La Prise De RISC Rapport de protection de ZEPHYR contre RIPE

***Abstract*—Dans le cadre du concours CV32A6 RISC-V softcore hackathon, notre équipe devait protéger le RTOS Zephyr contre des attaques visant à altérer le flot d'exécution. Ces attaques, générées par l'outil RIPE fonctionnaient selon différentes techniques, visant différentes zones mémoire.**

***Index Terms*—CV32A6, SystemVerilog, RIPE, Thales, CNFM, CNRS**

I. Introduction

Nous présentons ici les différentes méthodes utilisées pour protéger le coeur CVA6 et le RTOS Zephyr de ces attaques. Le dépot github et les instructions de déploiement sont sur https://github.com/LaPriseDeRISC/cva6-softcore-contest-lpdr .

II. Protection matérielle, modification de la RAS

A. Description des attaques cibles

Sur les 10 attaques à contrer, 3 d'entre elles changent le flot de contrôle en réécrivant, d'une manière ou d'une autre, le pointeur de retour d'une fonction lors de son exécution. Ce pointeur est normalement contenu dans un registre spécial, x1, sous le nom de l'ABI "ra" (pour Return Address) Il doit cependant être sauvegardé et restauré dans la pile lors des appels de sous fonction, de même que les autres registres. C'est a ce moment la que sa valeur peut être altérée, et donc que l'attaque se produit.

B. Systèmes de protection du pointeur de retour

Il existe plusieurs méthodes pour protéger le pointeur de retour lorsqu'il est sauvegardé dans la pile. Certaines se basent sur des techniques génériques de protection d'overflow (comme les canaries de pile), d'autres en font des copies sécurisées (comme les Shadow Stacks). Il existe des avantages et inconvénients aux deux méthodes :

- Les canaries prennent de la place sur la stack, les vérifier consomme du temps de calcul, et il faut faire attention à bien les placer et bien les générer. (Un canary statique peut être facilement deviné par brute-force)
- La shadow stack prend de la place en mémoire, il est difficile de l'adapter à un système multithreadé, il faut bien la placer et la protéger en mémoire.

C. Système utilisé

Nous avons opté pour une méthode analogue à la Shadow Stack, mais purement matérielle. Un composant déjà présent dans le CV32A6, la RAS (Return Address Stack), sauvegarde déjà les pointeurs de retours lors des appels de fonctions, pour des raisons d'optimisation de performances, en effet, lors de l'exécution spéculative, quand une instruction 'ret' arrive au fetch, la RAS fournit une adresse à laquelle continuer l'exécution, en continuant ainsi la spéculation.

D. Faiblesses de la RAS

Cependant ce composant ne peut pas être utilisé en état comme moyen de vérifier que ces pointeurs ne sont pas altérés lors de leur restauration depuis la pile. Comme c'est un prédicteur de branchement, son résultat n'est pas forcément juste, elle peut se retrouver "corrompue". Un exemple classique est :

- Lors de l'exécution spéculative d'une sous-fonction f, un branchement conditionnel suivi d'un 'ret' est rencontré.
- La BHT prédit le branchement comme non pris
- L'exécution continue sur l'instruction 'ret'
- L'entrée de la RAS est lue et l'exécution continue sur l'instruction suivant directement l'appel de f
- Si cette instruction est un nouvel appel de sous-fonction, l'entrée de la RAS se retrouve réécrite
- Le branchement conditionnel arrive à l'étage EX, il a été mal prédit
- L'exécution reprend dans la sous-fonction f, mais la RAS ne connait plus le pointeur de retour

E. Challenges de la modification à des fins de sécurité

1) Fiabilisation de la RAS: La principale difficulté à été de rendre la RAS résistante aux erreurs de spéculation. Ceci dans le but d'éviter sa corruption et de garantir son résultat.

Le système retenu à été de suivre les instructions modifiant la RAS lors de leur exécution dans le pipeline. Sur le CVA6, une instruction peut être annulée à 2 endroits : avant l'étage Execute (lorsqu'un branchement à été mal prédit), avant l'étage Commit (lorsqu'une exception est levée, ou qu'un composant à besoin de vider le pipeline). La RAS doit donc pouvoir restaurer son état en cas de suppression d'une instruction, à l'état antérieur au fetch de cette même instruction.

2) Implémentation matérielle: Les ressources disponibles sur un FPGA sont limitées, on ne peut donc pas imaginer une RAS de taille suffisante équipée d'une recherche associative. L'implémentation s'articule donc autour des ressources de

BRAM, disponibles sur le FPGA du concours. Ces BRAM (Block RAM) ne possèdent au maximum que 2 ports (True Dual Port, lecture / écriture indépendante sur les deux ports), il a donc fallu créer une logique de contrôle intelligente pour implémenter ce système.

