ÉLÉMENT DE PORTFOLIO 01

Publication

1 DÉFINITION DE CET ÉLÉMENT

Titre de l'élément : ARMISTICE : Modélisation de fuites micro-architecturales pour la vérification formelle de programmes masqués [1]

URL de l'élément : `https://hal.sorbonne-universite.fr/hal-03954892`

2 MOTIVATIONS DU CHOIX DE CET ÉLÉMENT

Armistice constitue une avancée significative dans le domaine de la vérification de masquage, car c'est le premier travail qui vérifie de manière formelle les programmes avec un modèle de fuite en prenant en compte la micro-architecture du processeur. Ce travail combine des aspects à la fois théoriques et pratiques, allant de la simulation de code sur un coeur à la vérification formelle d'expressions symboliques, en passant par la conception de programmes de tests pour la caractérisation physique des fuites sur un processeur.

3 PRÉSENTATION DE CET ÉLÉMENT

Les attaques par canal auxiliaire (SCA) exploitent les mesures physiques, comme la consommation d'énergie ou les émissions électromagnétiques (EM), pendant l'exécution d'une application pour récupérer des données secrètes. Elles permettent de casser des implémentations d'algorithmes cryptographiques prouvés algorithmiquement sûrs.

Une contre-mesure à ces attaques est le masquage, qui vise à coder une donnée secrète en $d+1$ parties appelées *shares*, de telle sorte que toute combinaison de moins de $d+1$ parts soit statistiquement indépendante du secret. Cela empêche théoriquement les SCA, car la consommation d'énergie et les émissions EM sont directement liées aux valeurs manipulées par le programme.

La mise en œuvre d'un schéma de masquage au niveau logiciel sans démasquer les secrets et la détection manuelle de ce démasquage sont loin d'être des tâches simples. Par conséquent, certaines techniques et outils de vérification ont été récemment proposés pour aider les concepteurs à détecter les failles dans leurs implémentations.

Avec Armistice, nous montrons que les outils et modèles de fuite actuels sont insuffisants pour garantir une sécurité pratique. À titre d'exemple, la figure 1 montre un code du AND masqué selon un schéma classique (appelé ISW), ainsi que le code assembleur correspondant généré. Ce dernier ne comporte pas de fuite avec un modèle prenant en compte les registres généraux du processeur (vue ISA). La partie droite de la figure illustre que malgré cela, la consommation est fortement corrélée avec les valeurs secrètes (entrées et sortie). Cela est dû au fait que le modèle de fuite considéré n'est pas assez proche du matériel.

Dans Armistice, nous avons analysé le code Verilog d'un processeur Arm Cortex-M3 afin d'en extraire un chemin de données et d'en faire un modèle qui serve de base à la modélisation de la consommation. Dans ce but, nous avons réalisé de nombreux "test vectors" de fuite, dont le but était triple : 1) Confirmer que les valeurs qui transitent dans les éléments matériels modélisés du cœur ont un effet visible sur la consommation ; 2) Déduire un modèle matériel de la mémoire pour laquelle nous n'avions pas accès au RTL ; 3) Déterminer pour chaque composant matériel modélisé le nombre de traces requises pour caractériser une fuite.

Tous ces tests vectors et les résultats associés sont disponibles à l'URL suivante : `https://www-soc.lip6.fr/armistice/`

Une fois le modèle du processeur implanté, nous l'avons associé à un outil de vérification d'expressions symboliques (figure 2). Le modèle du cœur simule l'exécution du code assembleur cycle par cycle, générant ainsi des expressions symboliques masquées dans les différents éléments modélisés du cœur (registres, bus). Ces expressions sont ensuite vérifiées par l'outil LeakageVerif [2], qui analyse ces expressions à la recherche de fuite secrète. En cas de fuite, l'outil est donc capable de déterminer le cycle de la fuite, l'élément matériel impliqué, ainsi que

Listing 1 – ET logique selon le schéma ISW

```
// Inputs: - Secrets a = a0 ^ a1, b = b0 ^ b1
//         - Mask m
// Output: - Secret c = c0 ^ c1
aux0 = m ^ (a0 & b1);
aux1 = aux0 ^ (a1 & b0);
c0 = (a0 & b0) ^ m;
c1 = (a1 & b1) ^ aux1;
```

