RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Support architectural pour identification de programmes chiffrés dans une architecture sécurisée sans système d'exploitation de confiance

Guillaume Duc et Ronan Keryell

TÉLÉCOM Bretagne
CS 83818
29238 Brest Cedex 3, France
{guillaume.duc,ronan.keryell}@telecom-bretagne.eu

Résumé
Plusieurs architectures informatiques sécurisées ont été proposées ces dernières années afin d'offrir aux applications un environnement d'exécution sûr. Ces architectures utilisent des mécanismes de chiffrement et de vérification de l'intégrité de la mémoire pour garantir aux applications qu'un attaquant ne peut pas espionner les données ou le code du processus ni perturber sa bonne exécution. Les applications qui s'exécutent dans cet environnement d'exécution sont chiffrées et donc totalement opaques, y compris pour le propriétaire de l'architecture. Cependant, dans certains cas, un utilisateur peut vouloir avoir accès au code source d'une telle application sécurisée afin de vérifier qu'elle réalise bien ce pour quoi elle a été conçue et qu'elle ne contient pas de portions malicieuses, alors même que cette application doit s'exécuter dans l'environnement sécurisé pour cacher des informations sensibles, telles des clés de chiffrement, à l'utilisateur. Dans cet article nous présentons un mécanisme qui permet de résoudre ce problème en calculant l'empreinte caractéristique d'une application sécurisée. Cette empreinte est calculée sur les données déchiffrées de l'application sécurisée pour que l'utilisateur puisse la comparer avec l'empreinte de l'application recompilée depuis le code source, sans connaître les clés de chiffrement utilisées pour protéger la confidentialité de l'application sécurisée. Nous décrivons également comment réaliser sans microcode ce mécanisme dans l'architecture sécurisée CRYPTOPAGE en délégant au système d'exploitation, pas nécessairement digne de confiance. Ce mécanisme peut permettre de rassurer les utilisateurs qui doivent exécuter des programmes sécurisés complètement opaques sur leurs processeurs sécurisés et, ainsi, peut être faciliter l'adoption des architectures informatiques sécurisées. Cela résout aussi de manière élégante la confiance dans les architectures distribuées à grande échelle et peut permettre de faire émerger de nouvelles applications telles que les grilles de calcul de confiance ou même des DRM *open source*.

Mots-clés : Informatique de confiance, processus sécurisé, sécurité matérielle, identification de programme, empreinte de programme

1. Introduction

De nombreuses applications informatiques nécessitent un certain niveau de sécurité qui est hors de portée des architectures actuelles. Bien sûr, de nombreux algorithmes cryptographiques, des protocoles, des applications et des systèmes d'exploitations sécurisés existent, mais ils reposent tous sur une hypothèse forte : le matériel sous-jacent doit lui-même être sécurisé. Or, cette hypothèse critique n'est jamais vérifiée, excepté pour de petites applications pouvant loger sur des cartes à puce par exemple.
Durant ces dernières années, plusieurs architectures (comme XOM [11], AEGIS [16] et CRYPTOPAGE [6, 3, 1]) ont été proposées pour fournir aux applications un environnement d'exécution sécurisé. Ces architectures utilisent des mécanismes de chiffrement et de protection mémoire pour empêcher un attaquant de perturber le bon fonctionnement d'un processus sécurisé (*intégrité*), ou l'empêcher d'obtenir des informations sur le code ou les données de celui-ci (*confidentialité*). Elles essaient de prévenir des

attaques physiques contre les composants de l'ordinateur (par exemple, la X-BOX, la console de jeu de Microsoft, a été attaquée dans [5] par l'analyse des données transitant sur le bus de son processeur) ou des attaques logiques (comme par exemple un administrateur malveillant qui essaierait de voler ou de modifier le code ou les données d'un processus).
De telles architectures sécurisées peuvent être utiles dans de nombreux domaines comme par exemple le calcul distribué. Actuellement, des entreprises ou des centres de recherche peuvent hésiter à utiliser la puissance de calcul fournie par des ordinateurs d'une tierce partie car ils ne les contrôlent pas. En pratique, les propriétaires de ces nœuds peuvent voler ou modifier les algorithmes ou les résultats de l'application distribuée. En revanche, si chaque nœud de la grille utilisait un processeur sécurisé qui garantit l'intégrité et la confidentialité de l'application et de ses résultats, ce problème de sécurité disparaîtrait.
Le modèle de sécurité de ces architectures suppose que l'attaquant a un contrôle total sur tout ce qui est situé à l'extérieur du processeur (qui est supposé inaltérable et inattaquable), ce qui impose que tout ce qui est à l'extérieur du processeur doit être chiffré.
Cependant, le propriétaire d'un ordinateur peut être réticent à l'idée d'exécuter des programmes complètement opaques (du fait du chiffrement) sur sa machine, car il n'a aucun moyen de contrôle pour vérifier que les programmes font bien ce qu'ils sont censés faire et qu'ils ne contiennent pas de portions malicieuses.
La solution évidente serait de faire fonctionner ces programmes en clair, mais dans ce cas, ils seraient attaquables facilement ou tout au moins facilement espionnables. On pourrait, par exemple, voir des clés d'authentification et des algorithmes de chiffrement fonctionner pas à pas. On ne peut donc pas faire fonctionner ces programmes en clair.
Afin de réduire ce problème, nous proposons un mécanisme permettant à l'utilisateur d'obtenir l'empreinte d'un processus sécurisé, sans avoir accès aux clés de chiffrement utilisées pour protéger la confidentialité de ce processus. Grâce à cette empreinte, l'utilisateur peut vérifier qu'elle correspond à un code source donné ou qu'elle est présente dans une liste d'applications autorisées produite par un tiers de confiance. Ce mécanisme peut permettre à l'utilisateur d'avoir confiance dans les applications sécurisées qu'il exécute tout en garantissant toujours la confidentialité de ces applications.
La suite de cet article se décompose comme suit : la section 2 décrit le contexte et la nécessité d'un mécanisme d'identification de programme ; la section 3 présente le fonctionnement d'un tel mécanisme ; la section 4 décrit son implémentation sur l'architecture sécurisée CRYPTOPAGE et la section 5 présente une estimation des performances.

2. Le contexte et les besoins

Dans cette section, nous présentons quelques applications qui peuvent bénéficier des architectures informatiques sécurisées ainsi que la nécessité pour ces dernières de proposer un mécanisme d'identification de programme.

