CHAPITRE 1

# Outils et méthodes

chapitre sur les outils + moyens pour détecter

intro

## 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 traces | Suivi des appels de fonctions, des accès réussis ou manqués à la mémoire. |

Ces types d'attaques forment une base pour la conception de nos mdodèles d'attaquant. Considérer le mode opératoire «récupération de traces» induit un modèle plus fort. Des travaux comme ceux de GAUDIN et al. [Gau+23] portent directement sur des améliorations matériel permettant une défense contre ce modèle. 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 notamment 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.

Avec ces règles, il est alors possible de créer un outil qui analyse les programmes à sécuriser. C'est de cette façon que le premier outil existant a été produit : `ctgrind` (2010).

D'autres chercheurs comme DANIEL, BARDIN et REZK [DBR19] s'attellent à la création de modèles formels. Cette méthode demande un travail de formalisation du comportement de programmes binaire et une implémentation plus rigoureuse de leurs outils. Cela permet en retour une évaluation complète et correct de programmes complexes (*i.e.* primitives cryptographiques asymétriques).

### Formalisation de modèle - [DBR19]

Si nous voulons concevoir un modèle formel, nous pouvons nous appuyer sur l'article "Secure Compilation of Side-Channel Countermeasures : The Case of Cryptographic "Constant-Time"" [BGL18].

Nous commençons par définir un programme. Il s'agit d'une suite d'instruction binaire. Et une instruction est une action sur la mémoire. Cela nous permet de définir notre programme comme une suite de configuration $(l, r, m)$ ; $l$ la ligne d'instruction, $r$ le dictionnaire de registre et $m$ la mémoire. La configuration initiale est défini par $c_0 \triangleq (l_0, r_0, m_0)$ où $l_0$ est l'adresse de l'instruction d'entrée du programme, $r_0$ un dictionnaire de registre vide et $m_0$ une mémoire vide.

Ainsi, avec cette modélisation, une instruction est un changement appliqué à notre configuration. Ce changement peut être représenté par $c_0 \xrightarrow[t]{} c_1$, $c_0$ et $c_1$ deux configurations successives, $\rightarrow$ la transition entre les deux et $t$ une fuite émise par cette transition. Notons que certaines instructions ne produisent pas de fuites.

Une fois ce préambule installé nous définissons formellement le comportement de nos instruction. Regardons par exemple comment se formalise un chargement :

FIGURE 1.1 – Instruction `chargement`

$$\text{LOAD}\ \frac{(l, r, m)\ e \vdash_t \mathtt{bv}}{(l, r, m)\ @\ e \vdash_{t \cdot [\mathtt{bv}]} m\ \mathtt{bv}}$$

Ici, l'évaluation de l'expression `e` sur une configuration $(l, r, m)$ produit une fuite de la valeur $bv$. En haut nous retrouvons la notation de l'opération effectuée et au dessous la formalisation de la fuite : $t \cdot [bv]$ signifie que la valeur $bv$ s'ajoute à la liste des fuites. Ce second exemple 1.2 présente une opération de branchement en fonction de $e$ vers les instructions $l_1$ et $l_2$. On voit que la valeur est différente de zéro, ce qui nous produit une fuite vers l'instruction $l_1$. Cette fuite est à ajouter à notre liste $t$.

FIGURE 1.2 – Instruction `branchement`

$$\text{T-ITE}\ \frac{P.l = \mathtt{ite}\ e\ ?\ l_1 :\ l_2 \qquad (l, r, m)\ e \vdash_t \mathtt{bv} \qquad \mathtt{bv} \neq 0}{(l, r, m) \xrightarrow[t \cdot [l_1]]{} (l_1, r, m)}$$

Nous pouvons retrouver l'ensemble des règles formelles en Annexe 3.1.

## 1.2 Analyse d'un programme

Nous avons conçus un modèle pour contrôler ou détecter les erreurs. Nous pouvons maintenant concevoir notre analyse pour vérifier ce modèle avec un support. Plusieurs techniques existent et nous allons les passer en revue : [Gei+23].

### Analyse statique

Cette méthode consiste à déduire le fonctionnement d'un programme. Nous souhaitons vérifier que son fonctionnement respecte les propriétés de sécurité que nous avons définies. Cette analyse sans exécution réalise une simulation du programme en explorant les chemins d'exécution possibles. De fait, les résultats obtenues sont souvent approximé car une exploration totale peut se révéler irréalisable. Historiquement il s'agit de la première méthode étudiée et employée, en revanche elle a été dérivée en plusieurs approches.

**Non interférence.** Pour renforcer les résultats obtenues et réduire le nombre de faux positifs ont peux vérifier la propriété de non-interférence. Cette propriété est inhérente aux programmes. Un programme a des entrées et des sorties. Celles-ci peuvent être classées *faibles* (peu importantes) ou *hautes* (données secrètes, sensibles). Un programme est noninterférent si et seulement si pour n'importe quel entrée faible le programme ressort la même sortie faible peu importe les entrées hautes qui peuvent être préciser.

