Espionnage des bus de communication dans un système embaqué

Problématique et solutions

Guillaume Duc
<guillaume.duc@telecom-paristech.fr>
2016–2017

Plan

Objectifs du cours

- Étudier les différents mécanismes permettant de garantir la confidentialité et l'intégrité des données dans un système embarqué en présence d'un adversaire capable d'espionner les différents bus (mémoire, périphériques...) de ce dernier
- Étudier les différentes solutions permettant de démarrer de façon sécurisée un système d'exploitation (*Secure Boot*)

Exemple : Microsoft XBox I

Exploit [9]

- Objectif de sécurité : empêcher l'exécution de logiciels non prévus (OS ou applications)
- Chaîne de confiance : chaque étape déchiffre et vérifie la suivante avant de l'exécuter
  - Bloc de démarrage secret (ASIC Southbridge)
  - Bootloader (Flash)
  - Système d'exploitation (Flash)
  - Applications (Disque dur ou support optique)

Exemple : Microsoft XBox I

Architecture matérielle

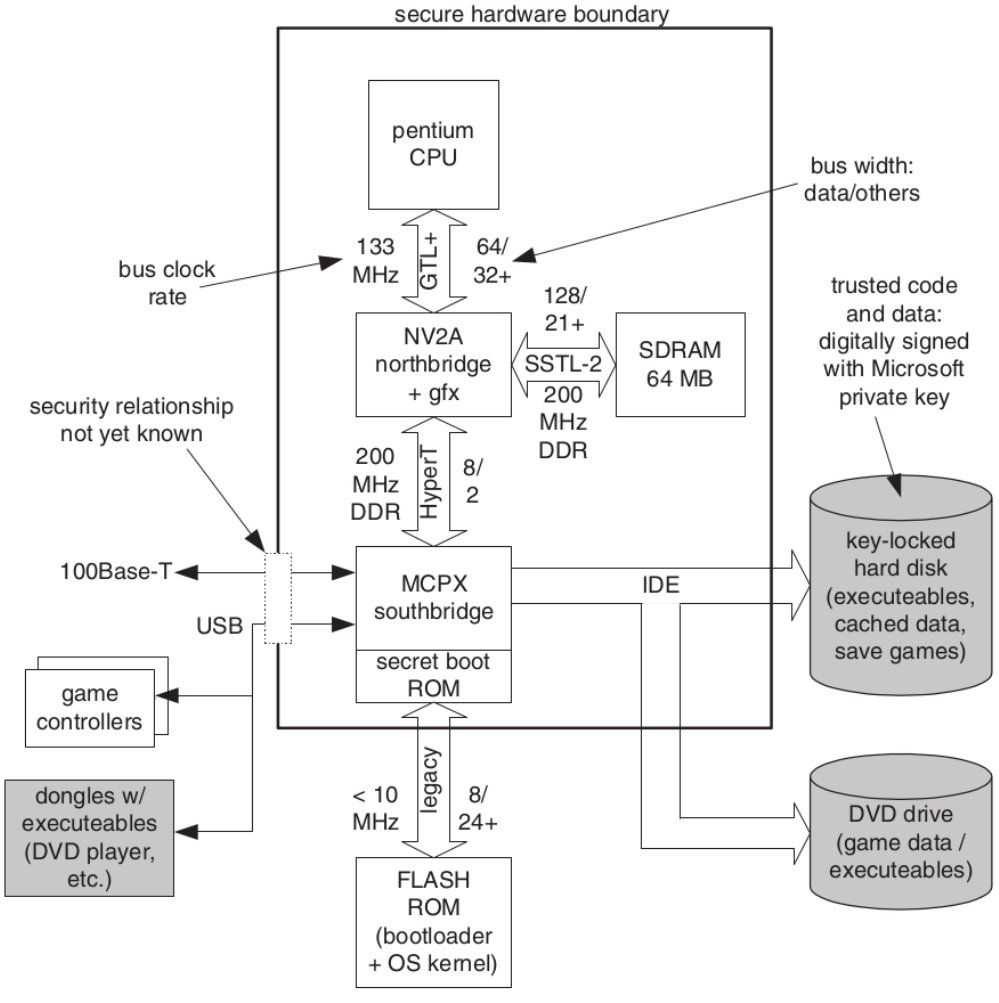


Fig. 1. Overview of the Microsoft Xbox hardware.

Source : Andrew “bunnie” Huang [9]

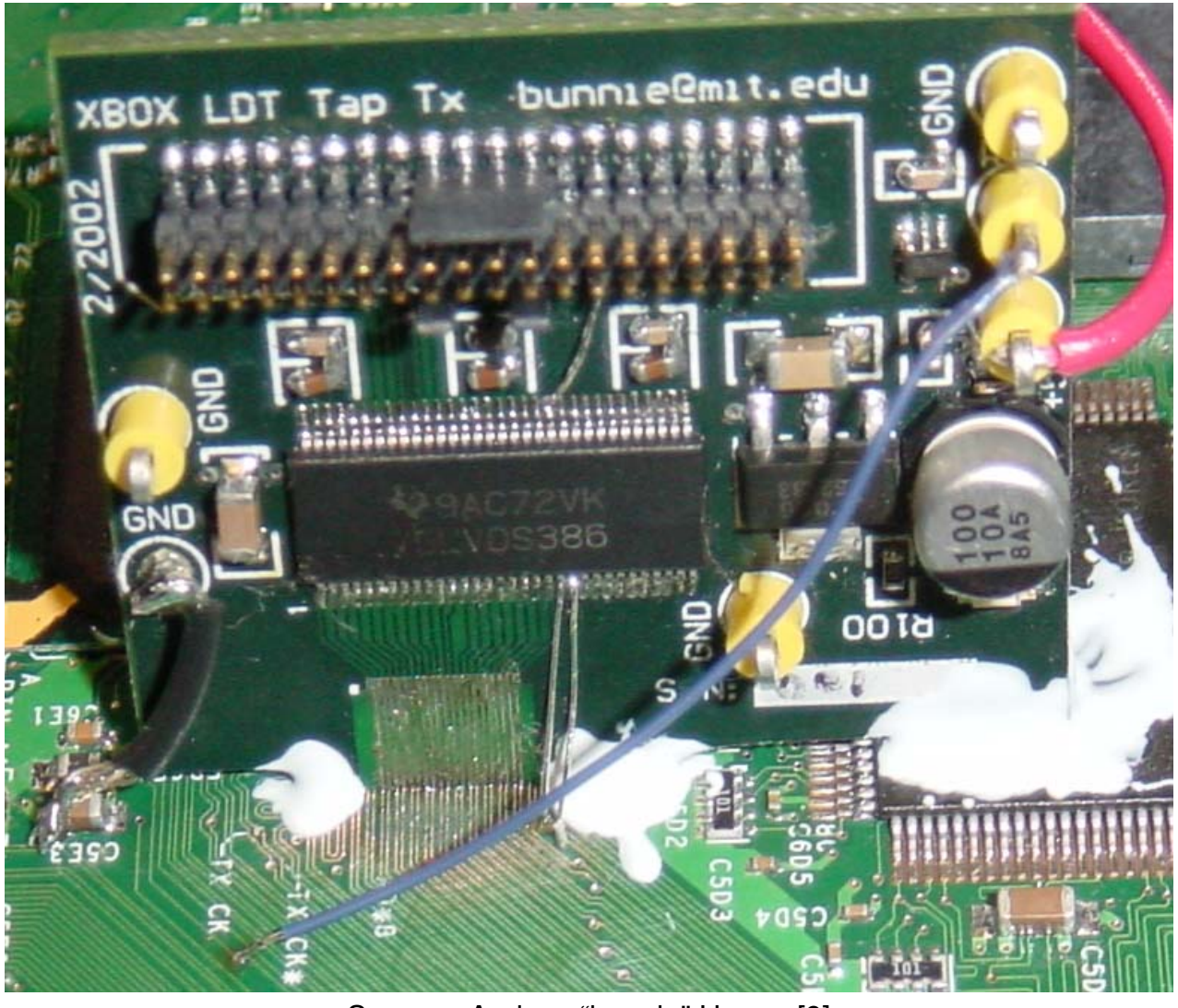
Exemple : Microsoft XBox I

Espionnage bus Northbridge/Southbridge

- Au démarrage, le code du bloc de démarrage secret transite par le bus Northbridge $\leftrightarrow$ Southbridge (*HyperTransport*) pour être exécuté par le processeur
- Ce code a été récupéré (voir [9]) en espionnant ce bus et analysé
- Il effectue plusieurs opérations
  - Initialisation du processeur et des périphériques (via une petite machine virtuelle exécutant des instructions en mémoire flash)
  - Chargement et déchiffrement du bootloader depuis la flash
  - Vérification de l'intégrité du bootloader
  - Si le bootloader est bien celui attendu : lancement de ce dernier

Exemple : Microsoft XBox I

Espionnage bus Northbridge/Southbridge



Source : Andrew “bunnie” Huang [9]

