Une école de l'IMT

Support matériel pour la sécurité logicielle

Guillaume Duc
guillaume.duc@telecom-paristech.fr
2018–2019

Plan

Objectifs du cours

- Connaître les mécanismes de sécurité classiques offerts par le matériel
- Comprendre comment ces mécanismes sont utilisés pour garantir la sécurité de la partie logicielle
- Contexte : dans la suite on considérera par défaut le cas d'un ordinateur "classique"
  - La plupart des dispositifs mentionnés sont maintenant également disponibles dans la plupart des architectures pour systèmes embarqués

Plan

Plan

Anneaux de protection

Protection rings

- Concept introduit dans le matériel développé pour le système *Multics* (8 anneaux utilisés)
- Mécanisme matériel offrant une certaine protection des données contre des fautes ou des comportements malicieux du logiciel
- Différentes parties du logiciel (système d'exploitation, pilotes de périphériques, bibliothèques, applications, etc.) vont s'exécuter dans des anneaux différents avec des privilèges différents

Anneaux de protection

Protection rings

- Hiérarchie entre les anneaux
  - L'anneau le plus privilégié (généralement numéroté 0) a accès à toutes les fonctionnalités et les données du système
  - Chaque anneau à accès aux fonctionnalités et aux données des anneaux moins privilégiés
  - Par contre, dans un anneau donné, l'accès aux fonctionnalités et aux données de niveaux plus privilégiés doit passer par des interfaces bien spécifiées (appels systèmes...)

Anneaux de protection

Exemple architecture x86

- Présents dans l'architecture x86 depuis le processeur 80386 (1985)
- 4 niveaux de protections offerts lorsque le processeur est en mode *protégé* (par défaut le processeur démarre en mode *réel* par compatibilité)
  - Du *ring 0* le plus privilégié
  - au *ring 3* le moins privilégié
- Le niveau courant est stocké dans le registre interne CPL (*Current Privilege Level*)
  - Généralement, le CPL est égal au DPL (*Descriptor Privilege Level*) indiqué dans le descripteur du segment de code courant (registre CS)

Anneaux de protection

Exemple architecture x86

- Le CPL est utilisé pour effectuer des contrôles
  - Le chargement d'un descripteur de segment de données (DS, ES, FS, GS, SS) n'est possible que si le niveau de privilège indiqué dans le descripteur (DPL) est numériquement supérieur ou égal au CPL (moins privilégié)
  - Lors d'un saut vers un autre segment de code (*far* JMP ou CALL)
    - le DPL du segment cible doit être égal au CPL,
    - ou, si le bit *conforming* est positionné dans le descripteur de segment cible, le DPL du segment cible peut être inférieur ou égal au CPL (dans ce cas, ce dernier ne change pas lors du saut). Ce cas est utilisé pour des segments contenant du code devant être partagé entre plusieurs niveaux (bibliothèques)

Anneaux de protection

Exemple architecture x86

- Le CPL est utilisé pour effectuer des contrôles (suite)
  - Lors de l'exécution de certaines instructions affectant les mécanismes de protection (instructions dites *privilégiées*) pour lesquelles le CPL doit être à 0
    - Exemples : Chargement du pointeur vers les tables de descripteurs de segments (LDT et GDT), table des vecteurs d'interruption (IDT), modification des registres de contrôle, arrêt du processeur...
  - Lors de l'exécution d'instructions concernant les I/O (instructions dites *sensibles*) pour lesquelles le CPL doit être inférieur ou égal au *I/O Privilege Level* (IOPL) défini dans le registre de drapeaux (*flags*)

Anneaux de protection

Exemple architecture x86

- Changement de niveau CPL
  - Principalement lors d'une interruption ou d'une exception (en provenance du matériel, problème d'exécution, debug, ou interruption logicielle)
  - D'autres mécanismes existent (notamment lors d'un CALL sur une *call gate*) mais ne sont plus couramment utilisés

Anneaux de protection

Exemple architecture x86

- Ces dernières années, le modèle d'anneaux du x86 s'est complexifié avec l'introduction
  - De la virtualisation (considéré comme un *ring -1*)
  - Du *System Management Mode* (considéré comme un *ring -2*)

Anneaux de protection

Exemple architecture `ARMv7-A`

- Dans l'architecture `ARMv7-A`, trois niveaux de privilèges existent (deux dans les architectures antérieures sans virtualisation) dans la zone non sécurisée (hors *TrustZone*) liés aux modes de fonctionnement du processeur
  - PL0 : Niveau le moins privilégié, typiquement pour les applications (mode `User`)
  - PL1 : Niveau plus privilégié, typiquement pour le système d'exploitation (modes `FIQ`, `IRQ`, `Supervisor`, `Abort`, `Undefined` et `System`)
  - PL2 : Niveau le plus privilégié (mode `Hyp`)

Anneaux de protection

Exemple architecture `ARMv7-A`

- Passage de PL0 vers PL1
  - Via une interruption logicielle (instructions `SWI` (*Software Interrupt*) ou `SVC` (*Supervisor Call*)) : basculement en mode `Supervisor`
  - Via une interruption matérielle : basculement en mode `FIQ` ou `IRQ`
  - Via une exception : basculement en mode `Abort` ou `Undefined`
- Passage de PL1 vers PL0
  - Explicitement en écrivant dans le registre `CPSR` (*Current Processor Status Register*) pour changer de mode (et donc de niveau de privilège)
  - Implicitement via la restauration du `CPSR` lors d'un retour d'interruption vers le mode `User`

Plan

Introduction

- Mécanisme introduit dans les années 1960 (monde des *mainframes*) mais disponible sur les PC depuis le 80386 d'Intel
- Avantages
  - Isolation entre les espaces mémoires de différentes applications
  - Partage explicite de certaines zones mémoires (bibliothèques de code, données partagées...)
  - Pagination (permet aux applications de disposer de plus de mémoire qu'il en existe physiquement)

