CHAPITRE 1

Outils et méthodes

chapitre sur les outils + moyens pour détecter

1.1 Modélisation d'une attaque

En sécurité informatique, la première étape, essentielle avant de développer une solution, c'est de produire un modèle du danger que l'on souhaite cibler. On parle parfois de *modèle de fuite*. Cette étape de synthèse et d'abstraction est importante pour identifer les risques encourus par le futur système, souvent en identifiant les point de fuites employés par les attaques déjà publiées. SCHNEIDER et al. [Sch+25] nous donne les trois modèles d'adversaires que l'on doit considérer lorsque l'on souhaite se défendre contre les attaques temporelles :

TABLE 1.1 – Modèles d'adversaires pour les attaques temporelles [Sch+25]

Type d'attaque	Description
Par chronométrage	Observation du temps de calcul.
Par accès mémoire	Manipulation et observation des états d'un ou des caches mémoires.
Par récupération de trace	Suivi des appels de fonctions, des accès réussis ou manqués à la mémoire.

Ces trois modèles sont notre source de méfiance et si on peut argumenter quand à l'inclusion de notre dernier modèle ; des travaux comme [Gau+23] portent directement sur des améliorations matériel pour contrecarrer ce type d'attaque. Considérer un attaquant plus puissant, avec des accès à des ressources supplémentaires, potentiellement hyptotétique, permet de concevoir un système plus sûr. Certains outils comme [HEC20 ; Wei+18] ou cette étude [Jan+21] exploitent cette mécanique pour attester de la sécurité d'un programme.

Puis, avec ces modèles et les contre-mesures connus, on peut constituer un ensemble de règles qui valident ces risques. [Mei+21] résume celles-ci en une liste de trois règles :

1. Toute boucle révèle le nombre d'itérations effectuées.
2. Tout accès mémoire révèle l'adresse (ou l'indice) accédé.
3. Toute instruction conditionnelle révèle quelle branche a été prise.

D'autres comme [DBR19] emploient des modèles de fuites en représentation formelle. En s'appuyant sur les travaux de BARTHE, GRÉGOIRE et LAPORTE [BGL18]
- formalisation des regles de securite

Nous adoptons une position pessimiste, en supposant que chaque violation individuelle de ces règles fuit parfaitement vers l'adversaire.

La règle 1 se justifie par une observation triviale : une boucle plus longue utilise plus d'opérations. En pratique, il est difficile d'observer la durée de chaque boucle dans un programme plus vaste, ce qui rend cette règle pessimiste.

La règle 2 est justifiée par divers canaux auxiliaires et attaques basées sur le cache [**Ber05** ; **YGH17** ; **CAPGATB19**]. Puisque les caches ne chargent l'information qu'une ligne entière

à la fois, cette règle peut sembler trop pessimiste. Peut-être que seule la ligne de cache accédée devrait rester secrète [**Bri11**]. Malheureusement, il est possible de mener des attaques basées sur des accès à l'intérieur d'une ligne de cache [**BS13** ; **OST06** ; **YGH17**]. C'est pourquoi nous adoptons une position pessimiste, et supposons que les accès révèlent leur adresse exacte.

La justification de la règle 3 est double. Premièrement, si différentes branches d'une instruction conditionnelle exécutent un nombre différent d'opérations, on peut observer quelle branche a été prise. Deuxièmement, même si les deux branches exécutent des opérations identiques, le prédicteur de branche du processeur peut être exploité pour révéler des informations sur la branche sélectionnée [**AKS06** ; **AKS07** ; **EPAG16**].

En plus de ces règles, nous avons besoin d'un ensemble de base d'opérations de confiance pour construire nos programmes. Nous supposons que l'addition, la multiplication, les opérations logiques et les décalages, tels qu'implémentés matériellement, sont en temps constant par rapport à leurs entrées. C'est le cas sur la plupart des processeurs, une exception notable étant certains microprocesseurs [**Por**]. Cette hypothèse est raisonnable pour les plateformes ciblées par notre bibliothèque.

[DBR19]

1.2 Analyse d'un programme

- analyse statique - analyse dynamique - analyse symbolique - analyse de trace

CHAPITRE 2

Automatisme et couverture

chapitre sur les architectures à couvrir
les problèmes et les enjeux
les benchmarks en place

2.1 Outils et mode d'emploi

2.2 Emploi d'un usage industriel

Le premier outil à être créé est *ctgrind* [Lan10], en 2010. Il s'agit d'une extension à *Valgrind* observe le binaire associé au code cible et signale si une attaque temporelle peut être exécuter. En réalité, *ctgrind* utilise l'outil de détection d'erreur mémoire de *Valgrind* : Memcheck. Celui-ci détecte les branchement conditionnels et les accès mémoire calculés vers des régions non initialisée, alors les vulnérabilités peuvent être trouvées en marquant les variables secrètes comme non définies, au travers d'une annotation de code spécifique. Puis, durant son exécution, Memcheck associe chaque bit de données manipulées par le programme avec un bit de définition V qu'il propage tout au long de l'analyse et vérifie lors d'un calcul d'une adresse ou d'un saut. Appliquée à *Valgrind* l'analyse est pertinente, cependant, dans le cadre de la recherche de faille temporelle cette approche produit un nombre considérable de faux positifs, car des erreurs non liées aux valeurs secrètes sont également rapportées.