2.1. Calcul distribué sécurisé

Un domaine en plein développement, où les architectures informatiques sécurisées pourrait être utiles, est le domaine du calcul distribué sur grille de calcul.
Les entités qui soumettent des applications sur une grille n'ont pas, en général, le contrôle physique de tous les nœuds de cette grille. Le propriétaire de l'un de ces nœuds peut très bien, en montant une attaque matérielle ou logicielle, récupérer le code ou les données de l'application qui s'exécute sur son nœud ou même la perturber afin qu'elle produise des résultats erronés, voire garder des résultats positifs pour lui et répondre à la place que rien d'intéressant n'a été trouvé. Ce manque de confidentialité et d'intégrité peut poser des problèmes de propriété intellectuelle et de performance (calculs redondants pour détecter les fraudes par exemple) et donc freiner l'utilisation des grilles de calcul.
Si tous les nœuds de la grille sont équipés d'un processeur sécurisé, il devient possible de protéger l'intégrité et la confidentialité des applications qui s'exécutent sur la grille contre des attaquants et donc de faire du calcul distribué réellement sécurisé.
Quand un utilisateur veut exécuter à distance une application sur des processeurs donnés, il construit un exécutable chiffré avec les clés publiques de ces processeurs de telle façon que seuls ces processeurs

(chacun disposant d'une paire de clés asymétriques qui lui est propre) puissent déchiffrer l'application et l'exécuter.
De cette façon, on peut garantir à l'utilisateur l'exécution correcte et opaque d'un programme. Cependant, le propriétaire d'un nœud peut vouloir connaître les applications qui s'y exécutent et peut ne pas vouloir octroyer ses ressources à n'importe qui pour n'importe quel usage.
Pour résoudre ce problème, on peut fournir le code source de l'application au propriétaire. Cependant, s'il compile l'application et la chiffre, il aura accès aux clés secrètes qui la protègent et donc, il sera en mesure d'accéder et de modifier toutes les données de l'application.
Pour empêcher le propriétaire d'un nœud distant de remplacer l'application requise par une autre version malicieuse, l'utilisateur qui a soumis l'application peut utiliser le mécanisme d'attestation, présent dans de nombreuses architectures sécurisées, afin de vérifier s'il s'agit bien de l'application originale qu'il a lui-même compilée ou au contraire une autre application.
Cependant, dans ce cas, le propriétaire n'a aucun moyen de vérifier que l'application chiffrée qu'il exécute correspond bien au code source qui lui a été fourni. Pour résoudre ce problème, nous proposons un mécanisme permettant de calculer une empreinte sur un processus sécurisé. Le processeur va calculer cette empreinte sur le code déchiffré, sur les données initiales déchiffrées et sur le contexte matériel initial du processus sécurisé. Ainsi, cette empreinte ne dépend pas des clés utilisées pour protéger l'application et donc, si le propriétaire recompile l'application à partir des sources (en admettant qu'il dispose du même environnement de compilation que celui utilisé initialement), il peut calculer également l'empreinte de l'application qu'il vient de recompiler et comparer les deux empreintes. Si elles sont identiques, le propriétaire peut être sûr que le code source et l'application sécurisée correspondent bien, sans avoir accès aux clés de celle-ci et percer tous ses secrets.

2.2. Logiciels libres de gestion de droits de contenus numériques

Les systèmes de gestion des droits sur les œuvres numériques (*Digital Rights Management*, DRM) se sont répandus ces dernières années afin d'obliger les utilisateurs à respecter les licences d'utilisation (par exemple, interdiction de réaliser des copies, nombre de visionnages limité, durée de vie limitée, etc.) des œuvres numériques (musiques, films, livres électroniques, etc.) qu'ils ont acquises.
Certains de ces logiciels de DRM s'exécutent sur des ordinateurs classiques sans dispositifs matériels particuliers. Ils sont donc vulnérables à des attaques logicielles ou matérielles. Par exemple, un attaquant peut analyser le code du programme afin d'extraire les algorithmes et les secrets utilisés pour protéger les œuvres. Le code de ces applications est donc fermé mais aussi camouflé afin de tenter d'empêcher ce type d'attaques. Du coup, il est très difficile de connaître ce que font réellement ces applications et savoir si, par exemple, elles ne divulguent pas d'informations personnelles.
Les architectures sécurisées peuvent garantir que personne ne peut espionner une application de DRM et donc peuvent augmenter leur niveau de sécurité mais, d'un autre côté, les utilisateurs ont encore moins de possibilités de vérifier ce que ce processus opaques font, s'ils manipulent correctement leurs informations personnelles ou s'ils n'endommagent pas l'ordinateur.
Une solution consiste à fournir à l'utilisateur le code source de l'application de DRM pour qu'il puisse la vérifier. Mais dans ce cas, l'utilisateur doit avoir un moyen de vérifier que le fichier exécutable chiffré contenant l'application correspond bien au code source qui lui a été fourni. De plus, le code source ne doit évidemment pas contenir de secrets et les fournisseurs de contenus doivent pouvoir vérifier que l'utilisateur n'a pas modifié ce code source dans le but de modifier le comportement de l'application.
Nous proposons la solution suivante. Le distributeur de l'application de DRM génère un exécutable chiffré à partir du code source de l'application et de la clé publique du processeur cible. Ensuite il distribue l'exécutable chiffré et le code source à l'utilisateur.
À la première exécution sur l'architecture sécurisée, l'application de DRM génère (à l'aide d'un algorithme intégré et du générateur de nombre aléatoire fourni pas la plate-forme) une paire de clés asymétriques, stocke la partie privée à l'aide de la fonction de stockage sécurisé fournie par l'architecture, demande au processeur d'attester que la clé publique a bien été générée par l'application sécurisée et distribue la clé publique ainsi que l'attestation aux fournisseurs de contenus.
Les fournisseurs de contenus peuvent ainsi vérifier que la clé publique qu'ils reçoivent a été générée par l'application de DRM qui s'exécutait sur un processeur sécurisé authentique. Grâce à cette vérification, ils peuvent être sûrs que la clé privée correspondante a été correctement générée et est correctement

protégée. À l'aide de la clé publique, ils peuvent chiffrer les œuvres numériques et être sûrs qu'elles ne pourront être déchiffrées que par l'application de DRM.
Quant à l'utilisateur, en utilisant le mécanisme d'identification de programme, il peut vérifier que l'application sécurisée et le code source fourni correspondent bien. Cependant, s'il essaie d'exécuter l'application qu'il a recompilée, les fournisseurs de contenus peuvent s'en apercevoir (car l'identifiant de l'application inclus dans l'attestation dépend de la globalité de l'application, y compris les clés qui ne sont pas connues par l'utilisateur) et donc peuvent refuser de communiquer avec elle.
Si cette action semble trop lourde à mettre en place, on peut imaginer déléguer cette vérification à des organismes tiers qui feront tourner la vérification dans leur environnement de compilation. Afin de pouvoir aussi authentifier cet environnement de vérification distant, ce sera fait dans une machine virtuelle complète qui tournera dans un processus chiffré lui-même vérifiable par tous. On constituera ainsi l'équivalent des certificats des autorités de certification dans les infrastructures à clé publique.

