997 : proceedings*, vol. 1361 de *Lecture Notes in Computer Science*, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998, Springer-Verlag.

[BER 04] BERMAN P., KARPINSKI M., NEKRICH Y., « Optimal Trade-Off for Merkle Tree », rapport n° TR04-049, juin 2004, Electronic Colloquium on Computational Complexity, `http://eccc.uni-trier.de/eccc-reports/2004/TR04-049`.

[BES 80] BEST R. M., « Preventing Software Piracy with Crypto-Microprocessors », *Proc. IEEE Spring COMPCON'80*, février 1980, p. 466–469.

[BES 81] BEST R. M., « Crypto Microprocessor for Executing Enciphered Programs », rapport n° US4278837, juillet 1981, United States Patent, Consulté le 21 février 2000 sur `http://patent.womplex.ibm.com/details?\&pn=US04278837\_\_`.

[BES 84] BEST R. M., « Crypto Microprocessor that Executes Enciphered Programs », rapport n° US4465901, août 1984, United States Patent, Consulté le 21 février 2000 sur `http://patent.womplex.ibm.com/details?\&pn=US04465901\_\_`.

[BLU 89] BLUM M., KANNAN S., « Designing programs that check their work », *STOC89*, 1989, p. 86–97.

[BLU 91] BLUM M., EVANS W. S., GEMMELL P., KANNAN S., NAOR M., « Checking the Correctness of Memories », *IEEE Symposium on Foundations of Computer Science*, 1991, p. 90-99.

[BOC04] « Bochs —think inside the bochs », février 2004, `http://bochs.sourceforge.net`.

[COC 73] COCKS C., « A note on 'non-secret encryption' », rapport, novembre 1973, Government Communications Headquarters, Cheltenham, England, `http://www.cesg.gov.uk/site/publications/media/notense.pdf`.

[COX 01] COX I., MILLER M., BLOOM J., MILLER M., *Digital Watermarking*, Morgan Kaufmann, 2001.

[Dal99a] Dallas Semiconductor, « DS5000FP Soft Microprocessor Chip », novembre 1999, http://www.dalsemi.com/DocControl/PDFs/5000fp.pdf.

[Dal99b] Dallas Semiconductor, « DS5002FP Secure Microprocessor Chip », mai 1999, http://www.dalsemi.com/DocControl/PDFs/5002fp.pdf.

[DIF 76] DIFFIE W., HELLMAN M. E., « New Directions in Cryptography », *IEEE Transactions on Information Theory*, vol. 22, n° 6, 1976, p. 644–654.

[DUC 04] DUC G., « CryptoPage —An architecture to run secure processes », Diplôme d'Étude Approfondie, ENSTBr, DEA de l'université de Rennes 1, juin 2004, http://www.lit.enstb.org/~keryell/eleves/ENSTBr/2003-2004/DEA/Guillaume.Duc.

[DUC 05a] DUC G., KERYELL R., « The concept of secure processes for LINUX on the CRYPTOPAGE/x86 secure architecture », *20th ACM Symposium on Operating Systems Principles – SOSP'2005*, octobre 2005, Submitted.

[DUC 05b] DUC G., KERYELL R., « Portage de l'architecture sécurisée CRYPTOPAGE sur un microprocesseur x86 », *Symposium en Architecture de Machines (SympA'2005)*, vol. soumis, Le Croisic, presqu'île de Guérande, France, avril 2005, http://www.lit.enstb.org/~keryell/publications/conf/2005/SympA.

[DWO 96] DWORK C., LOTSPIECH J., NAOR M., « Digital signets : self-enforcing protection of digital information », *28th Symposium on the Theory of Computation*, 1996, p. 489–498.

[DWO 01] DWORKIN M., « Recommendation for Block Cipher Modes of Operation —Methods and Techniques », NIST Special Publication n° 800-38A, 2001, National Institute of Standards and Technology, http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[ENG 03] ENGLAND P., LAMPSON B., MANFERDELLI J., PEINADO M., WILLMAN B., « A Trusted Open Platform », *Computer*, vol. 36, n° 7, 2003, p. 55–62.

