INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Enseignant : *Vincent Migliore*
vincent.migliore@insa-toulouse.fr

Sécurité des Architectures - Introduction aux Attaques Temporelles sur les Caches

Introduction

Les caches sont des composants incontournables des processeurs modernes permettant d'assurer une utilisation optimisée des ressources mémoires. Ils respectent toujours deux principes : la localité temporelle en gardant dans sa mémoire les données récentes demandées par le cœur de processeur; et la localité spatiale en anticipant ses besoins futurs en récupérant les données dont les adresses sont proches des données accédées précédemment. Cependant, les caches restent des composants partagés : l'ensemble des tâches qui s'exécutent sur un processeur partagent l'utilisation caches. Bien que l'isolation de l'accès aux données des autres tâches s'effectue par l'utilisation de mémoires virtuelles distinctes, l'exécution d'un programme peut influencer le comportement du cache et ainsi faire fuiter de l'information potentiellement sensible vers des tâches malveillantes observant le comportement de ce dernier.

C'est typiquement le cas des attaques temporelles sur les caches, qui vise à exploiter le fait que le temps d'accès à une adresse mémoire dépend de sa présence ou non dans les caches du processeur. Bien que théoriquement, ces attaques soient simples à comprendre, en pratique elles sont loin d'êtres faciles à implémenter. Dans la suite du TP, nous allons chercher à démystifier leur utilisation, et analyser leurs limites.

Ayant besoin d'accès assez bas niveau au matériel, nous allons utiliser le langage c pour nos expérimentations. Afin de vous aider à contrôler le comportement du cache, nous vous mettons à disposition un certain nombre de fonctions dont une description détaillée est fournie en annexe de ce document. Parmi les plus utilisées nous avons la fonction **clflush** qui permet de supprimer une donnée du cache, **memaccesstime** permettant de mesure le nombre de cycle d'accès à une adresse et **map_address** qui permet de mapper un fichier dans la mémoire du processus.

Partie 1 - Profilage des temps d'accès au cache

Cette première partie vise à mettre en évidence le profile du cache de votre processeur en étudiant les temps d'accès à ce dernier dans le cas d'un cache hit et d'un cache miss. Pour se faire, rendez-vous dans le dossier **1_CACHE_PROFILING** dans le dossier du TP et ouvrez le fichier **profiling.c** que vous aurez à compléter.

L'idée est de mener une campagne de test qui vise à récupérer de nombreuses mesures du nombre de cycles d'accès au cache dans le cas d'un cache hit d'une part, et d'un cache miss d'autre part. Pour vous aider, on vous fourni un pointeur nommé **ptr** que vous pouvez utiliser pour faire vos mesures. Ce pointeur est issu de la projection dans la mémoire du fichier **profiling.c**, qui sera à une adresse stable de la mémoire pendant toute la durée du programme. Il est tout à fait possible de faire les mêmes expériences avec des variables déclarées dans le code, il faut s'assurer cependant que le compilateur ne fera pas d'optimisation dessus. Pour se faire, il est préférable d'utiliser une variable globale, et de lui attribuer le qualificatif **volatile** pour réduire au maximum les optimisations possibles par le compilateur sur cette variable.

Afin de visualiser les résultats, on propose d'utiliser un histogramme qui prendra en abscisse le nombre de cycles mesurés, et en ordonnée le nombre d'occurrence ou un cache hit (resp. cache miss) a duré ce nombre précis de cycles. Ainsi, si pendant la campagne de test, 150 cache hits ont été exécutés en 100 cycles processeurs, alors la valeur de l'ordonnée de l'histogramme à l'abscisse 100 sera de 150.

Pour mener à bien ce profilage, vous utiliserez les constantes prédéfinies **RUNS** et **LIMIT**. **RUNS** définit le nombre de fois que l'on va tester le nombre de cycles d'accès au cache dans le cas d'un cache hit et un cache miss, et **LIMIT** donne la valeur maximale de l'abscisse de l'histogramme. Vous n'avez pas à gérer l'affichage du graphe, celui-ci a déjà été préparé en utilisant gnuplot. Pour tester votre programme, il suffira d'utiliser la commande **make** pour produire l'exécutable.

Partie 2 - Exemple d'exploitation du cache : La détection d'ouverture de fichiers

Cette deuxième partie vise à exploiter les temps d'accès au cache pour déterminer depuis un processus espion si un fichier a été ouvert par l'utilisateur ou un autre processus. En principe, cette information n'est pas divulguée naturellement, seul le noyau a par nature la connaissance de cette information. Vous coderez l'attaque dans le fichier **spy.c** qui est dans le dossier **2_SPY_FILE**. Pour cela, vous allez vous aider de la fonction fournie **map_offset**, elle-même utilisant la fonction standard **mmap** qui permet de mapper un fichier dans son espace d'adressage. Pour vous aider, vous pouvez chercher la description de la fonction **mmap** pour comprendre son fonctionnement. Petite remarque cependant, **mmap** nécéssite un fichier non vide. Autre indication également, vous aurez probablement à ajouter un délai dans votre boucle de test. Pour cela vous pouvez utiliser la fonction **delay**.