3. Le mécanisme d'identification de base

Dans cette section nous présentons le mécanisme d'authentification de programme et comment il peut être implémenté sur une architecture sécurisée classique.

3.1. Description

L'objectif de ce mécanisme est de permettre à un utilisateur de vérifier qu'un programme, stocké dans un exécutable chiffré et chargé en mémoire afin d'être exécuté dans un environnement d'exécution garantissant sa confidentialité, correspond à un code source donné ou à un autre programme identique présent dans une liste d'applications de confiance.
Le fichier exécutable est chiffré à l'aide d'une clé de session symétrique, elle-même chiffrée avec la clé publique du processeur sécurisé sur lequel l'application doit s'exécuter. Durant l'exécution, les instructions et les données qui sont stockées à l'extérieur du processeur (en mémoire ou sur un disque) sont chiffrés à l'aide d'une clé symétrique. On suppose que l'utilisateur n'a pas accès à la clé privée du processeur ni aux clés garantissant la confidentialité et l'intégrité du programme et de ses données durant son exécution.
L'idée principale est de calculer l'empreinte d'une application sécurisée et de la retourner à l'utilisateur. Cette empreinte doit respecter les propriétés suivantes :

- deux programmes identiques, qui ont exactement le même code, les mêmes données initiales et le même contexte matériel initial, compilés grâce à un environnement de compilation rigoureusement identique mais qui peuvent être chiffrés avec des clés différentes (par exemple pour tourner de manière étanche sur 2 processeurs différents) doivent avoir la même empreinte ;
- deux programmes différents doivent, avec une très forte probabilité, avoir des empreintes différentes ;
- il doit être impossible, en pratique, de créer un programme qui a la même empreinte qu'un autre programme ;
- il doit être impossible, en pratique, d'obtenir la moindre information à propos d'un programme à partir de son empreinte.

Une méthode simple pour garantir ces propriétés est d'utiliser une fonction de hachage résistantes aux collisions comme SHA-1 [15] pour calculer cette empreinte.
Afin d'identifier un programme spécifique, l'empreinte doit inclure le contexte matériel initial, le code initial et les données initiales du processus sécurisé. De plus, pour permettre à l'utilisateur de comparer le processus sécurisé avec un code source alors qu'il ne possède pas les clés secrètes utilisées pour protéger le processus, l'empreinte doit être calculée sur des informations déchiffrées et donc, elle doit être calculée directement par le processeur.
L'adresse virtuelle des différentes pages mémoires utilisées initialement par le processus doit également être incluse dans le calcul de l'empreinte afin d'empêcher qu'un programme dont les adresses des pages sont différentes (permutation du contenu des pages) d'avoir la même empreinte.
L'empreinte doit être calculée avant le début de l'exécution du processus sécurisé car, après, elle pourrait contenir des informations calculées par le processus et qui doivent rester secrètes, ce qui pourrait permettre à un attaquant d'essayer de les récupérer. Mais surtout, cela perdrait de son sens puisque ce qui nous intéresse ici est l'authentification de l'état initial de processus, pas de leur futur.

Indentify(p)
  $\mathtt{ident}_{\mathtt{address}} = \bot$
  $\mathtt{ident} = \mathrm{H}(\mathrm{ContexteDéchiffréÉpuré(p)})$
  pour page **dans** Pages(p)
    pour (donnée,adresse) **dans** Données(page)
      $\mathtt{ident} = \mathrm{H}(\mathtt{ident} \| \mathtt{donnée} \| \mathtt{adresse})$
      $\mathtt{ident}_{\mathtt{address}} = \mathrm{H}(\mathtt{ident}_{\mathtt{address}}, \mathtt{adresse})$
  retourne $\mathtt{sign\_result}(\mathtt{ident} \| (\mathtt{ident}_{\mathtt{address}} \stackrel{?}{=} \mathtt{ident}_{\mathtt{addressInit}}(\mathtt{p})))$

FIG. 1 – Principe de l'identification d'un programme p.

Enfin, après le calcul de l'empreinte, le processeur doit la signer numériquement afin d'attester de son authenticité et la retourner au système d'exploitation qui, par la suite, la retournera à l'utilisateur.
Le principe de cette authentification de programme est résumée sur la figure 1. On a rajouté une identification d'adresse qui va servir dans la section suivante, mais tous ce qui concerne $\mathtt{ident}_{\mathtt{address}}$ est à ignorer pour l'instant.