[ESK 03] ESKICIOĞLU A. M., « Protecting Intellectual Property in Digital Multimedia Networks », *Computer*, vol. 36, n° 7, 2003, p. 39–45.

[ESP98] « IP Encapsulating Security Payload (ESP) », novembre 1998, http://www.ietf.org/rfc/rfc2406.txt.

[FU 02] FU K., KAASHOEK M. F., MAZIÈRES D., « Fast and secure distributed read-only file system », *ACM Transactions on Computer Systems*, vol. 24, n° 1, 2002, p. 1–24, http://www.scs.cs.nyu.edu/~dm/sfsro-tocs.pdf.

[GAS 03] GASSEND B., CLARKE D., SUH G. E., VAN DIJK M., DEVADAS S., « Caches and Hash Trees for Efficient Memory Integrity Verification », *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, février 2003.

[GIL 98] GILMONT T., LEGAT J.-D., QUISQUATER J.-J., « An Architecture of Security Management Unit for Safe Hosting of Multiple Agents », *Proceedings of COST 254*, novembre 1998, p. 79–82, http://ldos.fe.uni-lj.si/cost254/papers/022.ps.

[GOL 96] GOLDREICH O., OSTROVSKY R., « Software Protection and Simulation on Oblivious RAMs », *Journal of the ACM*, vol. 43, n° 3, 1996, p. 431–473.

[HUA 02] HUANG A., « Keeping Secrets in Hardware : the Microsoft XBox (TM) Case Study », rapport n° AI Memo 2002-008, mai 2002, MIT.

[IPS03] « IP Security Protocol (ipsec) », 2003, http://www.ietf.org/html.charters/ipsec-charter.html.

[KER 00] KERYELL R., « CryptoPage-1 : vers la fi n du piratage informatique ? », *Symposium d'Architecture (SympA'6)*, Besançon, France, juin 2000, p. 35–44, http://www.cri.ensmp.fr/~keryell/publications/ENSTBr_INFO_2000-001.

[KOC 96] KOCHER P. C., « Timing Attacks on Implementations of Diffi e-Hellman, RSA, DSS, and Other Systems », *Advances in Cryptology — Crypto'96 : 16th Annual International Cryptology Conference*, n° 1109 LNCS, Springer-Verlag, août 1996, p. 104-113, http://citeseer.ist.psu.edu/kocher96timing.html.

[KOC 99] KOCHER P. C., JAFFE J., JUN B., « Differential Power Analysis », *Advances in Cryptology - Crypto'99 : 19th Annual International Cryptology Conference on Advances in Cryptology*, n° 1666 LNCS, août 1999, p. 388–397, http://citeseer.ist.psu.edu/kocher99differential.html.

[KOC 03] KOCHER P., JAFFE J., JUN B., LAREN C., LAWSON N., « Self-protecting digital content », white paper, avril 2003, Cryptography Research Inc, http://www.cryptography.com/technology/spdc.

[KUH 98] KUHN M. G., « Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP », *IEEE Transactions on Computers*, vol. 47, n° 10, 1998, p. 1153–1157.

[LAU 03] LAURADOUX C., « Les cryptoprocesseurs et la preuve de mémoire », Diplôme d'Étude Approfondie, ENSTBr, DEA de l'université de Rennes 1, septembre 2003, http://www.lit.enstb.org/~keryell/eleves/ENSTBr/2002-2003/DEA/Lauradoux.

[LIE 00] LIE D., THEKKATH C. A., MITCHELL M., LINCOLN P., BONEH D., MITCHELL J. C., HOROWITZ M., « Architectural Support for Copy and Tamper Resistant Software », *Architectural Support for Programming Languages and Operating Systems*, 2000, p. 168-177.

[LIE 03a] LIE D., THEKKATH C., MITCHELL J., HOROWITZ M., « Specifying and Verifying Hardware for Tamper-Resistant Software », *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, mai 2003.

[LIE 03b] LIE D., TREKKATH C. A., HOROWITZ M., « Implementing an Untrusted Operating System on Trusted Hardware », *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03)*, octobre 2003, p. 178–192.