Principes

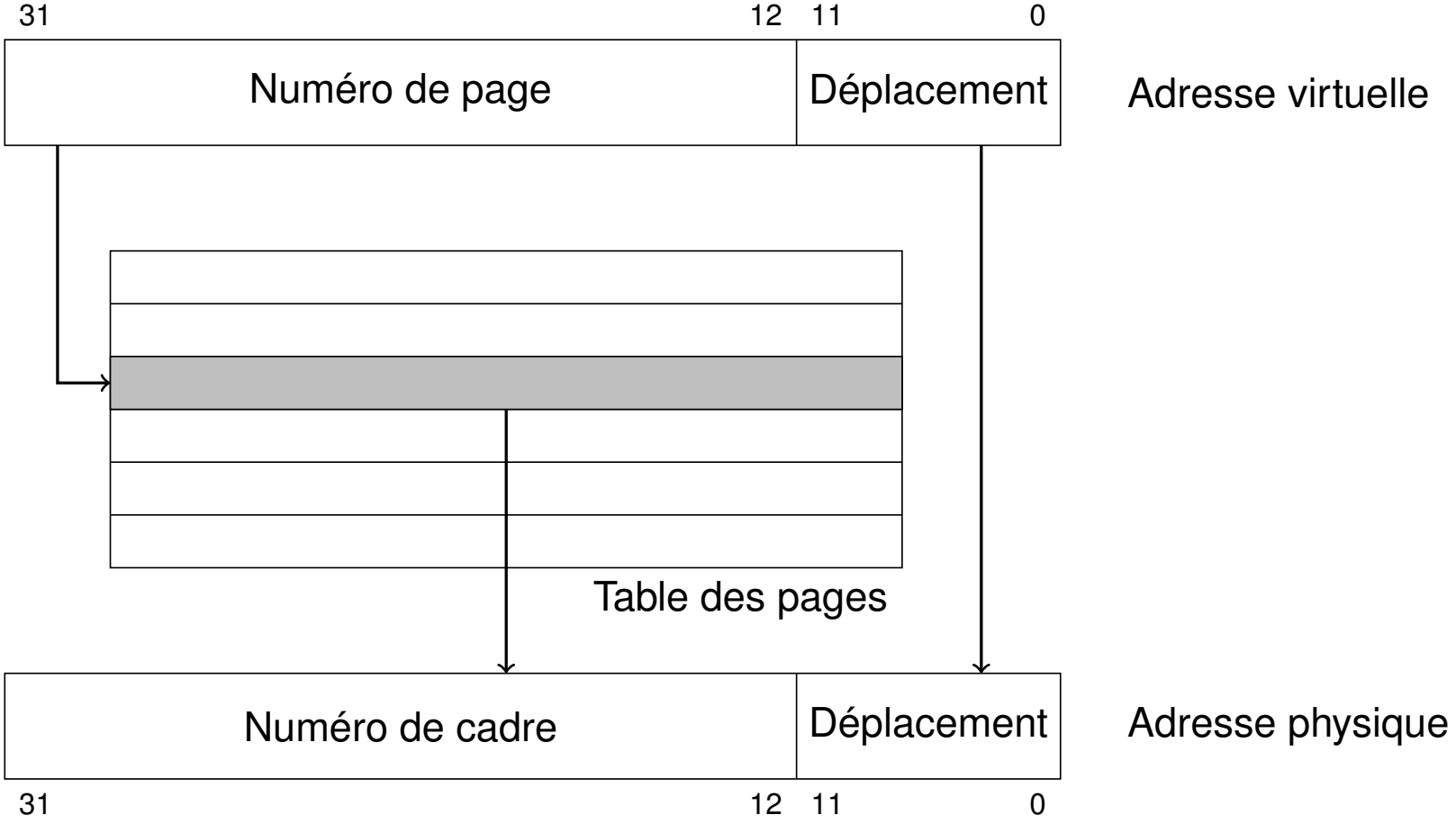
- Les applications ne manipulent pas directement des adresses *physiques* (adresses présentées aux puces de RAM) mais des adresses *virtuelles*
- Un mécanisme matériel, la MMU (*Memory Management Unit*), le plus souvent intégrée aux processeurs, se charge de faire la conversion entre les adresses virtuelles et les adresses physiques
- Cette conversion est effectuée en se basant sur une table de conversion, nommée la table des pages (*page table*), le plus souvent mise en place par le système d'exploitation

Fonctionnement

- Pour des raisons pratiques, la traduction n'est pas réalisée avec la granularité d'une adresse virtuelle (il y a 2^{32} adresses virtuelles dans le cas d'un processeur 32 bits, donc une table des pages nécessiterait de stocker les 2^{32} adresses physiques correspondantes ; chaque adresse étant codée sur 32 bits, la table ferait 16 Gio...)
- L'espace d'adressage virtuel ainsi que l'espace d'adressage physique sont découpés en *pages* (au minimum 4 kio sur `x86`)
- La conversion fait correspondre un numéro de cadre physique à un numéro de page virtuelle, le déplacement au sein de la page physique étant égal au déplacement au sein de la page virtuelle