3.2. Délégation à un système d'exploitation non digne de confiance

Le système d'exploitation peut participer à ces opérations afin d'améliorer la flexibilité du mécanisme et ainsi réduire les modifications matérielles nécessaires. Cependant, dans la majorité des architectures sécurisées proposées, le système d'exploitation n'est pas digne de confiance et donc, la répartition des tâches doit être réalisée de façon à ce qu'il ne puisse pas casser les propriétés de sécurité.
Le déchiffrement du code et des données du processus sécurisé ainsi que le calcul du résumé cryptographique sur ces informations déchiffrées doivent être réalisés par le processeur car le système d'exploitation ne doit pas avoir accès à ces informations confidentielles ni pouvoir les falsifier.
Afin de permettre la délégation de certaines opérations, le processeur doit fournir les instructions suivantes :
– une instruction pour initialiser le calcul de l'empreinte ;
– une instruction pour charger une page ou une ligne depuis la mémoire vers le processeur, la déchiffrer et vérifier son intégrité (ces opérations étant effectuées par le processeur) ;
– une instruction pour mettre à jour le résumé cryptographique avec l'adresse et le contenu de la ligne ou de la page préalablement chargée (le système d'exploitation ne doit pas avoir accès à ce contenu déchiffré) ;
– une instruction pour terminer le calcul du résumé, l'attester et le renvoyer.
Pendant les calculs, le processeur doit disposer de mécanismes lui permettant de détecter si le système d'exploitation essaie de corrompre le calcul de l'empreinte. Premièrement, le processeur doit vérifier que toutes les lignes ou les pages chargées par le système d'exploitation appartiennent bien au bon processus sécurisé. Cette vérification peut être effectuée facilement car l'intégrité des données chargées est vérifiée à l'aide du mécanisme de protection d'intégrité de l'architecture sécurisée.
De plus, le processeur doit s'assurer que l'exécution du processus n'a pas commencé avant la fin du calcul de l'empreinte car sinon, les données étant modifiées par l'exécution, cela modifierait l'empreinte. Cette vérification peut être implémentée en vérifiant, à chaque étape, la valeur d'un drapeau qui est positionné quand le processus sécurisé débute l'exécution.
Enfin, pour être sûr que le système d'exploitation a bien parcouru toutes les adresses définies dans le programme ou au moins choisies par le concepteur comme devant être incluse dans l'authentification on rajoute un hachage simultané des adresses, le $\mathtt{ident}_{\mathtt{address}}$ de la figure 1. À la fin de l'algorithme, $\mathtt{ident}_{\mathtt{address}}$ contient un haché qui dépend de toutes les adresses parcourue et de l'ordre du parcours. Si le concepteur du contexte chiffré d'origine du programme p a mis dans le champ $\mathtt{ident}_{\mathtt{addressInit}}$ du contexte chiffré ce même haché, une simple comparaison à la fin permet à quiconque de vérifier si le système d'exploitation a bien travaillé en séquençant les bonnes instructions dans le bon ordre.
En fonction de l'usage fait de cette empreinte, son calcul peut ne pas être protégé contre la corruption du système d'exploitation. En effet, si le but est de protéger le système d'exploitation lui-même contre l'exé-

cution de programmes malicieux, le système d'exploitation n'a pas intérêt à faire de fausses empreintes qui empêcherait cette vérification.

3.3. Utilisation

Quand l'utilisateur veut comparer une application sécurisée avec un code source, il demande au système d'exploitation qui aide le processeur à calculer l'empreinte de l'application en question.
L'utilisateur doit disposer du code source de l'application et d'un environnement de compilation strictement identique (compilateur, éditeur de liens, bibliothèques, heure de compilation qui est incluse dans les binaires, etc.) que celui utilisé lors de la création de l'application sécurisée. Avec ces outils, il recompile l'application puis, en appliquant le même algorithme que celui utilisé par le processeur, il calcule l'empreinte de l'application recompilée.
Ensuite, l'utilisateur récupère l'empreinte de l'application sécurisée et peut la comparer avec celle de l'application recompilée. Si elles sont identiques, les applications sont identiques (à moins d'une collision dans l'algorithme de hachage utilisé, ce qui est extrêmement rare).
Au cas où le distributeur de l'application sécurisée ne veut pas distribuer le code source, l'utilisateur peut toujours comparer l'empreinte de l'application avec une liste d'empreintes d'applications approuvées par un tiers de confiance ou par le distributeur du programme. Dans ce cas, l'utilisateur ne peut pas vérifier lui-même que l'application est correctement programmée mais il peut savoir que c'est bien la bonne application qu'il exécute et pas un programme malicieux.

4. Implémentation sur l'architecture CryptoPage

Dans cette section, nous présentons l'implémentation du mécanisme d'identification de programme sur l'architecture sécurisée CRYPTOPAGE.

4.1. Rappels sur l'architecture CRYPTOPAGE

L'architecture sécurisée CRYPTOPAGE a été introduite en 2000 [6] et a été améliorée dans [9, 2, 4, 3, 1] (vérificateur mémoire résistant aux attaques par rejeu, nouveaux mécanismes de chiffrement et d'intégrité, etc.).
La figure 2 présente cette architecture. Les parties en gris représentent les unités présentent sur un microprocesseur classique. Même si l'on voit qu'il y a beaucoup plus de blanc que de grisé, il ne faut pas perdre de vue que les parties en gris sont, pour des processeurs généralistes, bien plus complexes que l'ensemble de toutes les parties rajoutées pour CRYPTOPAGE.

4.1.1. Objectifs

L'architecture CRYPTOPAGE doit garantir aux processus sécurisés les deux propriétés suivantes :
– *confidentialité* : un attaquant doit pouvoir obtenir le moins d'information possible sur le code ou les données manipulées par un processus sécurisé ;
– *intégrité* : l'exécution correcte d'un processus sécurisé ne doit pas pouvoir être altérée par une attaque. En cas d'attaque, le processeur doit interrompre l'exécution du processus.

Le processeur offre deux nouveaux environnement d'exécution en plus de l'environnement normal non sécurisé. Le premier offre la protection de la confidentialité et de l'intégrité des processus tandis que le second offre uniquement la protection de l'intégrité.
Le processeur CRYPTOPAGE est capable d'exécuter des processus sécurisés et non sécurisés à l'aide d'un système d'exploitation adapté à l'architecture mais qui n'est pas nécessairement digne de confiance.
Tout ce qui est à l'extérieur de la puce du processeur (par exemple la mémoire, les unités de stockage de masse, etc.) est considéré comme étant sous le contrôle total d'un attaquant. Par exemple, ce dernier peut modifier le contenu de la mémoire, corrompre le système d'exploitation afin de lui faire espionner le processeur, exécuter les processus sécurisés et non sécurisés de son choix, etc. Cependant, l'attaquant n'a pas accès, directement ou indirectement, au processeur lui-même.
En particulier, les attaques temporelles [7], les attaques par analyse de la consommation (DPA [8]), etc. sont considérées déjà résolues par ailleurs. Des techniques de compilation adaptées peuvent déjà limiter un certain nombre de ces attaques.

