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] ou [Wei+18] exploitent cette mécanique pour vérifier les sources qui leurs sont soumises.

Avec cette base, on peut désormais cibler les éléments
[DBR19]
[Mei+21] We have distilled the concerns outlined in Section 2.1 into a simple list of rules, which constitute the threat model for our library : 1. Any loop leaks the number of iterations taken. 2. Any memory access leaks the address (or index) accessed. 3. Any conditional statement leaks which branch was taken. We take a pessimistic position, assuming that each individual violation of these rules leaks perfectly to the adversary. Rule 1 is justified by a trivial observation : a longer loop uses more operations. In practice, it is difficult to observe the duration of each loop in a larger program, making this a pessimistic rule. Rule 2 is justified by various cache based side-channels and attacks [Ber05, YGH17, CAPGATB19]. Since caches only load information an entire line at a time, this rule may seem too pessimistic. Perhaps only which cache line was accessed should be kept secret [Bri11]. Unfortunately, it is possible to perform attacks based on accesses inside of a cache line [BS13, OST06, YGH17]. This is why we take a pessimistic position, and assume that accesses leak their exact address. The justification for Rule 3 is twofold. First, if different

branches of a conditional statement execute a different number of operations, we can observe which branch was taken. Second, even if both branches execute identical operations, the CPU's branch predictor can be exploited to leak information about the selected branch [AKS06, AKS07, EPAG16]. 4In addition to these rules, we need a basic set of trusted operations to build our programs. We assume that addition, multiplication, logical operations, and shifts, as implemented in hardware, are constant-time in their inputs. This is the case on most processors, one notable exception being microprocessors [Por]. This assumption is reasonable for the platforms targeted by our library.

1.2 Exploration et d'une attaque

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.