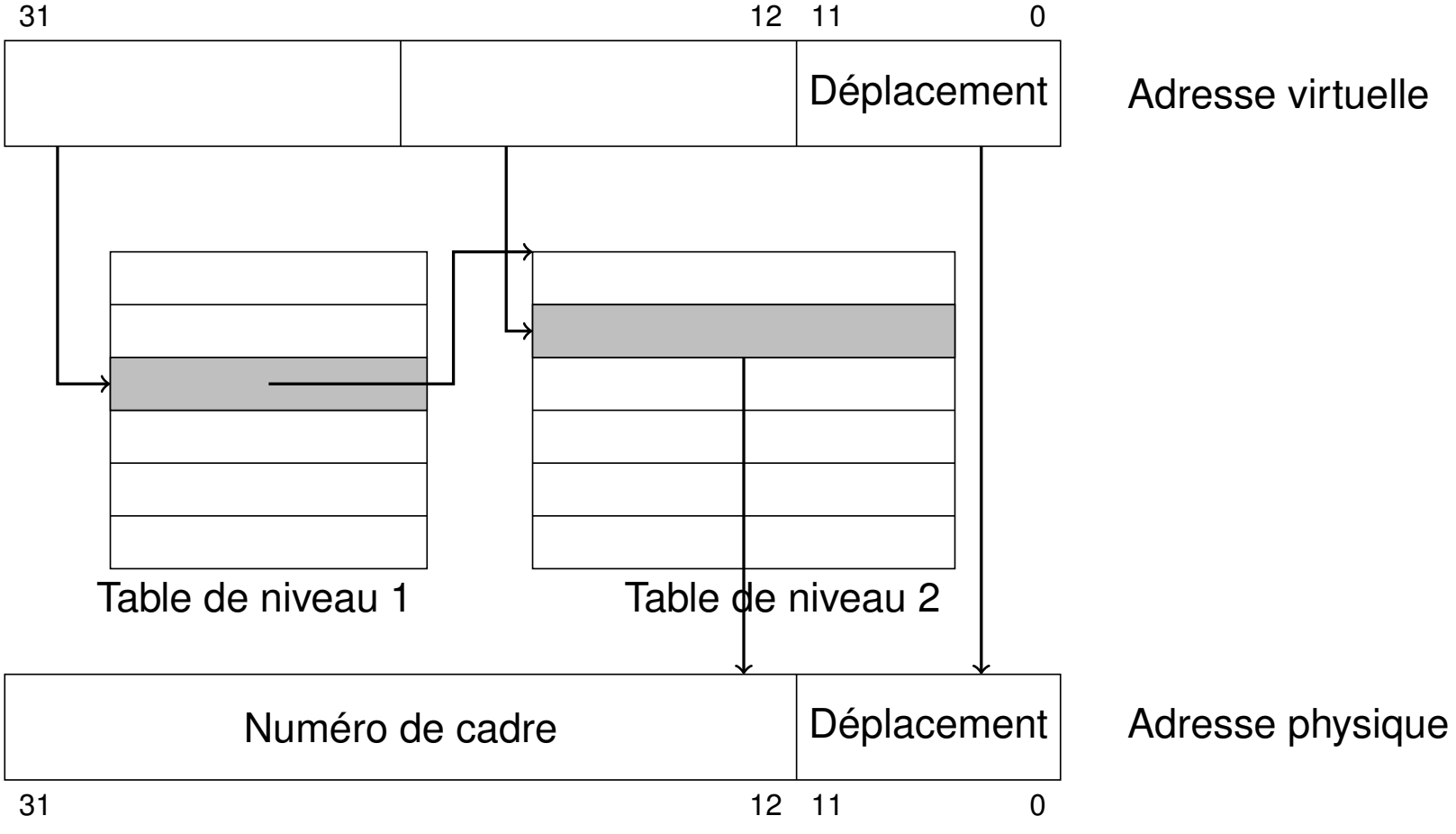
Fonctionnement

Exemple : adresses 32 bits, pages de 4 kio, 1 niveau de traduction



Fonctionnement

Exemple : adresses 32 bits, pages de 4 kio, 2 niveaux de traduction



Fonctionnement

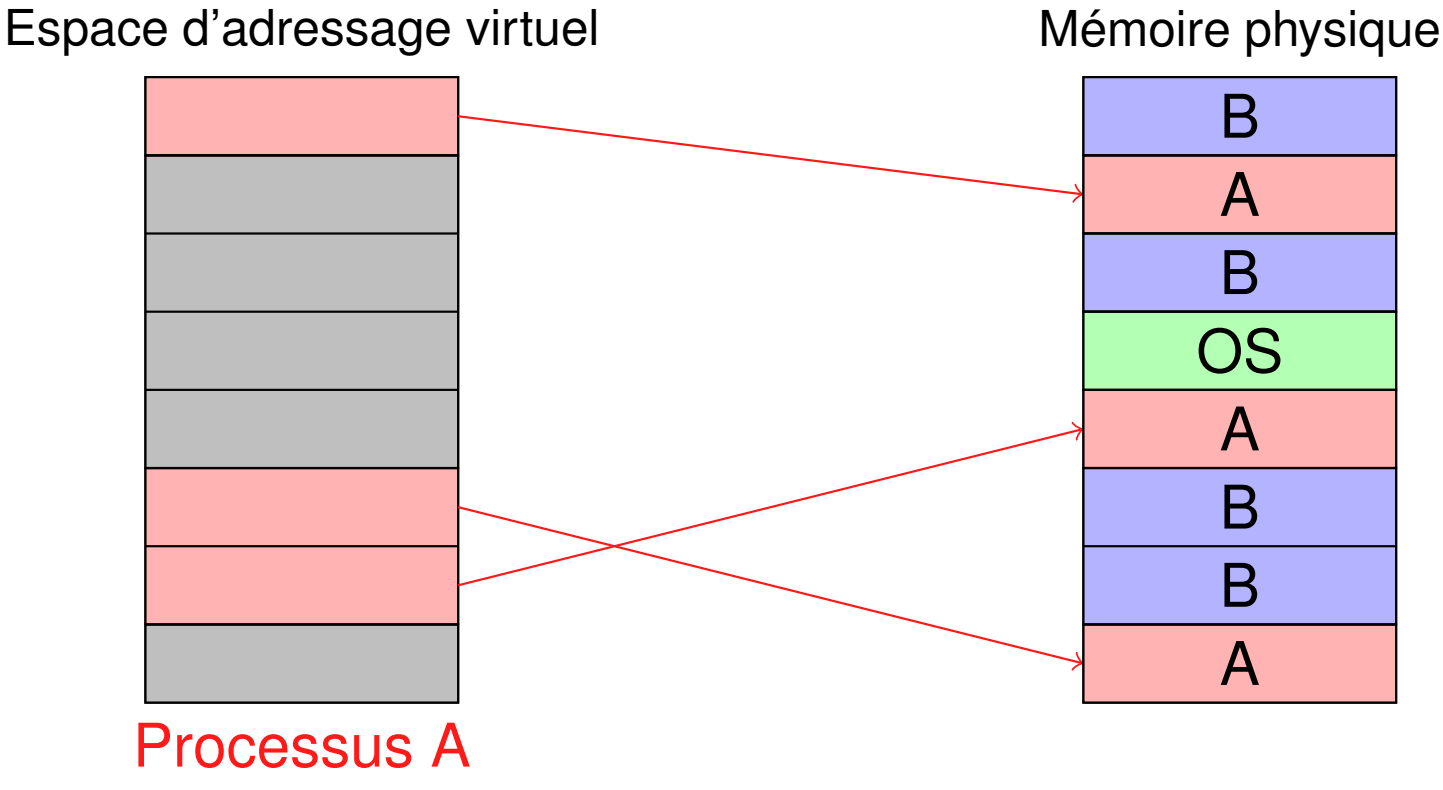
- Utilisation de plusieurs niveaux de tables de pages pour gérer plus efficacement les larges zones vides dans l'espace d'adressage virtuel d'un processus
- La MMU dispose d'un cache spécialisé (TLB, *Translation Lookaside Buffer*) destiné à conserver les traductions (page virtuelle, cadre physique) récemment utilisées
- En cas de défaut de TLB, le parcours de la table de pages (ou des différents niveaux de table) pour rechercher la traduction peut être effectué
  - Soit matériellement : la MMU connaît l'adresse de la table de premier niveau et va la parcourir elle-même (exemple architecture `x86`)
  - Soit logiciellement : la MMU déclenche une exception et c'est au système d'exploitation de charger explicitement une entrée du TLB avec la traduction demandée (exemple architecture MIPS)

Fonctionnement

- Dans une entrée de la table de pages (entrée de page, *page entry*) sont également stockées les informations suivantes
  - Validité (l'entrée est-elle valide ?)
  - Informations d'accès (les données de la page ont-elles été accédées et/ou modifiées ?)
  - Informations de protection
    - Page en lecture seule ou lecture/écriture
    - Accès en mode utilisateur ou en mode superviseur uniquement
    - Page contenant du code exécutable ou non (bit NX)