Exemple : Microsoft XBox I

Vulnérabilités du bloc de démarrage

- Bootloader chiffré grâce à une variante de l'algorithme RC-4 et une clé contenue dans le code de démarrage (qui a été récupérée lors de l'espionnage du bus)
- Vérification d'intégrité : valeur magique à la fin du bootloader...
- Instructions pour l'initialisation du système chargées depuis une partie non protégée de la flash : possibilité de les modifier pour configurer le système de afin de démarrer même si la vérification du bootloader échoue
- $\rightsquigarrow$ Avec ces informations, il est facile de générer un bootloader qui soit correctement déchiffré et vérifié (et donc accepté) par la XBox

Espionnage des bus

- Les différents bus de communication d'un système embarqués peuvent être espionnés
- Il est même possible, dans certains cas, pour un attaquant de modifier à la volée les informations circulant sur ces bus
- Plus simple à réaliser que des attaques contre les circuits électronique (*on-chip probing*)
  - Ex. Xbox : étudiant, environ 100$ de matériel
- Mais ces attaques deviennent tout de même de moins en moins simples
  - Fréquences des bus de plus en plus élevées
  - Pistes routées dans les couches internes du PCB
  - RAM et SoC superposés

Autres menaces

- L'espionnage ou la modification des données sur le bus ou dans la mémoire n'est pas seulement réalisable en se branchant physiquement sur le bus
- Attaques DMA (*Direct Memory Access*)
  - De nombreux périphériques sont capables, via le DMA, de lire ou de modifier des données en mémoire principale
  - Une exploitation par un adversaire d'un de ces périphériques lui permet ainsi d'attaquer la confidentialité ou l'intégrité des données en mémoire de façon en général plus simple
  - Exemple : attaque via le bus *FireWire*
  - Certaines architectures récentes tendent à utiliser le DMA sur des adresses virtuelles pour redonner à la MMU et à l'OS la possibilité de contrôler, au moins partiellement, ces transferts

Espionnage des bus

Contre-mesures passives

- Rendre l'accès aux composants difficile
- Empêcher l'identification des composants
- Utiliser des packages type BGA (*Ball Grid Array*)
- Supprimer les points de test
- Encapsuler les composants dans de la résine
- Routage (pistes internes, routage volontairement complexe)
- Bus plus complexes (fréquence, taille des données, etc.)

Espionnage des bus

Contre-mesures actives

- Contres-mesures passives : rendre l'attaque plus compliquée mais ne sont pas parfaites $\rightsquigarrow$ nécessité de contres-mesures actives
- Les données utiles transmises sur ces bus peuvent être chiffrées et protégées en logiciel
  - Bus MII/GMII/RMII (lien MAC $\leftrightarrow$ PHY Ethernet) : n'utiliser que des protocole réseaux sécurisés (IPsec, TLS...)
  - Bus IDE/SCSI/SATA : utiliser des mécanismes de chiffrement des disques
  - Bus *simples* SPI/I2C... : moins simple à faire mais utilité dans le cadre du modèle de faute considéré ?
- À l'exception du bus mémoire qui ne peut pas être protégé en logiciel uniquement

Espionnage du bus mémoire

- La suite de ce cours va se focaliser sur la protection des données transitant sur le bus mémoire
- Objectifs de sécurité sur ces données
  - *Confidentialité* (contre l'espionnage)
  - *Intégrité* (contre l'injection)
- Hypothèses pour la suite
  - L'attaquant a accès à tout le système (matériel et logiciel)
  - L'attaquant ne peut pas monter des attaques contre les circuits (l'espionnage ou la perturbation des composants par des méthodes passives ou actives doivent être résolu par d'autres méthodes)

Petit historique

Solutions académiques et industrielles

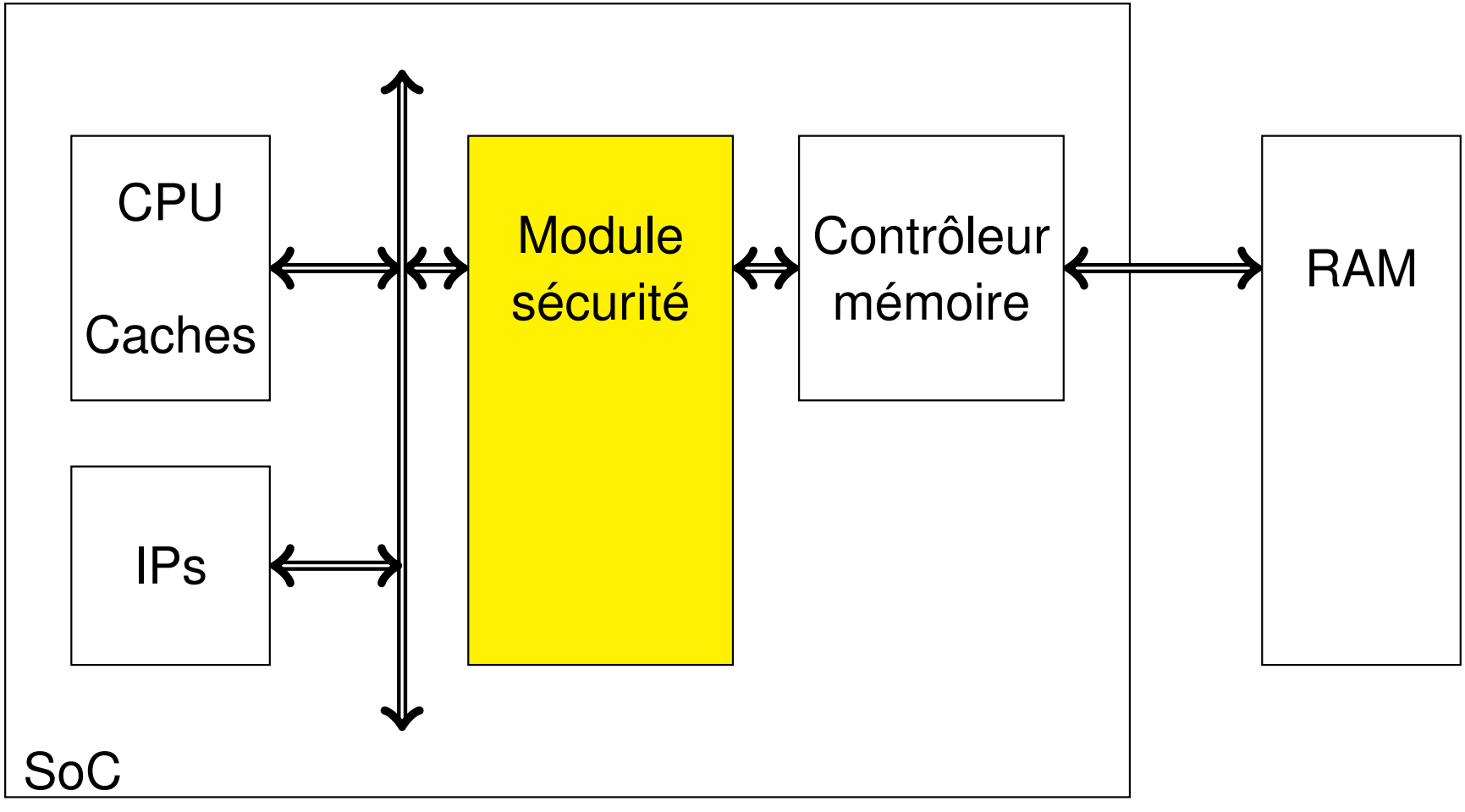
- Confidentialité uniquement
  - BEST, 1979 (brevets) [2, 4, 5, 3]
  - DALLAS DS500*x*, 1995 [6] (commercialisé, cassé par KUHN en 1998 [12])
  - KUHN (*TrustNo 1*), 1997 : chiffrement asymétrique et support d'un OS [11]
  - GILMONT, LEGAT et QUISQUATER, 1999 : chiffrement hybride [8]
- Confidentialité et intégrité
  - LIE, THEKKATH, MITCHELL, LINCOLN (XOM), 2000 [14]
  - KERYELL (CryptoPage), 2000 [10]
  - SUH, CLARKE, GASSEND, DIJK et DEVADAS (*Aegis*), 2003 : protection contre les attaques par rejeu [15]
  - KERYELL, LAURADOUX (CryptoPage 2), 2003 [13]
  - INTEL Software Guard Extensions (SGX), 2015 [1]

Plan

Introduction

- Pour protéger la confidentialité des données transmises sur le bus mémoire, il est nécessaire de les chiffrer
- Cette opération ne peut être réalisée en logiciel $\rightsquigarrow$ nécessité d'un module matériel *ad hoc*
- D'après le modèle d'attaque, tout ce qui est à l'extérieur du SoC est potentiellement sous le contrôle de l'adversaire donc le chiffrement/déchiffrement doit intervenir sur le SoC lui-même
- Afin de conserver au maximum les IP existantes, les données doivent transiter en clair au sein du bus interne au SoC
- Le chiffrement/déchiffrement intervient sur le SoC, au plus près du monde extérieur