[MEN 97] MENEZES A. J., VAN OORSCHOT P. C., VANSTONE S. A., *Handbook of applied cryptography*, The CRC Press series on discrete mathematics and its applications, CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[NAO 03] NAOR D., NAOR M., « Protecting Cryptographic Keys : The Trace-and-Revoke Approach », *Computer*, vol. 36, n° 7, 2003, p. 47–53.

[Nat01] National Institute of Standards and Technology, « Announcing the Advanced Encryption Standard (AES) », federal information processing standards publications 197 édition, novembre 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NAU 03] NAUMOVICH G., MEMON N., « Preventing Piracy, Reverse Engineering, and Tampering », *Computer*, vol. 36, n° 7, 2003, p. 64–71.

[RIV 78] RIVEST R. L., SHAMIR A., ADLEMAN L., « A Method for Obtaining Digital Signatures and Public-Key Cryptosystems », *Communications of the ACM*, vol. 21, n° 2, 1978, p. 120–126, http://theory.lcs.mit.edu/~rivest/rsapaper.pdf.

[SKO 02a] SKOROBOGATOV S. P., « Low temperature data remanence in static RAM », rapport, juin 2002, University of Cambridge Computer Laboratory.

[SKO 02b] SKOROBOGATOV S. P., ANDERSSON R., « Optical Fault Induction Attacks », *Cryptographic Hardware and embedded Systems Workshop,* août 2002.

[SUH 03] SUH G. E., CLARKE D., GASSEND B., VAN DIJK M., DEVADAS S., « The AEGIS Processor Architecture for Tamper-Evident and Tamper-Resistant Processing », rapport, février 2003, MIT Laboratory for Computer Science.

[TCP03] « Trusted Computer Platform Alliance », 2003, `http://www.trustedcomputing.org`.

[TRA 03] TRAW C. B. S., « Technical Challenges of Protecting Digital Entertainment Content », *Computer*, vol. 36, n° 7, 2003, p. 72–78.

[VAH 96] VAHALIA U., *Unix Internals : the New Frontiers*, Prentice-Hall, 1996.

[WAR 70] WARE W. H., « Security Controls for Computer Systems », rapport, 1970, The Rand corporation, `http://www.rand.org/publications/R/R609.1/R609.1.html`.

[WEI 00] WEINGART S. H., « Physical Security Devices for Computer Subsystems : A Survey of Attacks and Defenses », *Cryptographic Hardware Embedded Sytems*, 2000.

Article reçu le 5/3/2004.
Version révisée le 4/4/2005.
Rédacteur responsable : JEAN-LOUIS GIAVITTO

***Guillaume* DUC** *est titulaire d'un diplôme d'ingénieur de l'ENST Bretagne (2004) et d'un diplôme d'études approfondies de l'Université de Rennes 1 (2004). Il prépare actuellement une thèse dirigée par Jacques* STERN *& Ronan* KERYELL *et financée par une bourse* DGA. *Ses travaux de recherche concernent les architectures materielles et logicielles sécurisées.*

***Ronan* KERYELL** *est ancien élève de l'*ENS *(École Normale Supérieure) de la rue d'Ulm (1986) où il a effectué sa thèse en informatique (1992). Il a travaillé comme enseignant-chercheur au Centre de Recherche en Informatique de l'École des Mines de Paris et est au Département Informatique de l'*ENSTB*r depuis 1999. Ses centres d'intérêt tournent autour de l'informatique et l'architecture à haute performance, la compilation, les systèmes d'exploitation et la sécurité.*

***Cédric* LAURADOUX** *est diplômé de l'*ENIB *(École Nationale d'Ingénieur de Brest) (2003) et a effectué son stage de* DEA *sur* CRYPTOPAGE *à l'*ENSTB*r (2003). Il est actuellement en thèse à l'*INRIA *dans le projet* CODES *sur la génération d'aléas en utilisant la complexité architecturale des processeurs modernes. Il s'intéresse à l'architecture des processeurs et à la cryptographie.*