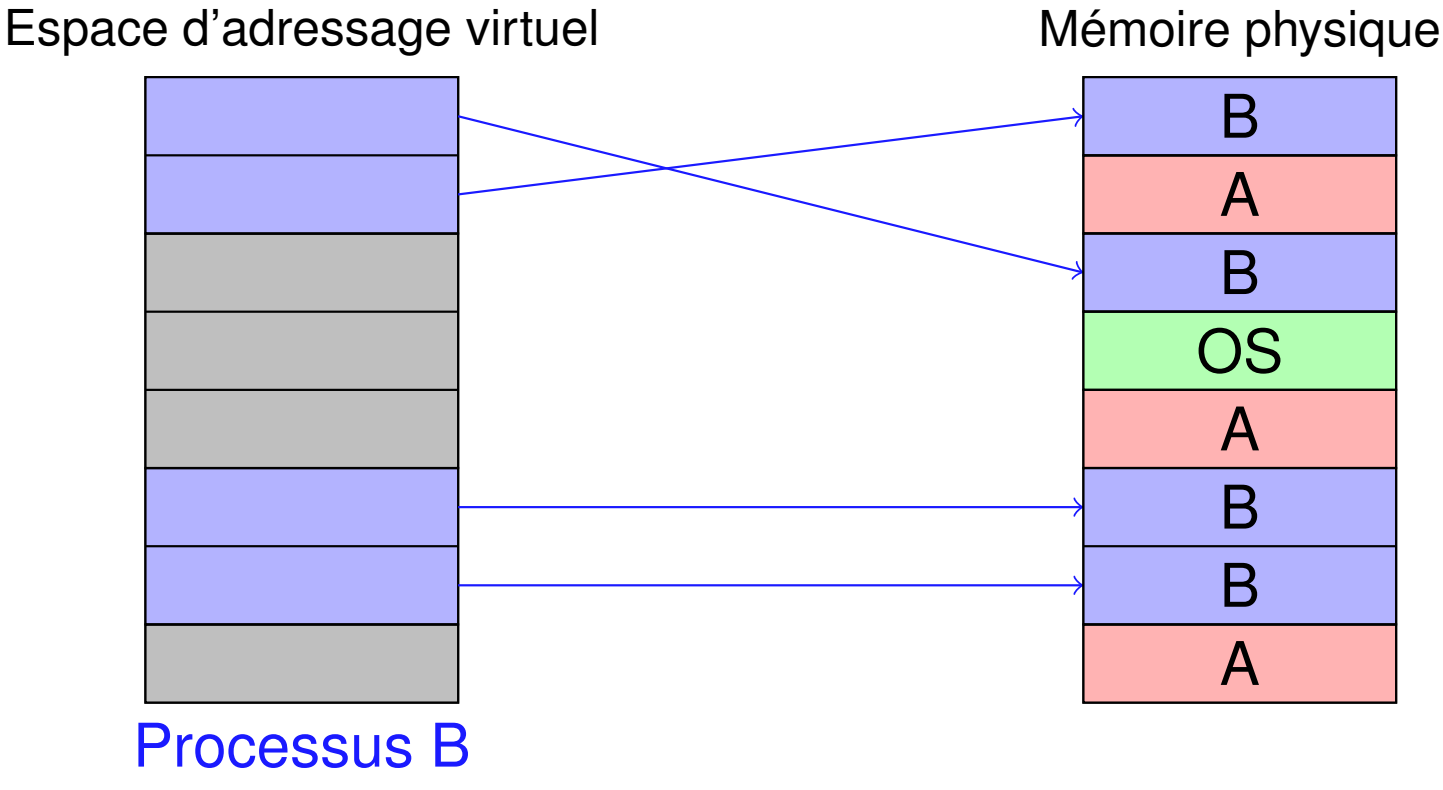
Utilisation

Flat memory model



Utilisation

Flat memory model



Utilisation

- Le système d'exploitation maintient une table de pages par processus
- Lorsque le SE bascule d'un processus à un autre, il configure la MMU pour utiliser la table de pages du nouveau processus
- La modification du registre de la MMU contenant le pointeur vers le premier niveau de la table de pages est une action qui est privilégiée (qui ne peut donc être réalisée que par le SE)
  - Lien avec les anneaux de protection
- Un processus ne peut donc pas modifier directement sa table de pages

Utilisation

- Lorsqu'un processus accède à une adresse virtuelle pour laquelle l'entrée correspondante dans la table des pages n'est pas valide (drapeau valide non positionné) ou si les informations de protection ne sont pas correctes (lecture seule pour un accès en écriture par exemple), la MMU déclenche une exception
- Le système d'exploitation reprend alors la main et peut
  - Tuer le processus en cas d'accès illégal (accès à une page mémoire non allouée, violation des attributs de protection...)
  - Dans certain cas, l'exécution peut reprendre après une action du système d'exploitation (restauration de la page demandée qui avait été évincée de la mémoire faute de place, mécanisme de *copy-on-write*...)

Utilisation

Isolation des processus

- Un processus ne peut donc pas accéder à une adresse physique si une traduction n'est pas explicitement mise en place par le système d'exploitation
- Donc le SE peut s'assurer qu'un processus ne peut pas accéder à des zones mémoires physiques appartenant à d'autres processus ou au SE lui-même

Utilisation

Pagination / *Swap*

- Si la mémoire vient à manquer, le système d'exploitation peut déplacer le contenu de certains cadres de mémoire physique vers un support de stockage de masse (disque dur) et marquer les pages de mémoire virtuelle correspondantes comme invalides
- Si un des processus dont la mémoire a été évincée essaie d'y accéder, la MMU déclenchera une exception
- Le SE reprend la main et voit que l'adresse demandée existe bien mais a été déplacée vers le disque
- Il peut remettre les données dans un cadre de mémoire physique et reconfigurer l'entrée de page correspondante
- Il relance alors l'instruction fautive du processus qui peut maintenant réussir

Exemple architecture x86

Depuis le 80386, schéma général

- *Adresse logique* (selecteur de segment + déplacement) transformée en *adresse linéaire* (mécanisme de segmentation)
  - En pratique, adresse linéaire = adresse logique (*Flat memory model*) dans la plupart des cas actuellement
- Adresse linéaire traduite en adresse physique par la MMU via une traduction à deux niveaux
  - Registre `CR3` (accès privilégié) pointe vers un répertoire de pages (*page directory*) : table de 1024 entrées de 32 bits
  - Bits 31–22 de l'adresse linéaire : index dans le répertoire de pages qui donne l'adresse de la table de pages à utiliser (table de 1024 entrées de page, *page entries*)
  - Bits 21–12 de l'adresse linéaire : index de l'entrée de page dans la table de pages
  - Bits 11–0 : Déplacement dans la page de 4 kio

Exemple architecture x86

Depuis le 80386, structure d'une entrée de page

- Bits 31–12 : bits 31 à 12 de l'adresse du cadre de page
- Bit 6 : page modifiée (*dirty*)
- Bit 5 : page accédée
- Bit 2 : accès réservé au mode superviseur (`CPL` < 3) ou autorisé en mode utilisateur
- Bit 1 : lecture seule ou lecture/écriture
- Bit 0 : validité de l'entrée

Exemple architecture x86

Extensions

- *Page Size Extension* (Pentium)
  - Une entrée dans le répertoire de pages peut directement correspondre à une entrée pour une page de 4 Mio (bit 7 (PS) à 1)
  - Avantage : n'occupe qu'une seule entrée du TLB (au lieu de 1024)
- *Physical Address Extension* (Pentium Pro)
  - Volonté d'utiliser des adresses physiques plus longues que 32 bits (pour pouvoir gérer plus de 4 Gio de RAM)
  - Passage d'entrées de pages de 32 à 64 bits pour pouvoir y stocker plus de bits pour l'adresse physique
  - Ajout d'un nouveau niveau de hiérarchie (pour conserver la taille d'une table de pages ou d'un répertoire de pages)

Exemple architecture x86

Extensions

- NX bit (*No eXecute*), Intel=XD bit, AMD=*Enhanced Virus Protection*
  - Depuis le 80286, possibilité d'interdire l'exécution seulement au niveau d'un segment, donc impossible dans le modèle *Flat Segment*
  - Ajout d'un bit au niveau de l'entrée de page pour indiquer que la page ne contient pas de code exécutable
  - Introduit tout d'abord par AMD dans les processeurs AMD64
  - Puis repris par Intel à partir du Pentium 4
  - Utile par exemple pour rendre la pile non exécutable (plus possibilité de faire de l'injection de code en pile)