Module matériel



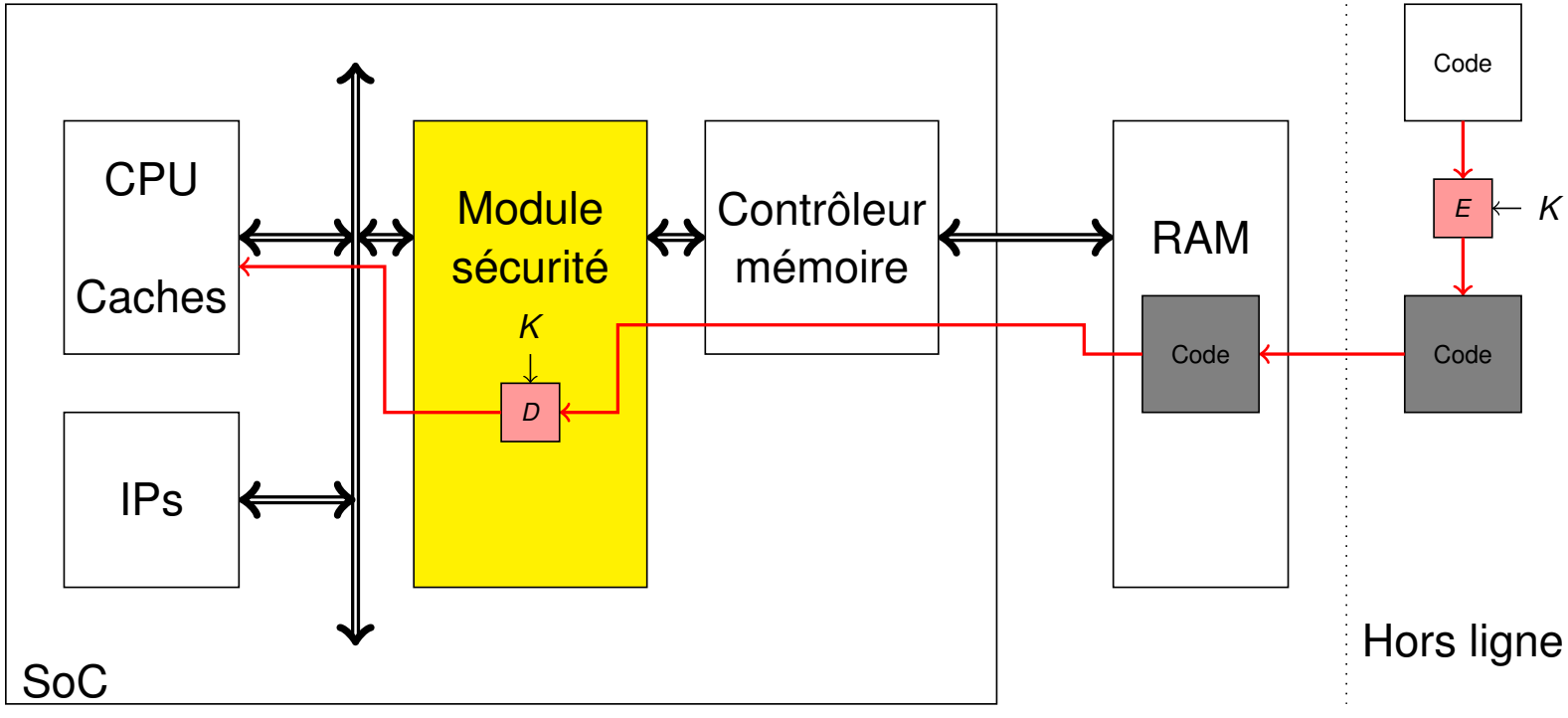
Solution matérielle uniquement

- Simple (pas d'interaction matériel/logiciel, pas besoin de se préoccuper de la sécurité d'une partie logicielle...)
- Protection monolithique (toute la mémoire ou une portion fixe)
- Manque de flexibilité (application du chiffrement en tout ou rien)
- Dégradation systématique des performances

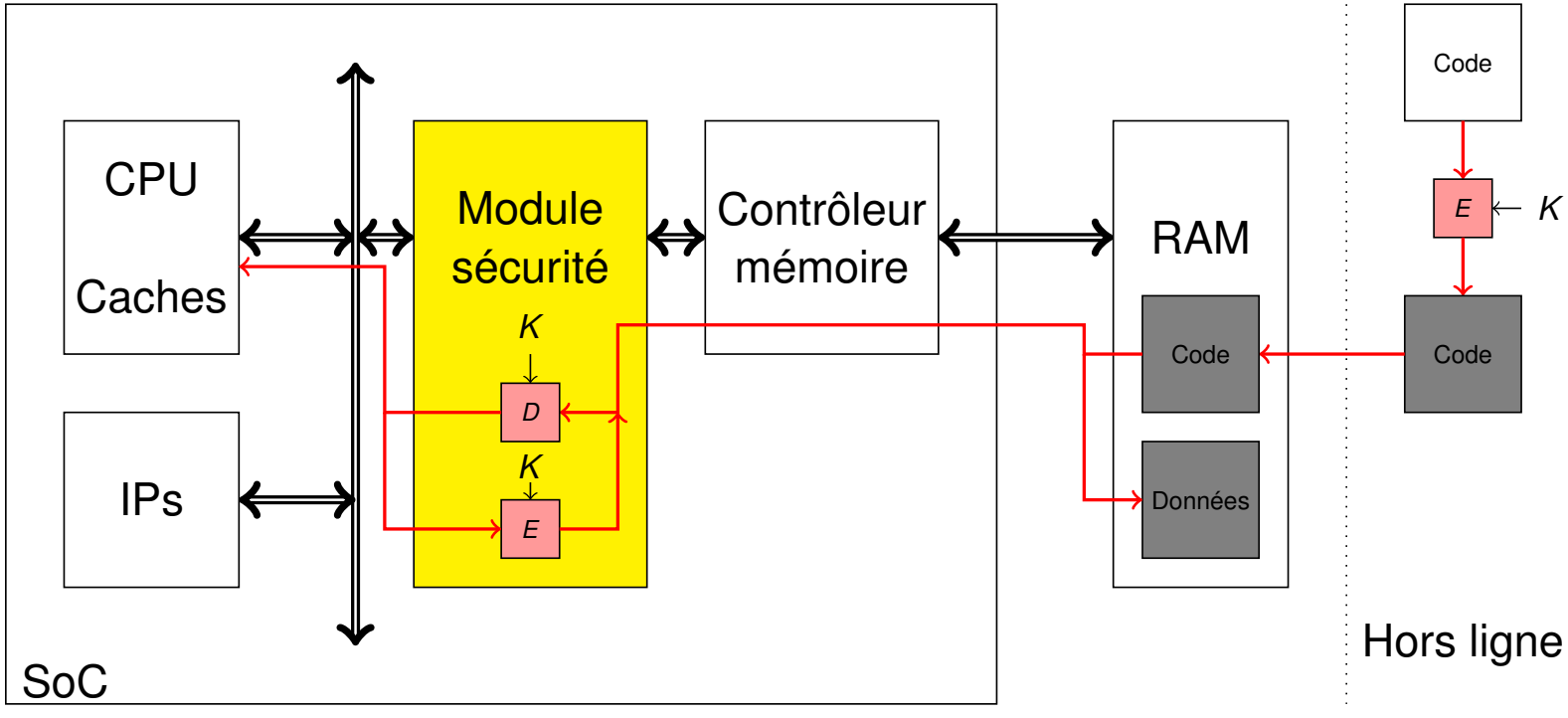
Solution mixte matérielle + logicielle

- Une partie logicielle indique au module matériel comment protéger chaque zone mémoire (à la manière d'une table de page pour une MMU par exemple)
- Flexible
  - Protection à granularité fine (page mémoire par exemple)
  - Politiques de sécurité différenciées en fonction des applications, des données à protéger...
- Impact sur les performances minimisé (application de la sécurité adaptée aux données)
- Complexité plus importante
- Il est nécessaire de garantir la sécurité du module logiciel
  - Garantir son intégrité au démarrage et pendant la vie de la plate-forme
  - Absence de bug : petit, vérification formelle
  - Peut être intégré dans un hyperviseur, un micro-noyau, etc.