Partie 3 - Rétroconception du prefetcher

Cette troisième partie vise à étudier le mécanisme du prefetch qui a un impact significatif sur les attaques temporelles sur les caches. Pour rappel, le prefetch est un mécanisme interne du processeur qui se déclenche automatiquement lors de l'accès séquentiel à une plage d'adresses. Le processeur va alors anticiper et précharger les données aux adresses suivantes, bien au delà de celles qui sont naturellement préchargées lors du remplissage d'une ligne de cache.

L'objectif de cette partie est de faire la rétro-conception du prefetcher du processeur, en mettant un place une méthodologie permettant de mettre en évidence le nombre d'adresse précis préchargées avec ce mécanisme.

Pour se faire, rendez-vous dans le dossier **3_PREFETCH_PROFILING** dans le dossier du TP. Vous aurez à compléter le fichier **profiling.c**. On vous propose d'utiliser le un tableau de 4096 octets nommé **data**, auquel vous pourrez faire des lectures (ou écritures) séquentielles pour déclencher le prefetch. On vous propose de tester le parcours de 64, 65, 128, 129, 256, 512, 1024 et 2048 octets du tableau **data**. Afin d'avoir des résultats lisibles, faites de préférence plusieurs tests et faites la moyenne de vos résultats.

Le reste est à faire en autonomie.

Partie 4 - Analyse d'une vulnerabilité sur une implémentation d'AES sur openSSL

Cette quatrième partie vise à analyser des potentielles vulnérabilités sur l'implémentation de l'algorithme de chiffrement AES sur une version ancienne d'OpenSSL (version 0.9.7a en particulier). OpenSSL est une bibliothèque de référence pour la mise en place de connexions sécurisées notamment

sur internet. Vous avez à votre disposition le sources de l'implémentation dans le dossier **4_AES_VULNERABILITY_ANALYSIS/openssl_0_9_7_a**. Vous analyserez en particulier comment a été implémenté la fonction de chiffrement nommé **AES_encrypt**, et dans quelle mesure le cache peut être utilisé pour récupérer de l'information sur la clé. En fonction des résultats de la partie précédente, vous regarderez s'il peut y avoir un effet du prefetch sur l'attaque.

Pour aller plus loin, on vous donne la dernière version d'OpenSSL dans le dossier **4_AES_VULNERABILITY_ANALYSIS/openssl_1_1_1_t**. Vous pourrez chercher si des évolutions notables ont été intégrées. Pour vous aider, vous pouvez regarder le document `https://eprint.iacr.org/2020/907.pdf`.

Partie 5 - Profilage du prefetch du cache d'instructions

Cette cinquième partie vise à étudier le mécanisme du prefetch, non plus sur le cache de données (comme la partie 3), mais le cache d'instruction. Pour se faire, rendez-vous dans le dossier **5_INSTRUCTION_CACHE_PREFETCH_PROFILING** dans le dossier du TP. En se reposant sur une approche similaire à la partie 3, étudier comment le prefetch fonctionne dans le cas d'exécution de fonctions. Pour cela, vous avez à votre disposition une bibliothèque dynamique nommée **libprofiling.so** dans lequel un certain nombre de fonctions dont le nom est de la forme **fun_***. Vous allez tout d'abord analyser le code des fonctions (vous pouvez utiliser **objdump**) puis mener vos expérimentations. Vous comparerez vos résultats avec la partie 3 et en déduire les avantages et inconvénients de cibler des données ou du code.

Annexe : Description des fonctions mises à disposition pour l'analyse du cache

void clflush(void* ptr)

Arguments

	type	description
ptr	void*	Adresse virtuelle cible

Description

Requête du processeur visant à invalider la ligne associée à l'adresse mémoire pointée par **ptr** de toute la hiérarchie cache.

uint64_t memaccesstime(void* ptr)

Arguments

	type	description
ptr	void*	Adresse virtuelle cible

Description

Revoie le nombre de cycles processeurs nécessaires pour accéder à l'adresse mémoire pointée par **ptr**.

void delay(uint64_t cycles)

Arguments

	type	description
cycles	uint64_t	Nombre de cycles

Description

Execute une temporisation de **cycles** cycles processeurs.

void* map_offset(const char *file, uint64_t offset)

Arguments

	type	description
file	const char *	Nom de fichier cible
offset	uint64_t	Décalage dans le fichier

Description

Créer une projection du fichier **file** dans l'espace mémoire virtuel et renvoie un pointeur vers l'adresse mémoire décalée de **offset** octets par rapport à l'adresse de début de cette projection.