Plan

Unité de protection mémoire

Memory Protection Unit (MPU)

- Version “simplifiée” d’une MMU pour les microcontrôleurs et petits microprocesseurs (exemple : ARM *Cortex-M*)
- Permet uniquement du contrôle d’accès (lecture, écriture, exécution en mode utilisateur ou superviseur, et éventuellement des informations liées à la gestion du cache) sur des zones mémoires en nombre limité
- Pas de traduction d’adresses (en général)

Unité de protection mémoire

Exemple : Cortex-M

- La MPU peut protéger un nombre fixe de régions (8 en général) de taille variable (32 octets à 4 Gio)
- Quelques contraintes sur les régions (leur taille est une puissance de 2 et elles doivent être alignées sur un multiple de leur taille)
- Pour chaque région
  - Actif/Inactif
  - *Execute Never*
  - Permissions d'accès (aucun, lecture seule, lecture/écriture, pour le mode privilégié et non privilégié)
  - Type de la mémoire (cachable, partageable...)

Plan

Problématique

Périphériques & DMA

- La plupart des périphériques peuvent accéder directement à la mémoire centrale en utilisant des adresses physiques via notamment le mécanisme de DMA (*Direct Memory Access*)
- Cela permet notamment aux périphériques de transférer des données vers la mémoire, sans avoir à monopoliser le processeur

Problématique

Attaque DMA

- Néanmoins, aucun contrôle n'est effectué et donc un périphérique malicieux (*firmware* corrompu) ou endommagé peut écrire ou lire n'importe où en mémoire, y compris dans les zones réservées au système d'exploitation, etc.
- Problème exacerbé dans le cas de la virtualisation puisqu'il empêche de fait l'utilisation directe d'un périphérique par une machine virtuelle (qui pourrait alors via ce dernier accéder aux données des autres machines virtuelles en RAM)

IO-MMU

- Dispositif, similaire à la MMU, permettant de traduire les adresses issues des périphériques (capables de faire du DMA) en adresses physiques pour la mémoire
- Un périphérique ne peut donc plus qu'accéder à des zones mémoires pour lesquelles le système d'exploitation ou l'hyperviseur à mis en place des traductions d'adresse
- Exemples de mise en œuvre
  - *Graphics Address Remapping Table* (GART) utilisée pour les cartes graphiques AGP et PCIe
  - Intel *Virtualization Technology for Directed I/O* (VT-d)
  - AMD *I/O Virtualization Technology*

IO-MMU

Avantages

- Protection mémoire contre les attaques DMA ou les périphériques défectueux
  - Notamment dans le cadre de la virtualisation pour laisser une machine virtuelle utiliser directement un périphérique
- Possibilité pour les périphériques d'utiliser plus de mémoire physique (PAE)
- Allocations de larges zones mémoires virtuellement contiguës

Plan

Plan

ARM TrustZone

- Ensemble de technologies permettant d'implémenter un environnement d'exécution de confiance (*Trusted Execution Environment*, TEE)
  - Modification du cœur (ARM *Security Extension*)
  - Extension du protocole AMBA3 AXI
  - IPs *ad-hoc*
  - Couche logicielle *ad-hoc*
  - Debug sécurisé
- Les données et les applications s'exécutant dans le TEE sont protégées contre une vaste gamme d'attaques
  - Attaques logicielles (applications, OS ou hyperviseur corrompu)
  - Attaques DMA, IP malicieuses...
- Disponible depuis l'architecture ARMv6KZ

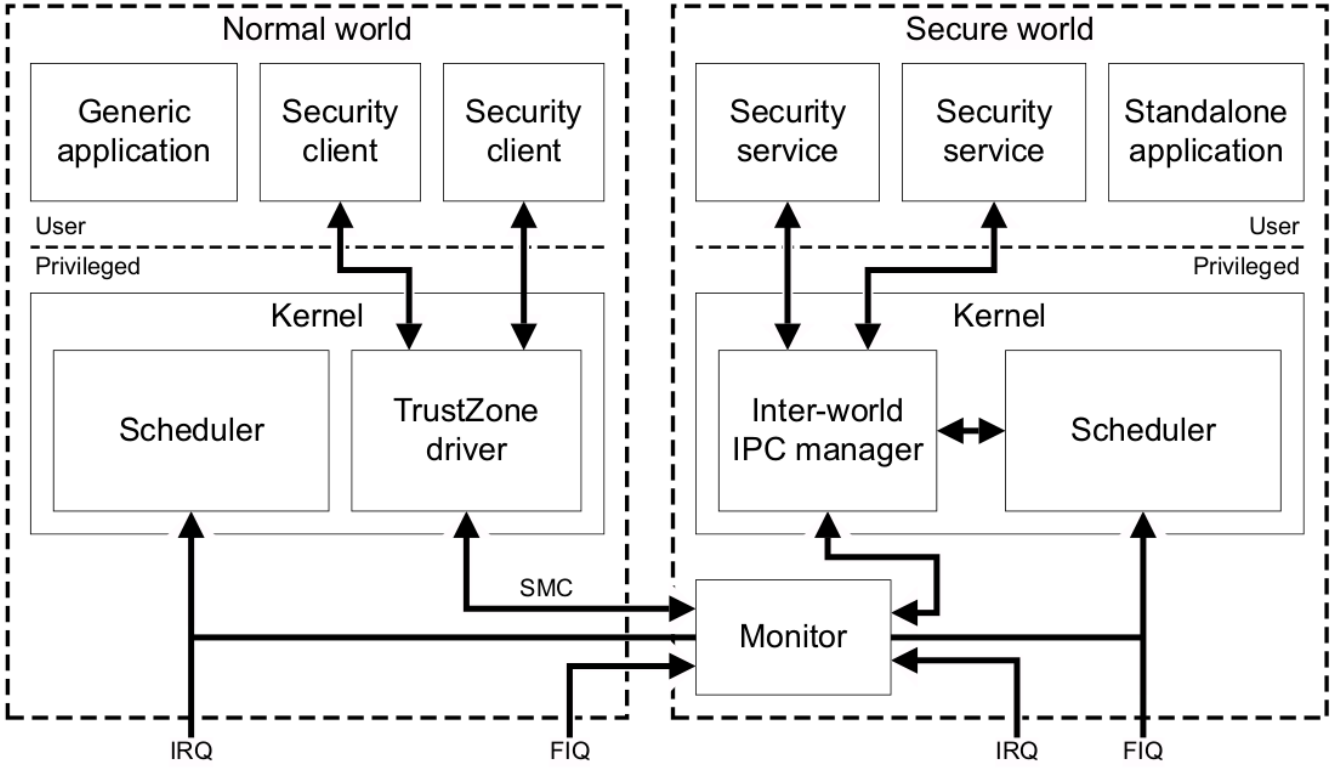
Implémentation matérielle

- Création de deux mondes : sécurisé et non sécurisé
- Le cœur du processeur est modifié pour prendre en compte ces deux mondes
- Orthogonal aux autres mécanismes du processeur (il y a également plusieurs niveaux de privilèges dans le monde sécurisé : OS sécurisé et applications sécurisées)
- Passages du monde non sécurisé au monde sécurisé bien définis
  - Instructions *Secure Monitor Call* / *Secure Gateway*
  - Certaines exceptions ou interruptions (configurables)
- Chaque ligne de cache est marquée
- Chaque monde a une table de pages propre