Chiffrement du code uniquement



Chiffrement du code et des données



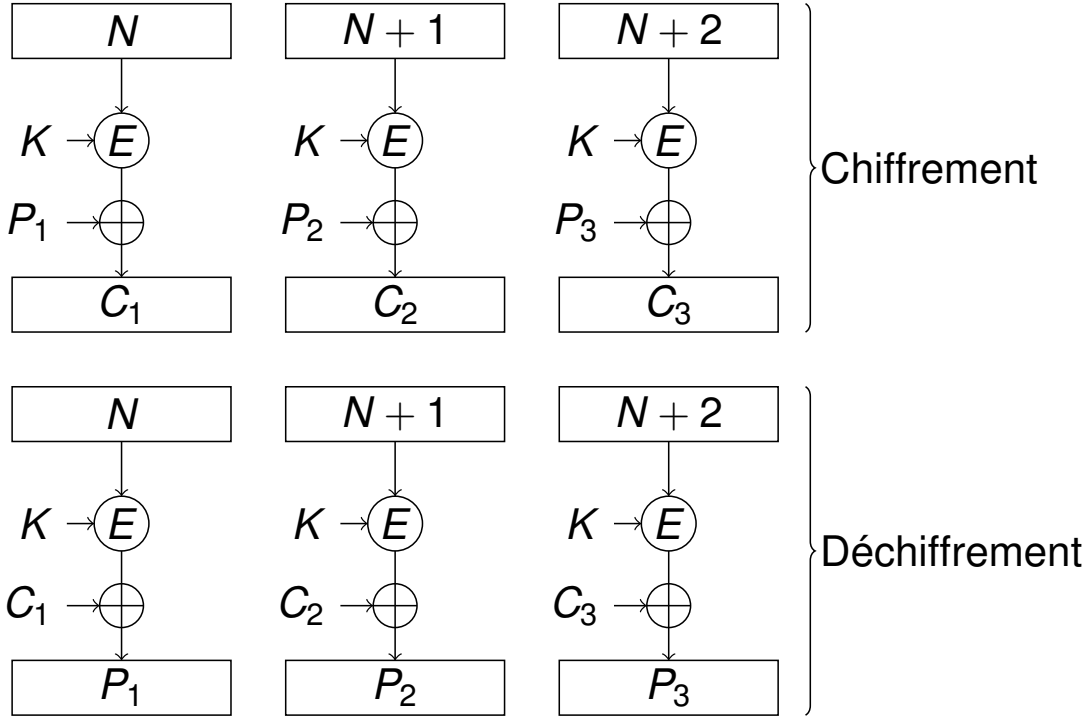
Coûts et Performances

- Les opérations de chiffrement et de déchiffrement entraînent une pénalité au niveau des performances
- Pénalité uniquement au moment d'un accès mémoire (défaut de cache du processeur ou accès à la RAM par une IP)
- Lors d'une lecture, la latence du déchiffrement s'ajoute à celle de l'accès mémoire augmentant ainsi la pénalité en cas de défaut de cache
- En général, la latence introduite par l'opération de chiffrement n'a pas d'impact sur les performances du système
- Le module matériel entraîne une augmentation de la surface silicium (dont du prix du circuit) et de la consommation

Optimisations

- Le mode d'opération de l'algorithme de chiffrement peut avoir un impact important sur les performances
- L'objectif est de réduire au minimum la latence supplémentaire lorsque la données chiffrée arrive (après avoir été lue depuis la mémoire) au sein du module de sécurité
- Le mode compteur d'un algorithme de chiffrement par bloc ou un algorithme de chiffrement de flux peut être utile (mais dans certains cas dangereux...)

Rappel : Mode compteur (*Counter Mode*)



- N est un nombre aléatoire
- On ne doit jamais chiffrer deux blocs avec le même compteur

Optimisations : Mode compteur

- Pour calculer le masque ($E_K(N)$), qui constitue l'opération la plus longue, seuls K et N sont nécessaires (pas la donnée chiffrée)
- Une fois le masque calculé, il suffit d'un XOR entre ce dernier et la donnée chiffrée pour déchiffrer
- Donc si K et N sont disponibles, il est possible de paralléliser l'accès mémoire pour récupérer la donnée chiffrée et le calcul du masque
  - K est par hypothèse toujours disponible
  - Comment choisir N pour qu'il soit toujours disponible dès le début de l'accès mémoire ?

Optimisations : Mode compteur

Choix du compteur *N*

- Il ne faut pas avoir besoin de le stocker en mémoire sinon on perd l'intérêt du mode compteur (latence mémoire pour le récupérer)
- Utiliser l'adresse comme valeur de compteur (*N*)
- Néanmoins, il ne faut jamais chiffrer plusieurs données avec la même clé et la même valeur de compteur
- On peux donc appliquer cette astuce uniquement aux données en lectures seules (code et données RO)

Optimisations

- Seules les données confidentielles ont besoin d'être chiffrées
- Il est possible de rendre l'opération de chiffrement dépendant de l'adresse de la donnée

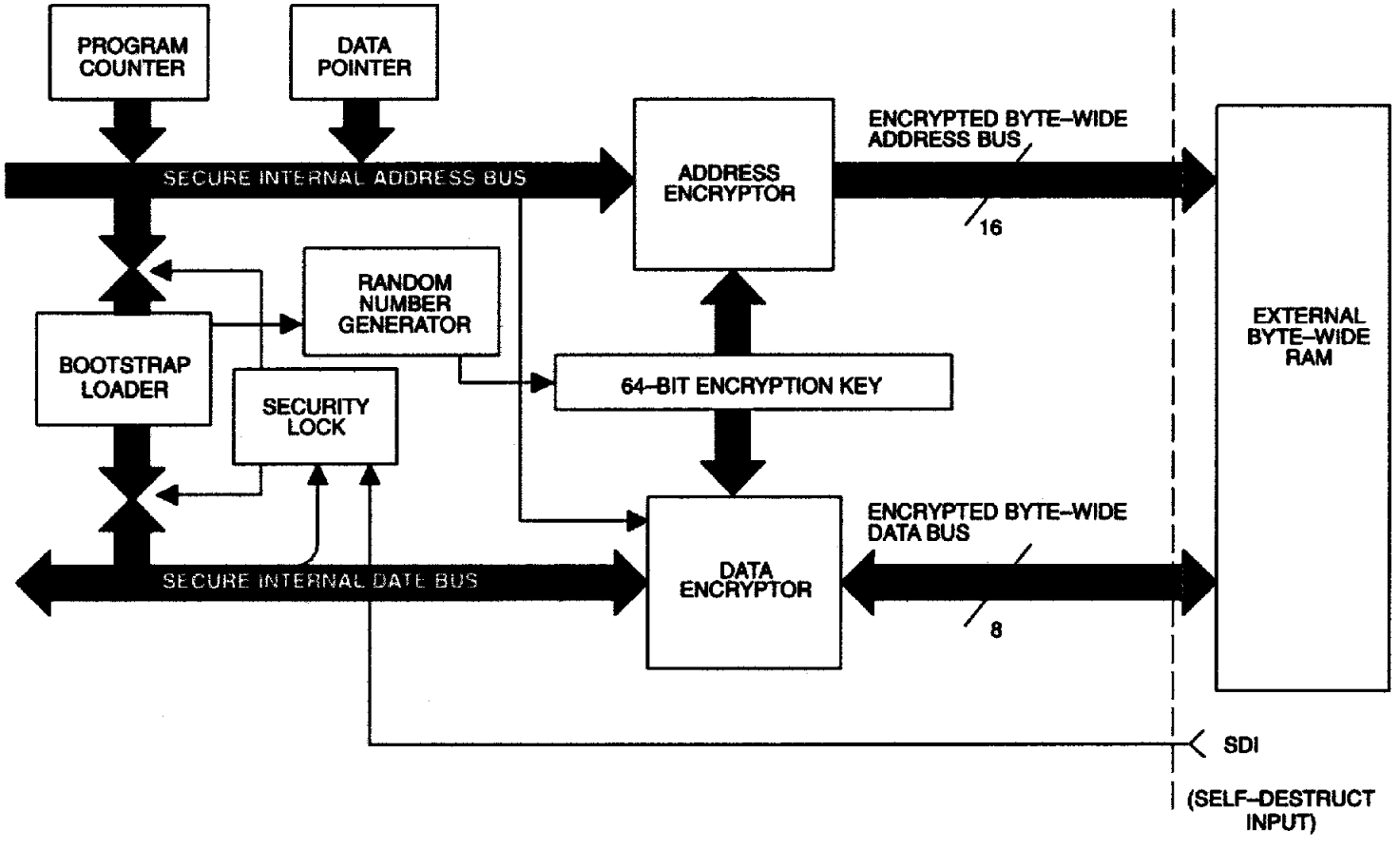
Chiffrement et intégrité

- Le chiffrement seul est en général insuffisant et ne garantie pas l'intégrité des données
- Si un adversaire peut modifier les données chiffrées, ces modifications ne seront pas détectées et le processeur va utiliser les données modifiées issues du déchiffrement
- Cela va perturber le fonctionnement du système voire, dans certains cas, laisser fuir des informations sensibles malgré le chiffrement
- Exemple : l'attaque du Dallas DS5002FP par M. Kuhn [12]

DS5002FP

- Microcontrôleur 8-bit fabriqué par Dallas Semiconductor
- Bus de données (8 bits) chiffré ($d' = \mathrm{ED}_{K,a}(d)$)
- Bus d'adresses (17 bits) chiffré ($a' = \mathrm{EA}_K(a)$)
- Clé K (64 bits) stockée dans une petite SRAM au sein du microcontrôleur et alimentée par une batterie
- Le programme est chargé en utilisant un mode spécial (le microcontrôleur génère une clé aléatoire, charge le programme en clair depuis un port série, le chiffre à la volée et le stocke chiffré dans la mémoire)
- Des accès mémoire inutiles sont ajoutés afin de cacher les motifs d'accès à la mémoire
- Un mécanisme d'*auto-destruction* efface la clé si nécessaire