Listing 2 – Code assembleur produit par GCC

```
; r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
and.w    r4, r0, r3    ; a0 & b1
eors     r4, r7        ; aux0 = (a0 & b1) ^ m
and.w    r5, r2, r1    ; a1 & b0
ands     r0, r1        ; a0 & b0
ands     r3, r2        ; b1 & a1
eors     r4, r5        ; aux1 = aux0 ^ (a1 & b0)
eors     r0, r7        ; c0 = (a0 & b0) ^ m
eors     r4, r3        ; c1 = aux1 ^(a1 & b1)
str r0, [r6, #0]
str r4, [r6, #4]
```

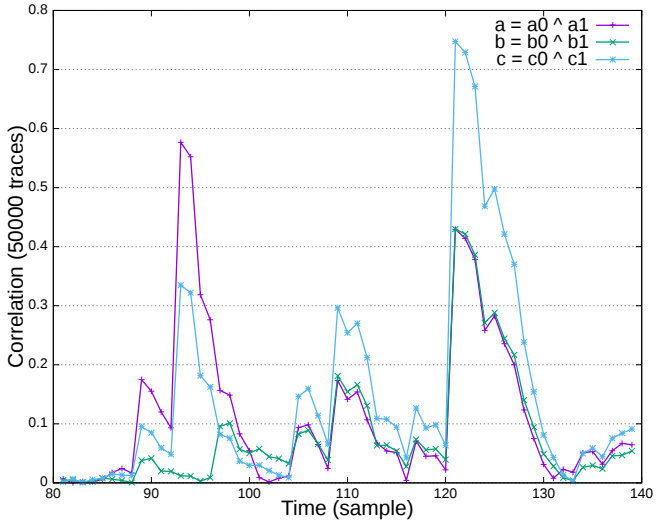
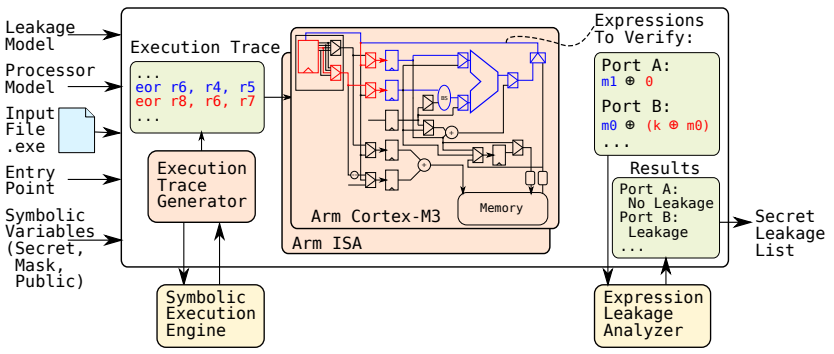


FIGURE 1 – ET logique masqué selon le schéma ISW et prenant en compte les registres généraux du processeur ; Consommation associée à l'exécution du code



Symbolic Expressions in Hardware Components

ISA Model

Initial	
r4	m1
r5	m0
r6	0
r7	k⊕m0
r8	0

Final	
r6	m0⊕m1
r8	k⊕m1

Subset of the Cortex-M3 Model

Cycle	0	1	2	3	4
r4	m1	m1	m1	m1	m1
r5	m0	m0	m0	m0	m0
r6	0	0	0	m0⊕m1	m0⊕m1
r7	k⊕m0	k⊕m0	k⊕m0	k⊕m0	k⊕m0
r8	0	0	0	0	k⊕m1
Port A	-	m1	0	...	...
Port B	-	m0	k⊕m0	...	...

Leakage in transition on Port B

FIGURE 2 – Framework de vérification Armistice

l'expression symbolique concernée. Ces informations permettent de corriger le code afin de supprimer les fuites. De nombreux codes ont été vérifiés avec Armistice, tous exhibant des fuites au niveau micro-architectural, et la suppression des fuites par ajout d'instructions assembleur spécialement conçues d'après la sortie de l'outil a été faite pour une des applications.

4 RÉFÉRENCES BIBLIOGRAPHIQUES

[1] Arnaud De Grandmaison, Karine Heydemann, and Quentin L. Meunier. ARMISTICE : Microarchitectural Leakage Modeling for Masked Software Formal Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11) :3733–3744, November 2022.

[2] Quentin L Meunier, Etienne Pons, and Karine Heydemann. Leakageverif : Efficient and scalable formal verification of leakage in symbolic expressions. *IEEE Transactions on Software Engineering*, 2023.