Implémentation matérielle

- Le protocole de bus AMBA3 est modifié pour indiquer si une transaction a été émise depuis le monde sécurisé ou non sécurisé
- Le contrôleur de bus peut ainsi empêcher le monde non sécurisé d'accéder à des ressources matérielles qui ont été configurées pour appartenir au monde sécurisé

Implémentation logicielle



Source : ARM

- Exemple de moniteur fourni par ARM : `https://github.com/ARM-software/arm-trusted-firmware`

Trustzone pour Cortex-M

Introduction

- Deux variantes de Trustzone : une pour les Cortex-A (processeurs applicatifs) et une pour les Cortex-M (microcontrôleurs)
- Disponible dans les Cortex-M23 et M33
- Plus de documentation disponible librement
- Variante réduite de Trustzone pour Cortex-A

Trustzone pour Cortex-M

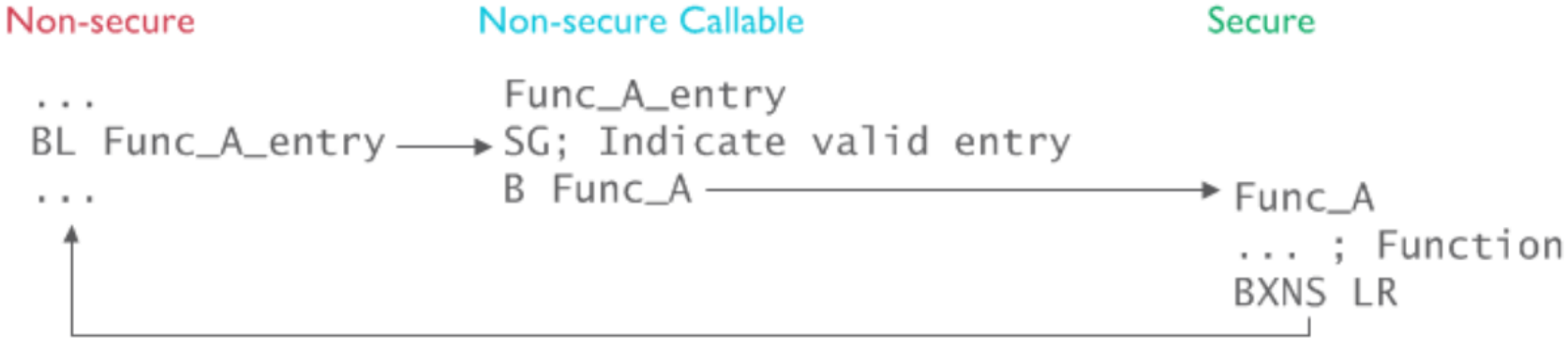
Partitionnement de la mémoire

- L'espace d'adressage est divisé en plusieurs zones
  - *Secure (S)* : mémoire ou périphérique accessible uniquement depuis un logiciel s'exécutant en mode sécurisé ou depuis un périphérique configuré comme sécurisé
  - *Non-secure callable (NSC)* : seule zone mémoire où l'instruction `SG` (*Secure Gateway*) est autorisée. Typiquement ne contiendra que des points d'entrées de fonctions sécurisées
  - *Non-secure (NS)* : mémoire ou périphérique accessible par tout logiciel ou périphérique
- Configuration statique ou dynamique

Trustzone pour Cortex-M

Passage d'un monde à l'autre

- Appel d'une fonction sécurisée depuis le monde non sécurisé

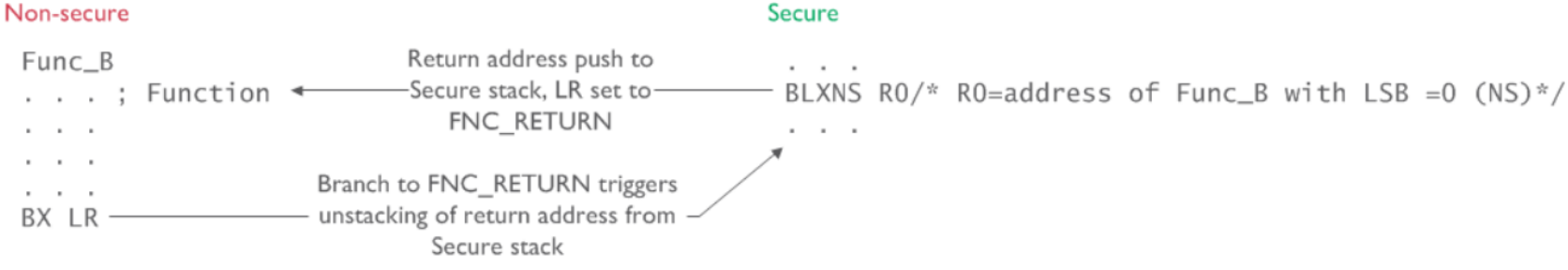


Source : TrustZone technology for the ARMv8-M architecture
(`https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf`)

Trustzone pour Cortex-M

Passage d'un monde à l'autre

- Appel d'une fonction non sécurisée depuis le monde sécurisé



Source : TrustZone technology for the ARMv8-M architecture
(`https://static.docs.arm.com/100690/0200/armv8m_trustzone_technology_100690_0200.pdf`)

Trustzone pour Cortex-M

Passage d'un monde à l'autre

- Via des interruptions
- Chaque interruption peut être configurée pour être traitée dans le monde sécurisé ou non sécurisé via le registre `NVIC_ITNS` (seulement modifiable en mode sécurisé)
- En cas de passage du monde sécurisé vers le monde non sécurisé à cause d'une interruption, le contenu de l'ensemble des registres est sauvegardé dans la pile sécurisée et effacé

Trustzone

Conclusion

- Vise principalement les attaques logicielles
- Permet d'exécuter du code sensible (exemple : application bancaire) à côté d'applications non dignes de confiance (OS et applications non contrôlées)
- Pas de prise en compte des attaques matérielles

Plan

Présentation

- Extensions développées par Intel pour permettre de garantir l'intégrité et la confidentialité d'opérations sensibles sur un ordinateur dont le code privilégié (système d'exploitation, hyperviseur...) est potentiellement malicieux
- Nouvelle solution au problème de l'exécution sécurisée à distance
- Disponibles depuis l'architecture *SkyLake* (2015)
- Analyse détaillée dans [1] (lecture très intéressante)

Fonctionnement

Protection de la mémoire

- Un processus qui a besoin d'exécution sécurisée va s'exécuter dans une enclave
- Une région particulière de la mémoire (*Processor Reserved Memory*, PRM) est dédiée à ces enclaves
- Le processeur protège cette zone contre tout accès qui ne proviendrait pas d'une enclave (noyau, hyperviseur, SMM, accès DMA en provenance d'un périphérique...)
- Les données qui sont situées dans cette zone sont aussi chiffrées et protégées en intégrité à la volée par un module matériel dédié sur le processeur
  - Empêche les attaques physiques par espionnage des bus du processeur ou du contenu de la RAM