DS5002FP



Source : Datasheet du DS5002FP

Attaque du DS5002FP

- Pour chaque adresse, la fonction de chiffrement $\mathrm{ED}_{K,a}$ est une bijection $\{0,1\}^{256} \to \{0,1\}^{256}$
- Montage : commutateur permettant de connecter le bus de données soit à la mémoire, soit à une FIFO remplie par l'adversaire
- On cherche tout d'abord l'instruction en clair `MOV 0xA0, V` (`0x75 0xA0 0x42`) qui envoie la valeur `V` en clair sur le premier port parallèle du microprocesseur
  - Injection de 5 octets : $X, Y, Z, 0, 0$ en itérant sur toutes les valeurs possibles de X, Y et Z (256^3 valeurs possibles) et on regarde la valeur T présentée sur le port parallèle
  - On cherche quand se produit une bijection entre T et Z

Attaque du DS5002FP

- Une fois trouvé une telle bijection on a :
  - $X = \mathrm{ED}_{K,a_0}(0x75)$
  - $Y = \mathrm{ED}_{K,a_0+1}(0xA0)$
  - On a tabulé la fonction ED_{K,a_0+2} ($Z \leftrightarrow T$)
- On cherche ensuite une instruction codée sur un seul octet et qui ne cause aucun effet de bord visible (exemple `NOP`)
  - Injection de $X, 0, Y, \mathrm{ED}_{K,a_0+2}(0xA0), Z, 0, 0$ en cherchant toujours une bijection entre Z et T
  - On tabule ainsi la fonction ED_{K,a_0+3}
- Et ainsi de suite, on peut tabuler la fonction de chiffrement/déchiffrement pour une plage d'adresses
- Il suffit ensuite de construire un petit programme dumpant le contenu de la mémoire vers le port parallèle

Chiffrement et intégrité (suite)

- Des solutions d'intégrité *imparfaites*, uniquement basées sur le chiffrement, sont néanmoins possibles
- Exemple : Avec un algorithme de chiffrement offrant une bonne diffusion d'erreur (ce qui exclut la plupart des algorithmes de chiffrement par flot et le mode compteur), pour du code (uniquement), avec une taille de bloc plus grande que celle des instructions, et avec un jeu d'instruction clairsemé, la plupart des modifications au hasard (pas les permutations spatiales ni temporelles) donneront des instructions invalides
- Ces solutions sont *imparfaites* !

Chiffrement des adresses

- Chiffrement des adresses par exemple sur le DS5002FP ($a' = \mathrm{EA}_K(a)$) et injection de fausses requêtes mémoire lorsque le bus mémoire n'est pas utilisé
- Néanmoins, ce mélange des adresses ne masque pas les motifs d'accès (boucles par exemple) et permet l'identification de certains algorithmes [16]
- Solution compliquée surtout si on veut conserver les principes de localité spatiale et temporelle (caches)

Plan

Introduction

- Fonctionnement correct de la mémoire : la valeur renvoyée par la mémoire lors d'une lecture à l'adresse A doit être la dernière valeur stockée par la processeur à cette même adresse
- Trois possibilités d'attaque
  - Injection
  - Permutation spatiale
  - Rejeu (ou permutation temporelle)

Solution 1

Fonction de hachage + stockage sécurisé

- Le processeur veut stocker D à l'adresse A
  - Le module de sécurité calcule $H = h(D)$ (où h est une fonction de hachage), envoie D à la mémoire et stocke H dans une mémoire sécurisée interne
- Le processeur lit à l'adresse A
  - La mémoire renvoie D' ($D' = D$ si tout va bien)
  - Le module de sécurité calcule $H' = h(D')$ et compare H' à la valeur de H
  - Si $H = H'$, alors $D = D'$ (avec une très forte probabilité), sinon détection d'une attaque

Solution 1

Fonction de hachage + stockage sécurisé

- Injection
  - Comme l'adversaire ne peut pas modifier H (stockage sécurisé), pour réussir une injection il doit trouver $D' \neq D$ tel que $h(D') = H = h(D)$ ce qui est difficile (propriété de résistance à la seconde pré-image des bonnes fonctions de hachage)
- Problème : besoin d'un stockage sécurisé sur la puce, H ne pouvant pas être stocké en mémoire (sinon il suffit que l'adversaire remplace aussi H)
- Cette solution ne passe pas à l'échelle

Solution 2

MAC (fonction de hachage à clé)

- Le processeur veut stocker D à l'adresse A
  - Le module de sécurité calcule $H = h_K(D)$ (où h_K est une fonction de MAC utilisant la clé secrète K), envoie D et H à la mémoire
- Le processeur lit à l'adresse A
  - La mémoire renvoie D' et H' ($D' = D$ et $H' = H$ si tout va bien)
  - Le module de sécurité calcule $H'' = h_K(D')$ et compare H'' à la valeur de H'
  - Si $H'' = H'$, alors $H' = h_K(D')$ et donc D' est bien une valeur stockée par le processeur (avec une grande probabilité)

Solution 2

MAC (fonction de hachage à clé)

- Injection
  - Pour réussir une injection l'adversaire doit trouver $D' \neq D$ et $H' \neq H$ tel que $H' = h_K(D')$ ce qui est difficile s'il ne connaît pas K (propriété des bonnes fonctions de MAC)
- Problèmes
  - Augmentation de l'empreinte mémoire (stockage des MAC)
  - Vulnérable aux permutations spatiales
    - Le processeur stocke à l'adresse A_0 : $D_0, H_0 = h_K(D_0)$
    - Le processeur stocke à l'adresse A_1 : $D_1, H_1 = h_K(D_1)$
    - L'adversaire copie le contenu de l'adresse A_1 à l'adresse A_0
    - Valeur à l'adresse A_0 après l'attaque : $D_1, H_1 = h_K(D_1)$
    - Attaque non détectée car $H_1 = h_K(D_1)$

Solution 3

MAC avec utilisation de l'adresse

- Prendre en compte l'adresse dans le calcul du hash ou du MAC
- Le processeur veut stocker D à l'adresse A
  - Le module de sécurité calcule $H = h_K(D, A)$ (où h_K est une fonction de MAC utilisant la clé secrète K), envoie D et H à la mémoire
- Le processeur lit à l'adresse A
  - La mémoire renvoie D' et H' ($D' = D$ et $H' = H$ si tout va bien)
  - Le module de sécurité calcule $H'' = h_K(D', A)$ et compare H'' à la valeur de H'
  - Si $H'' = H'$, alors $H' = h_K(D', A)$ et donc D' est bien une valeur stockée par le processeur à l'adresse A (avec une grande probabilité)

Solution 3

MAC avec utilisation de l'adresse

- Permutation spatiale
  - Le processeur stocke à l'adresse A_0 : $D_0, H_0 = h_K(D_0, A_0)$
  - Le processeur stocke à l'adresse A_1 : $D_1, H_1 = h_K(D_1, A_1)$
  - L'adversaire copie le contenu de l'adresse A_1 à l'adresse A_0
  - Valeur à l'adresse A_0 après l'attaque : $D_1, H_1 = h_K(D_1, A_1)$
  - Attaque détectée car $H_1 = h_K(D_1, A_1) \neq h_K(D_1, A_0)$

Solution 3

MAC avec utilisation de l'adresse

- Vulnérable aux attaque par rejeu
  - Au temps t_1, le processeur stocke à l'adresse A : $D, H = h_K(D, A)$
  - Au temps $t_2 > t_1$, le processeur stocke à l'adresse A : $D', H' = h_K(D', A)$
  - Au temps $t_3 > t_2$, l'adversaire, qui avait sauvegardé la valeur écrite à l'instant t_1 la replace en mémoire à l'adresse A
  - Au temps $t_4 > t_3$, le processeur récupère la valeur à l'adresse A : $D, H = h_K(D, A)$
  - L'attaque n'est pas détectée

Solution 4

MAC avec utilisation de l'adresse et du temps

- $H = h_K(D, A, t)$ où t est un compteur ou bien le temps lors de l'écriture
- Nécessité de stocker t pour chaque adresse (pour pouvoir recalculer le MAC lors de la lecture)
- De plus ces compteurs/temps doivent être stockés de façon non rejouable $\rightsquigarrow$ mémoire sécurisée interne
- Solution valable uniquement si les données à protéger sont petites (ne passe pas à l'échelle)