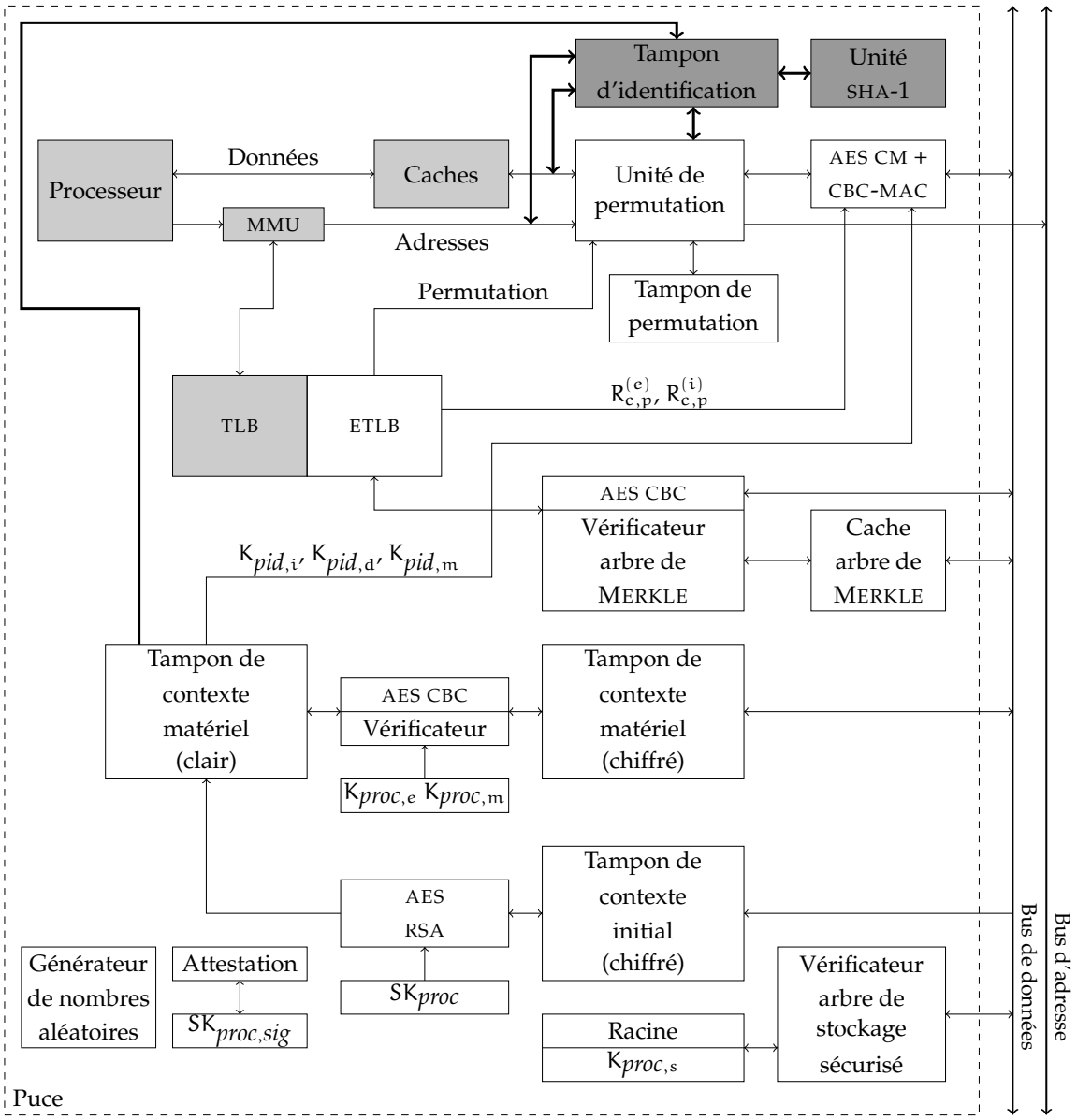

FIG. 2 – Schéma de l'architecture CRYPTOPAGE. En blanc ce qui est rajouté à un processeur standard en gris clair. En gris foncé ce qui est rajouté à un CRYPTOPAGE standard.

Enfin, les attaques par déni de service ne sont pas non plus considérées car inévitables : si l'utilisateur coupe l'alimentation ou que le système d'exploitation ne remplit pas son rôle, une application, même sécurisée ne pourra pas s'exécuter. Si l'intégrité du processus est visée, les mécanismes de CRYPTOPAGE stopperont l'exécution. Si le système d'exploitation et les composants périphériques ne réalisent pas honnêtement leurs tâches sans modifier l'intégrité du processus, CRYPTOPAGE ne détectera pas cette rupture de contrat. Pour résister à ces attaques, il convient au programmeur de vérifier avec des protocoles spécifiques que l'environnement remplit bien son contrat. Par exemple en gardant dans l'environnement mémoire sécurisée le hachage d'un fichier dont l'écriture a été demandée au système d'exploitation. Lors d'une relecture ultérieure, le processus revérifiera le hachage des données et en cas d'incohérence aura détecté que l'environnement ne joue pas le jeu.

4.1.2. Description

L'architecture CRYPTOPAGE est basée sur un processeur standard auquel sont ajoutés des mécanismes de chiffrement mémoire, de protection de l'intégrité de la mémoire et des protections contre les fuites d'information sur le bus d'adresse [1].

La protection contre les fuites d'informations sur le bus d'adresse est implémentée en utilisant l'infrastructure HIDE introduite par ZHUANG ET AL. dans [17]. Cette infrastructure utilise un mécanisme de verrouillage des lignes de cache et des permutations régulières des lignes au sein des pages mémoire afin de masquer les motifs d'accès en mémoire. Grâce à cette infrastructure, un attaquant qui observerait le bus d'adresse ne pourrait pas apprendre qu'une ligne est plus utilisée qu'une autre car, entre deux permutations d'une page, toutes les lignes de cette page sont écrites et lues une seule fois en mémoire et rechiffrées. Les tables de permutation, qui contiennent les adresses de chaque ligne au sein d'une page, sont stockées, pour chaque page mémoire, avec les descripteurs de cette page et chargées dans une extension des *Translation Lookaside Buffers* (ETLB) classiques du processeur afin d'être disponibles.

Le chiffrement des données en mémoire est effectué à l'aide d'un algorithme de chiffrement par bloc (comme par exemple AES [13]) utilisé dans le mode compteur [14] et de clés symétriques propres à chaque processus sécurisé. Le compteur utilisé pour le chiffrement d'une ligne donnée est la concaténation de l'adresse virtuelle de la ligne et d'un nombre aléatoire choisi pour chaque page lors de la permutation de la page en question. Comme l'adresse d'une ligne et le nombre aléatoire (qui est également stocké dans un ETLB) sont disponibles avant chaque accès mémoire à une ligne, le calcul des masques utilisés pour le déchiffrement dans le mode compteur peut être effectué en parallèle avec l'accès mémoire, ce qui pénalise très peu les performances.