De part l'architecture du coeur, la restauration de l'état de la RAS doit se faire en 1 cycle, on ne peut donc pas implémenter des boucles de restauration pour annuler de multiples actions si il en a.

La solution retenue à été de créer 3 RAS, communiquant entre elles, une dans chaque zone "effaçable" (Fetch - Execute), (Execute - Commit). La dernière étant celle dont les valeurs ne peuvent plus être altérées par les aléas de l'exécution. Elles sont connectées en cascade, ce qui permet de toujours accéder à la valeur la plus récente.

3) Support logiciel et Multithreading: Le multithreading reste un problème de cette solution. Il est possible de modifier la RAS afin de permettre le suivi de plusieurs files d'exécution (une ébauche de solution à été implémentée dans ce sens) mais pour ce concours, nous avons simplement fait en sorte d'invalider la RAS à chaque changement de contexte. Nous avons donc implémenté dans le support de RISC-V de Zephyr, un hook permettant d'appeler du code lors des changements de contexte. En utilisant ce hook, notre version de Zephyr commande la RAS grâce à des CSR, pour la réinitialiser lors des changements de contexte, désactivant la protection lors de ces phases mais permettant de tester la solution directement. Les applications du concours (dont RIPE) s'executent dans un thread unique, donc ça ne change pas les résultats.

F. Résultats

1) Sécurisation de l'exécution:

2) Utilisation des ressources:

3) Performances: Nous avons observé une grande amélioration des performances du coeur en utilisant la RAS modifiée. En effet, l'application perf_baseline fait beaucoup d'appels récursifs et bénéficie beaucoup de la précision de notre version de la RAS lors de l'exécution spéculative Sans modification de Zephyr autre que la commutation de contexte, son temps d'exécution passe de 25.3s (référence sur le git du concours) à 23.5s

III. Table des allocations

A. Problématique

Cette section traite des attaques schématisées dans la partie gauche de la **Figure ??**. La heap est une zone mémoire gérée par le logiciel et dont le comportement est inhérent au RTOS Zephyr. Ainsi, le moyen choisi pour protéger cette section de la mémoire se fait à travers une gestion plus développée de l'allocation dynamique. En effet l'implémentation présente dans la libc utilisée par le RTOS Zephyr ne contient aucun mécanisme d'enregistrement des allocations, qu'il s'agisse de leur identification ou de leur taille. Il est alors nécessaire de modifier l'implémentation de l'allocation dynamique pour permettre une protection contre le dépassement de tampon par les fonctions vulnérables.

B. Solution: Table des allocations

Une solution à ce problème est l'utilisation d'une table des partitions. Pour ce faire, nous devons modifier l'implémentation de la libc en modifiant le malloc() pour enregistrer les partitions ainsi que les fonctions sensibles à l'overflow comme sprintf ou memcpy.

C. Contraintes

La structure de donnée doit avoir une complexité minimale en lecture et une complexité réduite en écriture. Étant donné que l'évaluation se compte en nombre de cycle il est très important de minimiser les lectures et les écritures mémoires, notamment en utilisant correctement le cache.

Plusieurs structures de données se présentent alors à nous pour organiser la table des partitions, nous nous sommes concentrés sur les structures simples.

D. Arbre Binaire de Recherche

L'arbre binaire de recherche a une complexité au pire cas en insertion de $\Theta(n)$ et une complexité au pire cas en recherche de $\Theta(n)$, elle permet la recherche de valeur de meilleure façon en moyenne qu'une liste mais pas au pire cas (voir **Figure ??**). Une structure type pour les entrées de la table utilisant un arbre binaire de recherche est présentée dans le **Bloc ??**.

Les arbres binaires de recherches sont un bon point d'entrée mais manquent d'élégance quant aux accès en lecture à cause de la verticalité trop importante. L'importance de comparer les performances en lecture des ABR avec une autre structure de donnée se fait sentir pour optimiser le résultat final.

E. Arbre AVL

Les arbres AVL sont une évolution des arbres binaires de recherche avec notamment la modification de l'insertion et de la suppression pour y ajouter des opérations de rotations pour équilibrer les branches. Cette petite modification qui n'a d'effet qu'à l'écriture et à la suppression (rare dans notre cas) assure une complexité en log(n) à la recherche. La **Figure ??** présente un arbre AVL naturellement équilibré.

F. Comparaison de la rapidité en lecture

Nous avons comparé la rapidité de 3 méthodes (Naïf, AVL, ABR) sur un échantillon de 1000 entrées en comparant la vitesses de recherche séquentielle de la totalité des éléments la composant. Par rapport à la méthode naïve (recherche classique dans un tableau désordonné), l'arbre binaire non-AVL affiche 3500% de performances supplémentaires par rapport à la méthode naïve alors que l'AVL affiche un gain moyen de 8000% soit 2x plus qu'un arbre non équilibré.