Solutions 2–4

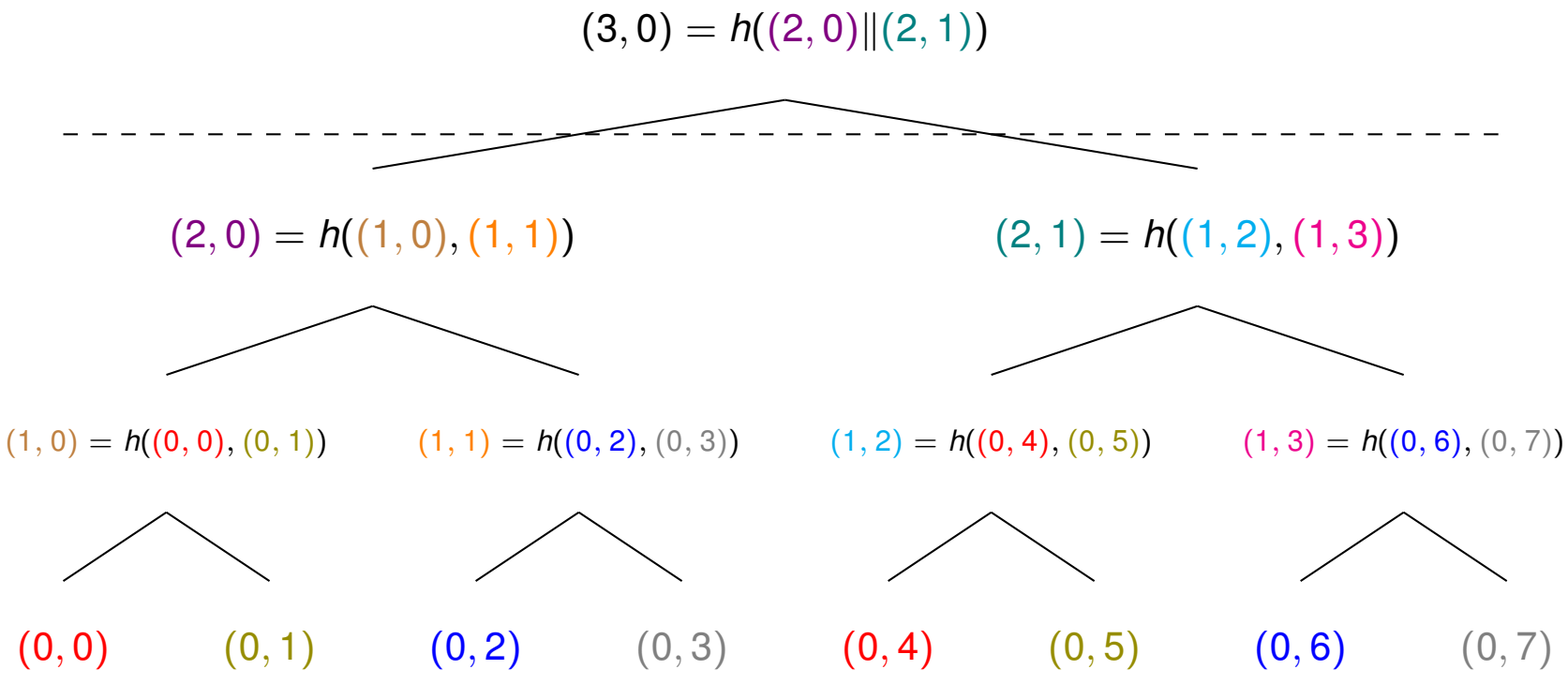
Couplage avec le chiffrement

- Si la confidentialité doit également être assurée, il existe plusieurs schémas possibles
  - *Encrypt-then-MAC*
  - *Encrypt-and-MAC*
  - *MAC-then-Encrypt*
- Utilisation possible de mode combinés comme les mode CCM (*Counter with CBC-MAC*), GCM (*Galois/Counter mode*)...

Arbres de hachage (ou arbres de Merkle)

- Objectif : calculer un résumé cryptographique de toute la mémoire et le stocker dans une mémoire protégée (petite)
- Problème : trop coûteux
- Idée : calculer ce résumé de façon hiérarchique grâce à une structure en arbre
  - Feuilles de l'arbre : données à protéger
  - Nœuds : haché (ou MAC) de ses fils
  - Racine stockée dans une mémoire sécurisée alors que les autres nœuds peuvent être stockés en mémoire ($\rightsquigarrow$ besoin en mémoire sécurisée très faible)

Arbres de hachage (ou arbres de Merkle)



Fonctionnement

- Lors d'une lecture, la branche correspondante (jusqu'à la racine) est recalculée et comparée
- Lors d'une écriture, la branche correspondante est mise à jour
- Voir algorithmes de lecture et d'écriture dans les transparents suivants
- Notations
  - a est l'arité de l'arbre
  - s est la profondeur de l'arbre

Algorithme de lecture vérifiée

1: $(u, v_0, v_1) \leftarrow (i, j_0, j_1)$ ▷ Temporary indices
2: **while** $u \leq s$ **do** ▷ For tree levels
3: **if** $u = s$ **then** ▷ If root node
4: $X_0 \leftarrow \text{Read}(s, 0)$ ▷ Read root from secure area
5: **if** $X_0 \neq D$ **then** ▷ If digest mismatch
6: **error** ▷ Abort
7: **end if**
8: **return** R ▷ Return requested verified node and stop
9: **end if**
10: **for** $k \in \{0, ..., a-1\}$ **do** ▷ For a siblings
11: $X_k \leftarrow \text{Read}(u, a \times v_0 + k)$ ▷ Un-verified read
12: **end for**
13: **if** $u = i$ **then** ▷ If starting level, no check
14: $R \leftarrow X_{v_1}$ ▷ Value to return
15: **else if** $X_{v_1} \neq D$ **then** ▷ If digest mismatch
16: **error** ▷ Abort
17: **end if**
18: $D \leftarrow \text{Digest}(X_0, ..., X_{a-1})$ ▷ Compute parent node
19: $(u, v_0, v_1) \leftarrow (u + 1, \lfloor \frac{v_0}{a} \rfloor, v_0 \bmod a)$ ▷ One level up
20: **end while**

Algorithme d'écriture vérifiée

1: $(u, v_0, v_1) \leftarrow (i, j_0, j_1)$ ▷ Temporary indices
2: **while** $u \leq s$ **do** ▷ For tree levels
3: **if** $u = s$ **then** ▷ If root node
4: $X_0 \leftarrow \text{Read}(s, 0)$ ▷ Read root from secure area
5: **if** $X_0 \neq D_{old}$ **then** ▷ If digest mismatch
6: **error** ▷ Abort
7: **end if**
8: $\text{Write}((s, 0), D_{new})$ ▷ Write new root node value in secure area
9: **exit** ▷ And stop
10: **end if**
11: **for** $k \in \{0, \ldots, a-1\}$ **do** ▷ For a siblings
12: $X_k \leftarrow \text{Read}(u, a \times v_0 + k)$ ▷ Un-verified read
13: **end for**
14: **if** $u = i$ **then** ▷ If starting level, no check
15: $D_{new} \leftarrow V$ ▷ Value to write
16: **else if** $X_{v_1} \neq D_{old}$ **then** ▷ If digest mismatch
17: **error** ▷ Abort
18: **end if**
19: $D_{old} \leftarrow \text{Digest}(X_0, \ldots, X_{a-1})$ ▷ Compute old parent node
20: $X_{v_1} \leftarrow D_{new}$ ▷ New node value
21: $\text{Write}((u, a \times v_0 + v_1), X_{v_1})$ ▷ Un-verified write new node value
22: $D_{new} \leftarrow \text{Digest}(X_0, \ldots, X_{a-1})$ ▷ Compute new parent node
23: $(u, v_0, v_1) \leftarrow (u + 1, \lfloor \frac{v_0}{a} \rfloor, v_0 \bmod a)$ ▷ One level up
24: **end while**

Coûts

Espace de stockage (exemple)

- Taille d'une ligne de cache : 32 octets (256 bits) (ex. ARM Cortex A9)
- Taille de la zone à protéger : 128 Mio (2^{27} octets)
- Taille élémentaire des blocs à protéger : 8 octets
- Nombre de blocs à protéger : $N = 2^{27}/2^3 = 2^{24} = 16777216$
- Arité de l'arbre : $a = 4$
- Profondeur de l'arbre : $s = \log_a(N) = 12$
- Nombre de nœuds intermédiaires dans l'arbre : $N_i = a + a^2 + \cdots + a^{s-1} = \frac{1-a^s}{1-a} - 1 = 5592404$
- Taille d'un haché : 8 octets / 64 bits (permet de stocker les 4 hachés dans une seule ligne) mais moins résistant
- Taille de l'arbre : $S = N_i \times 8 \approx 42{,}7$ Mio
- Surcoût en espace de stockage : 33,3 %

Coûts

Performances en lecture

- La lecture d'une ligne (32 octets) se transforme en
  - Lecture de la ligne utile (qui dans notre exemple contient les 4 feuilles)
  - Lecture des nœuds intermédiaires et de leur frères : $s - 1$ lignes (dans l'hypothèse où une ligne contient a nœuds)
    - Exemple : 11 lignes
  - Calcul des hachés : $s + 1$
    - Exemple : 13 calculs de la fonction de hachage