Fonctionnement (suite)

Protection de la mémoire

- Cette zone est divisée en page (*Enclave Page Cache*, EPC)
- Chaque EPC peut contenir du code ou des données d'une enclave
- L'affectation des EPC à une enclave est gérée par le logiciel (non digne de confiance) et contrôlée par la processeur pour garantir qu'une page n'est allouée qu'à une enclave et une seule
- Le système d'exploitation peut evincer des pages EPC vers la mémoire en utilisant des instructions dédiées
  - Le contenu de ces pages est protégé (confidentialité et intégrité) afin de ne pas pouvoir être altéré

Fonctionnement (suite)

Initialisation d'une enclave

- Le code et les données initiales de l'enclave sont chargés par un logiciel non digne de confiance (système d'exploitation par exemple)
  - Grâce à des instructions spécifiques permettant de copier des données depuis la mémoire (hors PRM) vers des pages EPC
  - Le code et les données initiales sont en clair et ne peuvent donc contenir de secrets
  - Pendant le chargement, un condensé cryptographique est calculé sur les données chargées, ce qui permettra d'attester du code exécuté
- À la fin du chargement, la procédure est désactivée ce qui empêche la modification du code et des données de l'enclave par du logiciel non digne de confiance

Fonctionnement (suite)

Exécution dans une enclave

- L'exécution du code d'une enclave ne peut être démarré que par une instruction spéciale
- Le code est exécuté en mode protégé, dans le niveau de privilèges 3 et avec les traductions d'adresse (MMU) mises en place par le système d'exploitation et éventuellement l'hyperviseur
- Lors d'une interruption du flot d'exécution d'une enclave (interruption, exception, etc.), le processeur sauvegarde l'état du processeur dans une zone dédiée au sein de l'enclave et efface les registres du processeur avant de transférer le contrôle
  - Le système d'exploitation n'a ainsi pas accès au contexte matériel de l'enclave lors de l'interruption

Fonctionnement (suite)

Attestation

- Mécanisme d'attestation à distance
- Permet de prouver à distance qu'un message a bien été émis par un code donnée, s'exécutant dans une enclave sur un processeur capable de la sécuriser

Plan

Plan

Bugs matériels

- Les processeurs actuels comportent plusieurs centaines de millions de transistors et incluent de nombreux mécanismes pour accélérer les traitement
  - Exécution out-of-order
  - Prédiction de branchement
  - Multi-threading
  - Renommage de registre...
- Tout comme les logiciels, ils ne sont pas à l'abri de bugs, pouvant remettre en cause la sécurité

Bugs processeur

- Le bug le plus connu est celui de la division fautive du Pentium découvert en 1994
- Autre bug connu du Pentium : le F0 0F
  - L'instruction `lock cmpxchg8b eax` (en code machine `f0 0f c7 c8`) qui devrait normalement lever une exception fait planter le processeur qui doit alors être reinitialisé
  - Comme cette instruction peut être exécutée sans privilèges, cela crée un déni de service (un bug similaire existe dans les processeurs Cyrix)

Bugs processeur

- Les bugs des processeurs sont souvent inconnus du grand public
- Mais, s'ils touchent des fonctionnalités liées à la protection mémoire, ils peuvent avoir un impact important sur la sécurité
- Et quid des bugs qui n'en seraient pas (portes dérobées) ?
- Il est assez facile d'introduire une porte dérobée dans un processeur... (cf. [2])

Plan

Introduction

- Du fait de leur évolution ces 30 dernières années, les processeurs x86 comportent de nombreux modes de fonctionnement
- Principalement pour garder la compatibilité ascendante avec le 8088...
- Mais certains de ces modes sont méconnus et peuvent poser des problèmes de sécurité importants

System Management Mode

- Mode spécial des processeurs x86 utilisé principalement pour gérer la carte mère, l'alimentation et le contrôle de la température (ventilateurs, etc.)
- Mode 16 bits
  - Accès libre à l'intégralité de la mémoire
  - Aucun mécanisme de protection mémoire (ni segmentation ni pagination)
  - Accès libre aux IO
- Durant l'exécution en mode SMM, l'OS est suspendu
  - L'OS n'est pas informé de cette interruption
  - Il ne peut pas faire respecter sa politique de sécurité
- $\rightsquigarrow$ Le code s'exécutant en mode SMM a le contrôle total sur la machine

System Management Mode

- Le processeur entre en mode SMM en recevant une System Management Interrupt (interruption matérielle générée par le chipset, peut être déclenchée par une instruction d'IO)
- Le code du gestionnaire d'interruption (qui donc s'exécute en mode SMM) est situé dans une zone de la RAM appelée SMRAM
- Initialement, cette zone mémoire n'était pas particulièrement protégée...
- Actuellement, le chipset protège l'accès à cette zone mais cette protection n'est pas parfaite et des failles ont été trouvées par le passé [3]

Plan

Attaque *Meltdown* [6]

- Exécution des instructions dans le désordre (*out-of-order*)
  - Les processeurs disposent de plusieurs unités de calcul et de traitement (plusieurs ALU...)
  - Afin de maximiser l'utilisation de ces ressources, les instructions sont réordonnées (en conservant les dépendances éventuelles)
- Exécution spéculative
  - Des instructions peuvent être exécutées avant d'être certain qu'elles doivent l'être (i.e. après un saut conditionnel, avant d'avoir vérifié que les instructions précédentes ne génèrent pas d'exceptions...)
  - Si elles ne devaient pas l'être, leurs résultats sont annulés et aucun effet n'est visible par le logiciel
  - *Néanmoins, l'exécution de ces instructions peut laisser des traces au niveau micro-architecture qui peuvent être détectées*

Attaque *Meltdown* [6]

Exemple simple

- Considérons le code suivant (exemple tiré de [6]) :

```
exception();
accès mémoire(tableau[donnée * 4096]);
```

- Normalement la seconde instruction n'est pas exécutée
- Mais à cause de l'exécution spéculative dans le désordre, elle peut être exécutée avant que le processeur ne s'aperçoive qu'il ne devait pas le faire
- L'effet de cette instruction sera annulé (registre de destination restauré) mais elle aura eu un effet de bord sur le cache

Attaque *Meltdown* [6]

Exemple simple, suite

- En effet, l'accès mémoire va placer la ligne située à l'adresse `tableau[donnée * 4096]` dans le cache du processeur
- Il est possible pour un autre processus, via une attaque par canal auxiliaire contre le cache, de détecter ce chargement et de retrouver l'adresse accédée (et donc ici, la valeur de `donnée`)

Attaque *Meltdown* [6]

Application à la lecture de toute la mémoire

- Pour des questions de performance, dans de nombreux systèmes d'exploitation, le code et les données du noyau sont mappés dans l'espace d'adressage de chaque processus
- De même pour l'intégralité de la mémoire physique (sur les systèmes 64 bits)
- Ces adresses sont simplement marquées comme non accessibles depuis l'espace utilisateur au niveau de la MMU
- Donc si un processus essaie d'accéder à ces adresses, la MMU va déclencher une exception et le processus va être tué par le système d'exploitation