L'intégrité des données en mémoire est garantie par la combinaison de deux mécanismes. Pour chaque ligne stockée en mémoire, un *Message Authentication Code* (MAC) est calculé avec les données de la ligne, l'adresse virtuelle de celle-ci, un nombre aléatoire également propre à chaque page et choisi lors de chaque permutation et une clé secrète propre au processus. Ces MAC garantissent l'intégrité des données en mémoire contre toutes altérations, à l'exception des attaques par rejeu. Afin de se prémunir contre ces attaques, les nombres aléatoires utilisés (un par page mémoire) sont protégés en utilisant un arbre de hachage (ou arbre de MERKLE [12]) couvrant les descripteurs étendus des pages mémoires (et donc des nombres aléatoires qui y sont stockés).

Lors d'une interruption, le contexte matériel du processus sécurisé qui était en cours d'exécution est copié dans l'un des tampons de contexte matériel disponible dans le processeur. Ces tampons sont uniquement accessibles par le processeur afin d'empêcher les fuites d'informations. Cependant, le système d'exploitation peut demander au processeur de chiffrer le contenu d'un tampon de contexte afin de pouvoir le copier en mémoire et libérer ainsi un tampon de contexte pour un autre processus sécurisé. Il peut également, après avoir chargé un tampon à l'aide d'un contexte matériel chiffré, demander au processeur de déchiffrer ce contexte afin qu'il puisse être exécuté.

4.2. Implémentation

Afin d'implémenter le mécanisme d'identification de programme sur l'architecture CRYPTOPAGE, un nouveau tampon spécialisé (appelé tampon d'identification), de même taille qu'une page mémoire est ajouté, ainsi qu'une unité implémentant une fonction de hachage cryptographique (par exemple SHA-1), que plusieurs instructions spécialisées et quatre registres spéciaux :

- SPIDA qui stocke l'adresse de la page mémoire actuellement stockée dans le tampon d'identification ;
- SPIDB qui contient le numéro du tampon de contexte stockant le contexte matériel du processus sécurisé qui est en cours d'identification ;
- SPIDC qui contient un compteur indiquant la ligne du tampon d'identification à inclure dans le résumé cryptographique. L'adresse utilisée dans la figure1 est donc la concaténation de SPIDA et SPIDC ;
- SPIDH qui contient l'empreinte du processus.

Le calcul de l'empreinte s'effectue selon la procédure suivante décrite aussi sur la figure 3 :

1. Le système d'exploitation exécute l'instruction `spid_init`. Cette instruction prend en paramètre le numéro du tampon de contexte contenant le contexte matériel initial du processus à identifier. Elle vérifie tout d'abord que l'exécution de ce processus n'a pas encore commencé, elle copie le numéro du tampon de contexte dans le registre spécial SPIDB et efface le contenu des registres SPIDA et SPIDH.

Indentify(n)
  spid_init(n) $\{ \mathrm{ident}_{\mathrm{address}} = \bot \}$
  spid_hash_ctx $\{ \mathrm{ident} = \mathrm{H}(\mathrm{ContexteDéchiffréÉpuré}(\mathrm{p}(\mathrm{n}))) \}$
  pour page **dans** Pages(p)
    spid_init_page(page)
    pour ligne **dans** Lignes(page) { *Remplit le tampon d'identification de la page* }
      spid_load_line(ligne)
    pour ligne **dans** Lignes(page)
      spid_hash_line $\{ \mathrm{ident} = \mathrm{H}(\mathrm{ident} \| \mathrm{donnée} \| \mathrm{adresse}) \; \mathrm{ident}_{\mathrm{address}} = \mathrm{H}(\mathrm{ident}_{\mathrm{address}}, \mathrm{adresse}) \}$
    spid_hash_page
  retourne sign_result$(\mathrm{ident} \| (\mathrm{ident}_{\mathrm{address}} \stackrel{?}{=} \mathrm{ident}_{\mathrm{addressInit}}(\mathrm{p}(\mathrm{n}))))$

FIG. 3 – Principe de l'identification d'un programme décrit par un descripteur de contexte matériel chiffré n avec en commentaire l'effet résumé des instructions.

2. Le système d'exploitation exécute ensuite l'instruction `spid_hash_ctx`. Cette instruction va calculer le résumé cryptographique du contexte matériel du processus et le stocker dans le registre SPIDH. Certaines informations ne sont pas prises en compte dans le calcul de ce résumé, comme les clés de chiffrement du programme, la racine de l'arbre de hachage protégeant les informations sur les pages mémoire, et les registres spéciaux.
3. Ensuite, pour chaque page mémoire contenant des données ou des instructions du programme :
   (a) Le système d'exploitation exécute l'instruction `spid_init_page` en lui passant en paramètre l'adresse logique de la page mémoire concernée. Cette instruction stocke cette adresse dans le registre SPIDA, invalide toutes les lignes du tampon d'identification, remet à zéro le compteur SPIDC ainsi que l'unité de hachage dédiée. Enfin, elle envoie la valeur contenue actuellement dans SPIDH et l'adresse de la page à l'unité de hachage dédiée pour commencer le calcul du nouveau résumé.
   (b) Pour chaque ligne l de la page (numérotées de 0 à $N-1$), le système d'exploitation demande le chargement de la ligne l à l'aide de l'instruction `spid_load_line`. Le processeur charge alors la ligne située à l'adresse l dans la page vers la ligne l du tampon d'identification et positionne un drapeau pour indiquer que la ligne est présente. l représente ici le numéro de ligne permuté au sens de HIDE, et donc des lignes contiguës en mémoire ce qui fait que la lecture se fait dans l'ordre croissant et donc il n'y a pas de fuite d'information sur la permutation initiale.
   (c) Une fois toutes les lignes de la page sont prétendument lues dans le tampon d'identification, pour chaque ligne de la page, le système d'exploitation exécute l'instruction `spid_hash_line`. Cette instruction vérifie tout d'abord si la ligne $P(l)$ (où l est la valeur contenue dans le compteur SPIDC et P est la permutation actuelle de la page mémoire concernée) contient des données ou des valeurs d'authentification. Dans ce dernier cas, elle ne fait rien d'autre. Si la ligne contient des données, elle les déchiffre, les vérifie et les envoie ensuite à l'unité de hachage pour mettre à jour le résumé cryptographique de la page en cours de calcul. Enfin, cette instruction incrémente le compteur SPIDC. Au passage, l'instruction met à jour un drapeau indiquant le cas échéant qu'une ligne demandée manque à l'appel ou est mal authentifiée : dans ce cas le système d'exploitation a fait un déni de service lors de la phase précédente ou il y a eu une attaque sur la mémoire. Mais on continue de même pour ne pas indiquer à l'attaquant l'endroit équivalent dans la permutation qui permettrait pas à pas de reconstruire la permutation initiale. Comme le hachage se fait dans l'ordre logique des adresses non mélangées au sens de HIDE à l'intérieur du tampon d'identification de page, un attaquant ne peut pas voir la permutation.
   (d) Le système d'exploitation exécute enfin l'instruction `spid_hash_page` qui vérifie que le compteur SPIDC contient bien la bonne valeur (c'est-à-dire le nombre de lignes dans une

page). Une exception est levée au cas où le drapeau signalant qu'une ligne maquait ou était invalide. Enfin, si tout est normal, demande à l'unité de hachage de terminer le calcul du résumé cryptographique et qui stocke la valeur renvoyée par cette unité dans le registre SPIDH.