Coûts

Performances en écriture

- Écriture d'une ligne (32 octets) se transforme en
  - Écriture de la ligne utile (qui dans notre exemple contient les 4 feuilles)
  - Lecture des nœuds intermédiaires et de leur frères : $s - 1$ lignes (pour vérification)
    - Exemple : 11 lignes
  - Écriture des nœuds intermédiaires : $s - 1$ lignes (mise à jour)
    - Exemple : 11 lignes
  - Calcul des hachés : $2 \times (s + 1)$ (vérification puis mise à jour)
    - Exemple : 26 calculs de la fonction de hachage

Optimisations

- Utilisation d'un cache stockant les nœuds intermédiaires les plus utilisés
- Le cache étant sur la puce, on peut stopper la vérification, lors d'une lecture, dès que l'on rencontre un nœud dans le cache (voir [7])
- De même, lors d'une mise à jour, on s'arrête dès que l'on rencontre un nœud dans le cache que l'on marque alors comme *dirty*. La mise à jour de la partie supérieure de l'arbre aura lieu quand ce nœud devra être évincé du cache

Gestion des clés

- Le chiffrement ainsi que le calcul des MAC nécessitent des clés
- Nombreuses possibilités pour la gestion de ces clés
  - Clé unique ou clés différenciées pour le chiffrement du code et des données ainsi que pour la confidentialité et l'intégrité
  - Clés différentes en fonction des applications
  - Clés statiques ou générées dynamiquement à chaque démarrage du SoC et/ou des applications
  - Mécanisme d'effacement automatique des clés en cas de détection d'attaque

Gestion des exécutables

- Le code d'une application et les données qu'elle manipule peuvent être protégés en confidentialité et en intégrité lors de son exécution
- Mais qu'en est-il de l'exécutable lui-même sur son support de stockage (flash, disque...) ?
- Dans de nombreuses situation, pour que la protection en mémoire ait un intérêt, il faut aussi que tout ou partie de l'exécutable (code et données) soit également protégé

Chiffrement des exécutables

Exemple de solution

- Matériel cryptographique intégré au sein du module de sécurité (par exemple : clé privée unique pour le circuit et clé(s) publique(s) servant à authentifier la (ou les) source(s) des applications)
- Exécutable chiffré avec la clé publique correspondant au module de sécurité ciblé (possibilité de chiffrement hybride) et signé numériquement
- Déchiffrement (vers une zone mémoire protégée en confidentialité et/ou intégrité en fonction des besoins des applications) et vérification en matériel par le module de sécurité (ou en logiciel si l'architecture permet le démarrage sécurisé du module de contrôle)

Plan

Objectifs

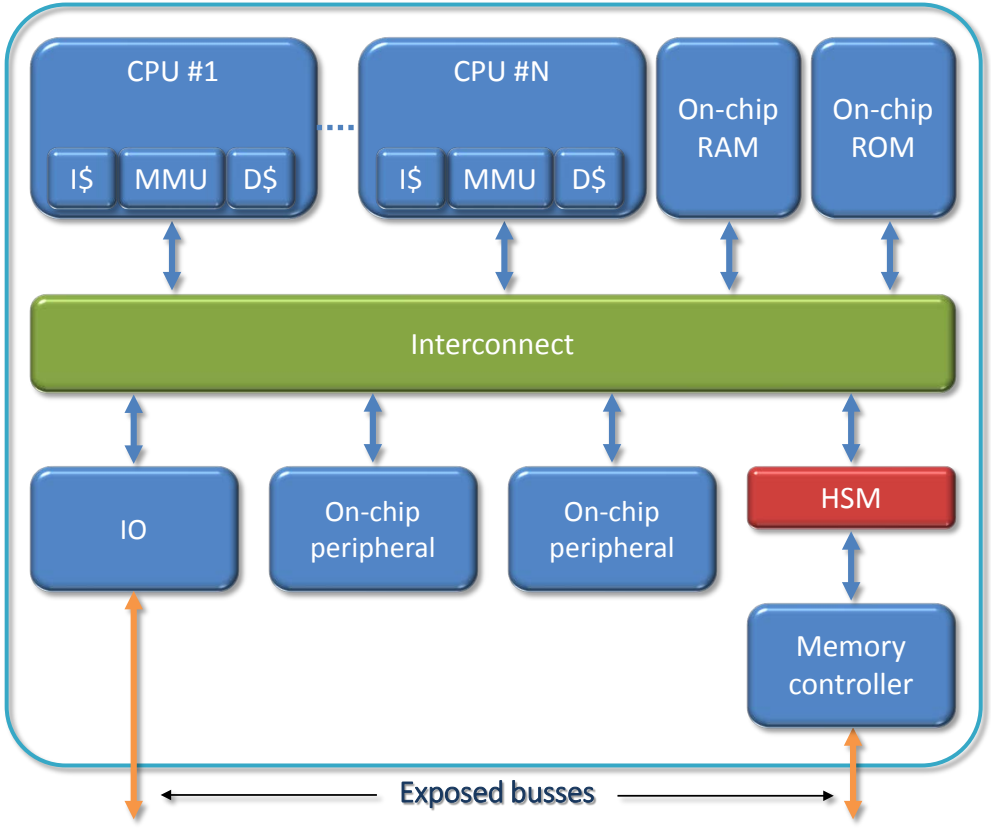
- Objectif : Garantir la confidentialité et/ou l'intégrité de tout ou partie de la mémoire contre un adversaire qui aurait le contrôle total sur tout ce qui est à l'extérieur du SoC principal du système embarqué
- Hypothèses
  - Garder la compatibilité avec les programmes existants et les outils de développement existants
  - Ne pas imposer de modifications au sein du (ou des) cœurs des processeurs du SoC

Développement

- Architecture SecBus développée principalement à Télécom ParisTech (L. Su, S. Courcambeck, P. Guillemin, C. Schwarz, R. Pacalet, J. Brunel, S. Ouaarab, A. Si Merabet, G. Duc)
- À l'aide de financements
  - Projet TRESCCA (European Community's Seventh Framework Programme)
  - Projet TOISE (ENIAC Joint Undertaking)
  - Financement ANRT (Association Nationale de la Recherche et de la Technologie)
  - STMicroelectronics
  - Région PACA
- Projet *open-source* : documentation, modèle SystemC et bientôt description VHDL disponibles gratuitement et sous licence libre à l'adresse

  `https://secbus.telecom-paristech.fr/`

SoC avec SecBus



Source : Renaud Pacalet

Partitionnement matériel/logiciel

- Afin de minimiser l'impact sur les performances, la protection (en confidentialité et/ou en intégrité) est configurable avec la granularité d'une page mémoire (i.e. chaque page peut être configurée indépendamment l'une de l'autre)
- Module matériel (*Hardware Security Module* : HSM) chargé d'appliquer, lors de chaque accès mémoire, la politique de sécurité
- Module logiciel (*Software Security Module* : SSM) chargé, en fonction des besoins des applications, de configurer le HSM

Primitives cryptographiques

- L'architecture ne fait pas d'hypothèses sur l'algorithme de chiffrement par bloc utilisé (laisse le choix du compromis performances/coûts)
- Pages en lecture seule
  - Confidentialité : mode compteur utilisant l'adresse
  - Intégrité : CBC-MAC
- Pages en lecture/écriture
  - Confidentialité : CBC
  - Intégrité : Arbres de MAC

Structures de données

- Deux types de pages mémoire
  - Pages maîtresses : données utiles
  - Pages esclaves : données supplémentaires pour la protection (MAC, arbres, IV)
- À chaque page est associée une PSPE (*Page Security Policy Entry*)
  - Pages maîtresses
    - Index de la *Security Policy*
    - Adresses des pages esclaves contenant les MAC ou l'arbre et les IV
    - Informations supplémentaires : taille, validité, protection
  - Pages esclaves
    - Racine de l'arbre (le cas échéant)
    - Taille

Structures de données

- *Security Policies*
  - Mode de confidentialité (aucun, CTR, CBC)
  - Mode d'intégrité (aucun, MAC, arbre)
  - Matériel cryptographique pour la confidentialité et l'intégrité
  - Validité

Master Block

- Stocké en mémoire externe
- Contient
  - Une PSPE par page mémoire
  - Un nombre suffisant de SP
- Les PSPE et les SP sont protégées en intégrité par un arbre (*Master MAC Tree*)
- Les SP sont également protégées en confidentialité (contiennent du matériel cryptographique)

Module matériel (HSM)

- Lors d'un accès vers la mémoire externe, le HSM :
  - Récupère la PSPE correspondante dans le MB
  - Récupère la SP indiquée par la PSPE
  - Applique la politique de sécurité (confidentialité ou intégrité)
- Pour des questions de performance, le HSM contient plusieurs caches : SP, PSPE, IV, MAC Tree, MAC

Module logiciel (SSM)