Attaque *Meltdown* [6]

Application à la lecture de toute la mémoire

- Exemple, cartographie mémoire sous Linux sur architecture x86_64 (cf. `Documentation/x86/x86_64/mm.txt`)

```
0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
hole caused by [47:63] sign extension
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffbffffffffff (=44 bits) kasan shadow memory (16TB)
... unused hole ...
fffffe0000000000 - fffffe7fffffffff (=39 bits) LDT remap for PTI
fffffe8000000000 - fffffeffffffffff (=39 bits) cpu_entry_area mapping
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffffffffffff (=64 GB) EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffff9fffffff (=512 MB) kernel text mapping, from phys 0
ffffffffa0000000 - (fixmap start) ( 1526 MB) module mapping space (variable)
(fixmap start) - ffffffffff5fffff kernel-internal fixmap range
ffffffffff600000 - ffffffffff600fff (=4 kB) legacy vsyscall ABI
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

Attaque *Meltdown* [6]

Application à la lecture de toute la mémoire

```
; rcx = kernel address
; rbx = probe array
retry:
  mov al, byte [rcx]   ; RAX[7:0] = RAM[RCX] -> Exception
  shl rax, 0xc         ; RAX = RAX * 4096
  jz retry             ; if RAX == 0 goto retry
  mov rbx, qword [rbx + rax] ; RAM[RBX+RAX] = RBX
```

Attaque *Meltdown* [6]

Application à la lecture de toute la mémoire

- Pendant que la première instruction s'exécute, les trois instructions suivantes (qui en dépendent) sont mises en attente
- Dès que le résultat de la lecture est disponible, ces trois instructions commencent à s'exécuter
- La première instruction déclenche une exception (accès invalide à une adresse noyau depuis l'espace utilisateur)
- Si la dernière instruction a le temps de commencer à s'exécuter, la ligne dont l'adresse est `RBX+RAX` est chargée dans le cache du processeur
- Or cette adresse dépend de la donnée lue par la première instruction

Attaque *Meltdown* [6]

Application à la lecture de toute la mémoire

- Certes, le processus est tué dans l'opération (il est possible sur certaines architectures d'empêcher l'exception sans pour autant permettre l'accès direct au registre impacté)
- Mais un processus complice peut récupérer l'adresse accédée via une attaque SCA contre le cache et donc en déduire la donnée lue dans la zone noyau
- Comme l'intégralité de la mémoire physique est mappée dans l'espace noyau, il est ainsi possible de lire le contenu de toute la RAM physique

Attaque *Meltdown* [6]

Solutions

- Au niveau logiciel, il est possible de limiter l'impact de cette attaque en arrêtant de mapper les structures du noyau dans l'espace utilisateur
- KAISER [4]
- Impact non négligeable sur les performances car nécessité de modifier les tables de page à lors des basculements espace utilisateur / espace noyau

Attaque *Spectre* [5]

- Dans les différentes variantes de l'attaque *Spectre*, les mécanismes de prédiction de branchement du processeur sont visés ainsi que l'exécution spéculative qui en résulte
  - Lors d'une instruction de branchement conditionnel, le processeur fait une hypothèse sur le résultat de ce branchement et commence à exécuter spéculativement les instructions suivantes
  - Lors d'une instruction de saut, le processeur prédit également l'adresse de destination du saut
- Ces deux mécanismes fonctionnent avec apprentissage et cet apprentissage peut être utilisé par l'attaquant pour forcer l'exécution spéculative d'une portion de code qui ne devrait pas l'être
- Cette exécution spéculative va laisser des traces, notamment au niveau du cache, qui vont pouvoir être exploitées comme pour l'attaque *Meltdown*

Attaque *Spectre* [5]

Variante 1

- Dans le code du processus visé (exemple tiré de [5])

```
if (x < array1_size)
    y = array2[array1[x] * 256]
```

- Hypothèses
  - `x` est contrôlable par l'adversaire et est choisi (hors des limites du tableau) de manière à ce que `array1[x]` pointe vers une valeur secrète *k*
  - `array1_size` et `array2` ne sont pas présents dans le cache mais *k* l'est
  - Lors des exécutions précédentes du test, la valeur de `x` fournie était correcte et donc le prédicteur de branchement va considérer que le résultat du test sera probablement vrai

Attaque *Spectre* [5]

Variante 1

- Lors de l'exécution avec la valeur de `x` malicieuse, `array_size` n'étant pas dans le cache, le processeur se base sur la prédiction de branchement et se met à exécuter spéculativement l'instruction suivante en attendant la lecture de `array_size`
- La valeur de `array1[x]` (k) étant connue, le processeur peut commencer tout de suite la lecture à l'adresse `array2[array1[x] * 256]`, qui n'étant pas dans le cache, sera rapatriée depuis la mémoire
- Lorsque le résultat du test est finalement connu, l'effet de la seconde instruction est annulé
- Néanmoins, une ligne de cache dont l'adresse dépend de la valeur secrète k aura été chargée dans le cache et donc une attaque SCA contre ce dernier permettra de retrouver k

Attaque *Spectre* [5]

Variante 2

- L'adresse de destination d'un saut n'est pas toujours connue (ex. instruction `jmp [eax]`)
- Si la détermination de la destination du saut est retardée par un défaut de cache, le processeur va utiliser le prédicteur de branchement pour la deviner et exécuter spéculativement les instructions
- Si ce mécanisme peut être entraîné malicieusement par l'adversaire, ce dernier peut s'en servir pour faire exécuter spéculativement des instructions à des adresses qu'il a choisies (similaire au ROP)
- L'adversaire choisi ces instructions de telle sorte à ce qu'elles fassent fuir de l'information sensible via le cache

Plan

Conclusion

- Le support d'exécution offre un certain nombre de mécanismes pour améliorer la sécurité logicielle
- Néanmoins, il appartient au logiciel (le plus souvent au système d'exploitation ou à l'hyperviseur) de bien s'en servir
- N'empêchent pas tous les problèmes logiciels...

Références

[1] Victor Costan and Srinivas Devadas.
Intel SGX explained.
Cryptology ePrint Archive, Report 2016/086, 2016.
http://eprint.iacr.org/2016/086.

[2] Loïc Duflot.
Bogues et piégeages des processeurs, quelle conséquence sur la sécurité ?
In *SSTIC*, June 2008.
http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lti/sstic08-duflot.pdf.

[3] Loïc Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard.
System management mode design and security issues.
In *IT Defense*, February 2010.
http://www.ssi.gouv.fr/IMG/pdf/IT_Defense_2010_final.pdf.

[4] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard.
KASLR is Dead : Long Live KASLR, pages 161–176.
Springer International Publishing, Cham, 2017.

[5] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom.
Spectre attacks : Exploiting speculative execution.
ArXiv e-prints, January 2018.
https://meltdownattack.com/spectre.pdf.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg.
Meltdown.
ArXiv e-prints, January 2018.
https://meltdownattack.com/meltdown.pdf.