4. Le calcul du résumé cryptographique est terminé, et le système d'exploitation peut le lire à partir du registre SPIDH et le renvoyer à l'utilisateur.

On constate que le système est capable de faire des identifications partielles si besoin est à condition que cela soit fait sur des pages entières. Par contre la vérification finale par rapport à $\mathtt{ident}_{\mathtt{addressInit}}(\mathtt{p}(\mathtt{n}))$ permet de s'assurer que si on a demandée une vérification complète elle est bien complète.
Un hachage des adresses parcourues permet de vérifier dans le contexte le hachage précalculé correspondant pour savoir si les adresses ont bien été toutes parcourues et dans le bon ordre.
Le contexte contient déjà un hachage si on fait confiance à l'emballeur du programme. Le recalcul permet de vérifier sa probité.

4.3. Sécurité

Tout d'abord, si l'utilisateur utilise la même méthode pour calculer le résumé cryptographique du binaire qu'il a recompilé à partir des sources (pourvu qu'il ait exactement le même environnement de compilation que celui utilisé pour compiler le binaire chiffré), si le système d'exploitation a bien suivi la procédure ci-dessus, et que le programme chiffré et le programme en clair recompilé par l'utilisateur sont identiques, alors les deux résumés cryptographiques seront bien identiques. En effet, le résumé cryptographique calculé par le processeur sur le processus sécurisé ne dépend pas de données chiffrées ou d'informations choisies lors du chiffrement (il intègre des adresses logiques, des données en clair et le contexte matériel initial dont on a exclu les informations choisies lors du chiffrement du programme).
Si la même ligne du programme en clair et du programme chiffré contient des valeurs différentes, les deux résumés cryptographiques ainsi calculés seront différents (sauf dans le cas exceptionnel d'une collision dans la fonction de hachage). Il en est de même si une page est stockée à une adresse logique différente de celle prévue, puisque l'adresse des pages est incluse dans le calcul.
Les résumés seront en revanche identiques si le programme chiffré contient des pages (de code ou de données) additionnelles par rapport au programme en clair et que le système d'exploitation a délibérément omis d'inclure ces pages dans la procédure de calcul du résumé cryptographique. *A priori* ces pages supplémentaires n'auront pas d'impact sur l'exécution du processus, sauf erreur de programmation (débordement de tampon par exemple) qui serait détectable lors l'analyse du code source par un utilisateur averti. De plus, le système d'exploitation ne peut pas, après le calcul de ce résumé, permuter deux pages d'un processus car les adresses logiques sont utilisées dans le calcul des valeurs d'authentification des lignes de cache.

5. Évaluation et Performances

Dans cette section nous allons essayer d'évaluer les performances de ce mécanisme.
Le premier point à mentionner est que ce mécanisme peut être optionnel. L'utilisateur peut par exemple n'avoir besoin de calculer l'empreinte d'un programme sécurisé que la première fois qu'il l'exécute si l'exécutable est stockée de façon sécurisée et que l'utilisateur fait confiance au système d'exploitation.
Ce mécanisme n'ajoute qu'une latence au démarrage de l'application. Une fois l'empreinte calculée, l'exécution du processus sécurisé n'est pas modifiée. On peut donc imaginer ce calcul effectué lors de l'installation du logiciel.
Plusieurs aspects sont à prendre en compte pour estimer le temps nécessaire au calcul de l'empreinte d'un processus sécurisé :

- le temps nécessaire pour charger le code et les données depuis le fichier exécutable sur le disque vers la mémoire qui dépend principalement de la vitesse du disque (les systèmes d'exploitation actuels chargent uniquement les pages du disque vers la mémoire lors de la première utilisation) ;
- le temps nécessaire pour charger toutes les lignes depuis la mémoire vers le tampon d'identification, qui dépend principalement de la vitesse de la mémoire ;
- le temps nécessaire pour déchiffrer et vérifier toutes les lignes chargées dans le tampon d'identification, qui dépend de la latence de déchiffrement et de vérification ;

– le temps nécessaire pour calculer le résumé cryptographique de toutes les pages du processus, qui dépend de la vitesse de l'unité de hachage.

Le pire cas est atteint lorsque toutes ces opérations sont effectuées séquentiellement.

Le débit des disques durs actuels est compris entre 50 et 100 Mo/s, pour la mémoire, il est compris entre 5 et 15 Mo/s. Dans l'architecture CRYPTOPAGE le déchiffrement d'une ligne nécessite uniquement un cycle d'horloge (0,5 ns à 2 GHz) et la vérification nécessite 15 cycles (7,5 ns). Dans [10], LEE, CHAN et VERBAUWHEDE présentent une implémentation de SHA-1 capable d'atteindre des débits entre 3 et 10 Gb/s (0.375 Go/s et 1.25 Go/s) en technologie ASIC 0.18 µm.

À l'aide de ces chiffres, on obtient un temps maximal de 22,8 ms pour calculer l'empreinte d'un programme de 1 Mo dans le pire des cas. Pour un programme de 100 Mo, le calcul de l'empreinte prendrait un peu plus de deux secondes.

Le goulot d'étranglement se situe au niveau du disque dur (l'empreinte doit être calculée sur le programme entier qui doit donc être lu complètement depuis le disque) et, dans une moindre mesure, de l'unité de hachage.

6. Conclusions

Le rajout de modes d'exécution sécurisés devrait permettre d'augmenter la sécurité des applications en évitant des attaques logicielles ou matérielles. Cet ajout permettra aussi de construire des architectures sécurisées distribuées à grande échelle en assurant une confiance mutuelle entre les nœuds.

Mais une conséquence de ces architectures sécurisées est que l'on ne peut plus savoir ce qui est réellement exécuté puisque les programmes et les données sont chiffrées. En particulier, même un logiciel libre que l'on veut exécuter à distance devient opaque et l'utilisateur distant, qui prête des ressources, ne peut plus vérifier que le programme est bien celui qu'il prétend être.