- Couplé au mécanisme d'allocation mémoire de l'OS ou de l'hyperviseur
- Configure les PSPE et les SP en fonction d'une politique de sécurité par défaut et/ou des demandes de chacune des applications
- L'authenticité et l'intégrité du SSM doivent être vérifiées au démarrage du système et le SSM doit être protégé par le HSM durant tout le fonctionnement de la plate-forme

Démarrage sécurisé du SSM

- Le SoC intègre une petite ROM et une petite RAM interne
- La ROM contient le code de démarrage
  - Configure le HSM pour protéger en intégrité et éventuellement en confidentialité la zone mémoire où sera chargé le SSM
  - Charge le SSM (et éventuellement l'OS ou l'hyperviseur) depuis une mémoire flash vers la mémoire dans la zone protégée
  - Pendant le chargement du SSM, calcule son empreinte (haché)
  - Compare avec l'empreinte stockée dans une mémoire non volatile interne
- Mécanisme pour mettre à jour l'empreinte de référence de façon sécurisée sans possibilité de retour en arrière

Démarrage sécurisé

Plus généralement

- Principe général : Chaîne de confiance
  - L'étape n charge l'étape $n+1$ en mémoire
  - Durant le chargement, l'étape n vérifie l'authenticité de l'étape $n+1$
  - À la fin du chargement, si tout est correct, l'étape n transfère le contrôle à l'étape $n+1$
- Problèmes principaux
  - Comment vérifier l'authenticité d'une étape ?
  - Quelle est l'étape 0 et comment vérifier son authenticité ?
  - Comment permettre la mise à jour d'une étape et empêcher le retour en arrière (*downgrade*) ?
  - *Time-of-Check vs. Time-of-Use* : que se passe-t-il si un adversaire peut modifier l'étape $n+1$ en mémoire après son chargement et sa vérification mais avant son exécution ?

Démarrage sécurisé

Modèle d'attaque

- Les solutions techniques à mettre en œuvre dépendent du modèle d'attaque considéré
  - Changement du contenu du disque dur (OS + applications) $\rightsquigarrow$ trivial
  - Changement du contenu d'une flash (bootloader) $\rightsquigarrow$ facile
  - Espionnage des bus de communication (y compris mémoire) $\rightsquigarrow$ moyennement difficile
  - Attaque contre les composants (*micro-probing*, injection de fautes...) $\rightsquigarrow$ difficile

Démarrage sécurisé

Comment vérifier l'authenticité d'une étape ?

- Calcul d'une empreinte (généralement un résumé cryptographique) sur le code , les données et éventuellement la configuration utilisée
- Comparaison de cette empreinte avec une empreinte de référence
- Si l'adversaire peut espionner le bus mémoire, le calcul de l'empreinte doit être fait soit en utilisant une RAM interne, soit en utilisant une architecture protégeant l'intégrité des bus (ex. SecBus)
- Plusieurs options pour la comparaison de l'empreinte avec une référence...

Démarrage sécurisé

Comparaison avec une référence : options

- Empreinte de référence stockée dans une mémoire non volatile non accessible par l'adversaire (pour la première étape) : en flash ou interne au processeur (en fonction du modèle d'attaque)
- Empreinte de référence signée numériquement et fournie avec le code
  - La signature est vérifiée avec une clé publique (ou symétrique mais plus dangereux) stockée dans une mémoire non volatile (ROM ou flash) non accessible par l'adversaire
- Utilisation d'un tiers de confiance qui effectue la comparaison et renvoie le résultat
  - Nécessite l'utilisation d'un protocole permettant d'authentifier ce tiers, de garantir l'intégrité des échanges et de se prémunir contre le rejeu d'une transaction précédente
  - Exemples : carte à puce, serveur...

Démarrage sécurisé

Mise à jour et comment empêcher le retour en arrière ?

- Permettre la mise à jour d'une étape tout en empêchant le retour vers une ancienne version dans laquelle une faille de sécurité aurait été découverte
- Cas du stockage de l'empreinte de référence
  - Authentifier la mise à jour
  - Remplacer l'empreinte de référence par la nouvelle
    - Le remplacement doit être atomique (si interrompu, risque de verrouiller le système)
    - Il ne doit être possible qu'après authentification de la mise à jour

Démarrage sécurisé

Mise à jour et comment empêcher le retour en arrière ? (suite)

- Cas de la signature numérique
  - En apparence plus simple car la mise à jour est déjà signée correctement
  - Mais pour empêcher le retour en arrière, il faut tout de même prévoir un stockage non volatile contenant un numéro de mise à jour (ou un compteur monotone) avec les mêmes problématiques que précédemment
  - Possibilité d'utiliser des eFUSE (technologie développée par IBM) pour implémenter ce compteur de version monotone
- Cas du tiers de confiance : simple

Démarrage sécurisé

Étape 0 ?

- L'étape 0 est parfois appelée racine de confiance
- Doit se situer sur un support non accessible par l'adversaire
  - ROM interne si l'adversaire peut espionner les bus
  - ROM interne si l'adversaire peut modifier la flash
  - ROM interne ou flash si l'adversaire n'a accès qu'au disque dur
- Mise à jour probablement impossible donc la bonne conception de cette étape est fondamentale
  - Doit faire le minimum de chose
  - Doit utiliser des algorithmes robustes
  - Pour les systèmes les plus critiques, on peut envisager une preuve formelle du bon fonctionnement de cette étape

Démarrage sécurisé

Time-of-Check vs. Time-of-Use

- Si l'adversaire est capable de modifier les bus de communication (et notamment le bus mémoire), il est capable de modifier les données d'une étape après leur vérification mais avant (ou pendant) leur utilisation
- Pas de solution satisfaisante sauf se prémunir contre l'espionnage et la modification des bus (ex. architecture SecBus)
- Certaines solutions proposent d'utiliser du contrôle dynamique mais insuffisant (fournir la "bonne" valeur lors du contrôle et la valeur modifiée lors de l'utilisation)

Plan

Conclusion

- L'espionnage des bus de communication d'un système embarqué est souvent plus simple à réaliser que les attaques contre les composants eux-mêmes
- Tous ces bus peuvent être protégés en utilisant des solutions logicielles à l'exception notable du bus mémoire
- La protection du bus mémoire requiert du matériel dédié inclus dans le SoC et peut être coûteux en fonction du niveau de sécurité désiré (notamment la protection contre le rejeu)
- Néanmoins, avec la généralisation des nouvelles technologies comme l'eDRAM (*embedded* DRAM), SiP (*System-in-Package*), *packaging* 3D, etc. ces attaques vont peut être disparaître

Références I

[1] Intel(r) software guard extensions.
`https://software.intel.com/en-us/sgx`, November 2016.

[2] Robert M. Best.
Microprocessor for executing enciphered programs.
Technical Report US4168396, United States Patent, September 1979.

[3] Robert M. Best.
Preventing software piracy with crypto-microprocessors.
In IEEE *Spring* COMPCON*'80*, pages 466–469. IEEE Computer Society, February 1980.

[4] Robert M. Best.
Crypto microprocessor for executing enciphered programs.
Technical Report US4278837, United States Patent, July 1981.

[5] Robert M. Best.
Crypto microprocessor that executes enciphered programs.
Technical Report US4465901, United States Patent, August 1984.

[6] Dallas Semiconductor.
DS5002FP Secure Microprocessor Chip, March 2015.
`http://datasheets.maxim-ic.com/en/ds/DS5002FP.pdf`.

[7] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas.
Caches and hash trees for efficient memory integrity verification.
In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (*HPCA*'03)*, pages 295–306, February 2003.

Références II

[8] Tanguy Gilmont, Jean-Didier Legat, and Jean-Jacques Quisquater.
Hardware security for software privacy support.
IEE *Electronics Letters*, 35 :2096–2098, November 1999.

[9] Andrew Huang.
Keeping secrets in hardware : The microsoft xbox(tm) case study.
In *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 2002.

[10] Ronan Keryell.
CRYPTOPAGE-1 : vers la fin du piratage informatique ?
In *Symposium d'Architecture (SYMPA'6)*, pages 35–44, Besançon, June 2000.

[11] M. Kuhn.
The TRUSTNO1 cryptoprocessor concept.
Technical Report CS555, Purdue University, April 1997.

[12] Markus G. Kuhn.
Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP.
In IEEE *Transaction on Computers*, volume 47, pages 1153–1157. IEEE Computer Society, October 1998.

[13] Cédric Lauradoux and Ronan Keryell.
CRYPTOPAGE-2 : un processeur sécurisé contre le rejeu.
In *Symposium en Architecture et Adéquation Algorithme Architecture (SYMPAAA'2003)*, pages 314–321, La Colle sur Loup, France, October 2003.

Références III

[14] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz.
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*)*, pages 168–177, October 2000.

[15] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas.
AEGIS : Architecture for tamper-evident and tamper-resistant processing.
In *Proceedings of the 17th International Conference on Supercomputing (*ICS*'03)*, pages 160–171, June 2003.

[16] Xiaotong Zhuang, Tao Zhang, and Santosh Pande.
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*)*, pages 72–84. ACM Press, October 2004.