Pour résoudre ce problème, nous avons décrit dans cet article un mécanisme efficace qui calcule une empreinte cryptographique d'un programme sécurisé dans le cas de l'architecture sécurisée CRYPTOPAGE mais qui peut être adapté à d'autre architectures de ce type. Le système d'exploitation n'a pas besoin d'être de confiance mais grâce à un mécanisme de délégation sécurisée, il cadence la vérification et un mécanisme du processeur permet de vérifier que le travail a correctement été effectué.

Le temps de calcul de l'empreinte d'une application est petit comparé au temps d'accès du programme du disque dur. Cette vérification est lancée à la demande par le système d'exploitation avant exécution et n'impacte donc pas la vitesse d'exécution. De plus, le système d'exploitation peut très bien avoir un cache de vérification indexé par l'identifiant de l'exécutable pour n'avoir à faire cette vérification qu'une seule fois quel que soit le nombre d'exécutions.

Avec ces mécanismes, qui déplacent la confiance au niveau matériel, on peut envisager de nouvelles infrastructures distribuées et tolérantes aux pannes pour du calcul, des dossiers médicaux gérant l'accès fin aux données en respectant la vie privée ou pourquoi pas des DRM dont les sources seraient ouverts ou des systèmes de vote électronique. Le fait d'avoir décrit ici un moyen de vérifier ce qui tourne sur son ordinateur, même en mode sécurisé, peut diminuer les réticences envers ces nouvelles technologies.

Cet aspect grille de calcul sécurisé avec l'aide de tels processeurs est actuellement étudié dans le cadre du projet ANR SAFESCALE.

7. Remerciements

Ce travail est soutenu par une bourse de thèse de la Délégation Générale pour l'Armement (DGA), par le projet ANR ARA/SSIA BGPR SAFESCALE et par le projet de l'Institut TÉLÉCOM TCP (Trusted Computing Platform). Merci aux collègues de TÉLÉCOM ParisTech de TCP (en particulier Sylvain GUILLEY et Renaud PACALET) et du projet SAFESCALE pour leur nombreuses interactions, (particulièrement Sébastien VARETTE, Jean-Louis ROCH et Christophe CÉRIN), à Jacques STERN et aux membres de l'équipe de cryptographie de l'ENS pour les discussions sur le projet. Merci à Loïc PLASSART pour la relecture de cet article.

Bibliographie

1. Duc (Guillaume). – *Support matériel, logiciel et cryptographique pour une exécution sécurisée de processus*. – Thèse de PhD, École Nationale Supérieure des Télécommunications de Bretagne, 2007. `http://enstb.org/~gduc/these/these.pdf`.
2. Duc (Guillaume) et Keryell (Ronan). – Portage de l'architecture sécurisée CRYPTOPAGE sur un microprocesseur x86. *In : Symposium en Architecture nouvelles de machines (*SYMPA*'2005)*, pp. 61–72. – avril 2005.
3. Duc (Guillaume) et Keryell (Ronan). – CRYPTOPAGE : an efficient secure architecture with memory encryption, integrity and information leakage protection. *In : Proceedings of the 22th Annual Computer Security Applications Conference (*ACSAC*'06)*. pp. 483–492. – IEEE Computer Society, décembre 2006.
4. Duc (Guillaume) et Keryell (Ronan). – CRYPTOPAGE/HIDE : une architecture efficace combinant chiffrement, intégrité mémoire et protection contre les fuites d'informations. *In : Symposium en Architecture de Machines (*SYMPA*'2006)*. – octobre 2006.
5. Huang (A.). – *Keeping Secrets in Hardware : the Microsoft XBox (*TM*) Case Study*. – Rapport technique nAI Memo 2002-008, Massachusetts Institute of Technology, mai 2002.
6. Keryell (Ronan). – CRYPTOPAGE-1 : vers la fin du piratage informatique ? *In : Symposium d'Architecture (*SYMPA*'6)*, pp. 35–44. – Besançon, juin 2000.
7. Kocher (Paul C.). – Timing attacks on implementations of DIFFIE-HELLMAN, RSA, DSS, and other systems. *In : Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (*CRYPTO*'96)*. pp. 104–113. – Springer-Verlag, août 1996.
8. Kocher (Paul C.), Jaffe (Joshua) et Jun (Benjamin). – Differential power analysis. *In : Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (*CRYPTO*'99)*. pp. 388–397. – Springer-Verlag, août 1999.
9. Lauradoux (Cédric) et Keryell (Ronan). – CRYPTOPAGE-2 : un processeur sécurisé contre le rejeu. *In : Symposium en Architecture et Adéquation Algorithme Architecture (*SYMPAAA*'2003)*, pp. 314–321. – La Colle sur Loup, France, octobre 2003.
10. Lee (Yong Ki), Chan (Herwin) et Verbauwhede (Ingrid). – Throughput optimized SHA-1 architecture using unfolding transformation. *In : Proceeding of the International Conference on Application-specific Systems, Architectures and Processors (*ASAP*'06)*, pp. 354–359. – septembre 2006.
11. Lie (David), Thekkath (Chandramohan), Mitchell (Mark), Lincoln (Patrick), Boneh (Dan), Mitchell (John) et Horowitz (Mark). – Architectural support for copy and tamper resistant software. *In : Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (*ASPLOS IX*)*, pp. 168–177. – octobre 2000.
12. Merkle (Ralph C.). – A certified digital signature. *In : Proceedings on Advanced in Cryptology (*CRYPTO*'89)*. pp. 218–238. – Springer-Verlag New York, Inc., 1989.
13. NIST. – Advanced Encryption Standard (AES), novembre 2001. Federal Information Processing Standards Publication 197.
14. NIST. – Recommendation for block cipher modes of operation. – Special Publication 800-38A, décembre 2001.
15. NIST. – Secure Hash Standard (SHS), août 2002. Federal Information Processing Standards Publication 180-2.
16. Suh (G. Edward), Clarke (Dwaine), Gassend (Blaise), van Dijk (Marten) et Devadas (Srinivas). – AEGIS : Architecture for tamper-evident and tamper-resistant processing. *In : Proceedings of the 17th International Conference on Supercomputing (*ICS*'03)*, pp. 160–171. – juin 2003.
17. Zhuang (Xiaotong), Zhang (Tao) et Pande (Santosh). – HIDE : an infrastructure for efficiently protecting information leakage on the address bus. *In : Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (*ASPLOS-XI*)*. pp. 72–84. – ACM Press, octobre 2004.