Appliqué à une analyse statique pour la vérification de programme, la mesure des ressources employé par l'ordinateur permet d'avoir une sortie faible pour comparer le comportement d'un programme en fonction de ses entrées (ici considérées secrète).

***Self-Composition***[1] La self-composition consiste à entrelacer deux exécutions d'un programme $P$ avec différents ensembles de variables secrètes dans un seul programme autocomposé $P; P'$. Des solveurs peuvent alors être utilisés pour vérifier la propriété de noninterférence. Cette approche a été utilisée par ALMEIDA et al. [Alm+13] pour vérifier manuellement des exemples limités, nécessitant de nombreuses annotations pour limiter l'explosion (quadratique) des états à comparer et explorer. [DBR19] emploie cette approche asocié à des solveurs SMT pour vérifier uniquement les propriétés définies dans leur modèle. La restriction aux propriétés temps constant permet l'exploitation de cette méthode.

**Systèmes de types** Cette approche diffère des précédentes car elle nécessite un travail supplémentaire du développeur. Il doit ajouter la spécification `secret` au valeurs employée pour que cette information se diffuse dans le compilateur et que des mesures adaptés soit effectuées au niveau du binaire. Cette approche est intéressante car elle permet une fléxibilité plus importante lors de la production du code et permet de s'abstenir des contre mesures d'écrite au chapitre **??**; en revanche elle nécessite un compilateur spécialisé et aucune vérification sur le binaire produit n'est effectuée.

**Interprétation abstraite** Un programme est (généralement) trop complexe pour être entièrement formellement vérifié, donc il y a une sur approximation des états atteignables par l'analyse. Ainsi, si l'analyse approximé est sécurisé alors le programme est sécurisé. Cette approche ce retrouve dans CacheAudit [Doy+13] : modélisation par un graphe de flot de l'état des caches, de la mémoire et des successions d'évènements.

**Exécution symbolique** L'exécution symbolique consiste à exécuter le programme avec des entrées symboliques. Les chemins explorés sont associés à une formule logique, et un solveur vérifie si un ensemble de valeurs concrètes satisfait les formules générées. Cette méthode est utiliser pour vérifier l'absence de dépendance aux secrets dans les comportements temporels ou mémoire du programme.

### Analyse dynamique

L'analyse dynamique emploie la preuve par l'exemple pour garantir la sécurité du programme cible. Nous exécutons le programme et nous collectons sa trace : informations issus des évènements (accès mémoires, sauts,*etc*) rencontrés au fur et à mesure de l'execution. Les approches diffèrent dans la collecte et la production de ces traces.

**Trace unique** Explorer tout les comportements d'un programme est couteux en temps, et pour les besoins du développement il peut être préférable d'étudier quelques cas particulier entièrement. Cette approche simplifie le modèle de l'attaquant est réalise sa vérification plus rapidement. `ctgrind` [Lan10] réutilise l'analyse dynamique de Valgrind pour vérifier les propriétés temps constant. Pour ajouter de la précision, il est possible d'utiliser l'exécution symbolique pour rejouer la trace avec le secret comme valeur symbolique et vérifier la violation du temps constant (CacheD [Wan+17]).

**Comparaison de traces** Les tests statistiques peuvent vérifier si différents secrets induisent des différences significatives dans les traces enregistrées. Des outils comme DATA [Wei+20] ou MicroWalk [Wic+18] utilisent diverses méthodes statistiques ou d'apprentissage pour

1. Construction personnelle, le terme anglais est conservé.

détecter et localiser les fuites. D'autres outils comme dudect [RBV17] enregistrent simplement le nombre total de cycles d'horloge et comparent leur distribution selon les secrets.

Le fuzzing peut aussi être utilisé pour trouver des entrées maximisant la couverture et la fuite via canal auxiliaire, comme dans ct-fuzz [HEC20].

*Transition*

CHAPITRE 2

# Automatisme et couverture

chapitre sur les architectures à couvrir
les problèmes et les enjeux
les benchmarks en place
introduction Binsec
- intro

## 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.

https ://blog.cr.yp.to/20240803-clang.html

*Transition*

CHAPITRE 3

# Références

TABLE 3.1 – Liste des options de compilations et leurs effets (non exhaustive), `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

| **Option de compilation** | **Effet** |
|---|---|
| -O0 | Compile le plus vite possible |
| -O1 / -O | Compile en optimisant la taille et le temps d'exécution |
| -O2 | Comme -O1 mais en plus fort, temps de compilation plus élevé mais exécution plus rapide |
| -O3 | Comme -O2, avec encore plus d'options, optimisation du binaire |
| -Os | Comme -O2 avec des options en plus, réduction de la taille du binaire au détriment du temps d'exécution |
| -Ofast | optimisations de la vitesse de compilation |
| -Oz | optimisation agressive sur la taille du binaire |

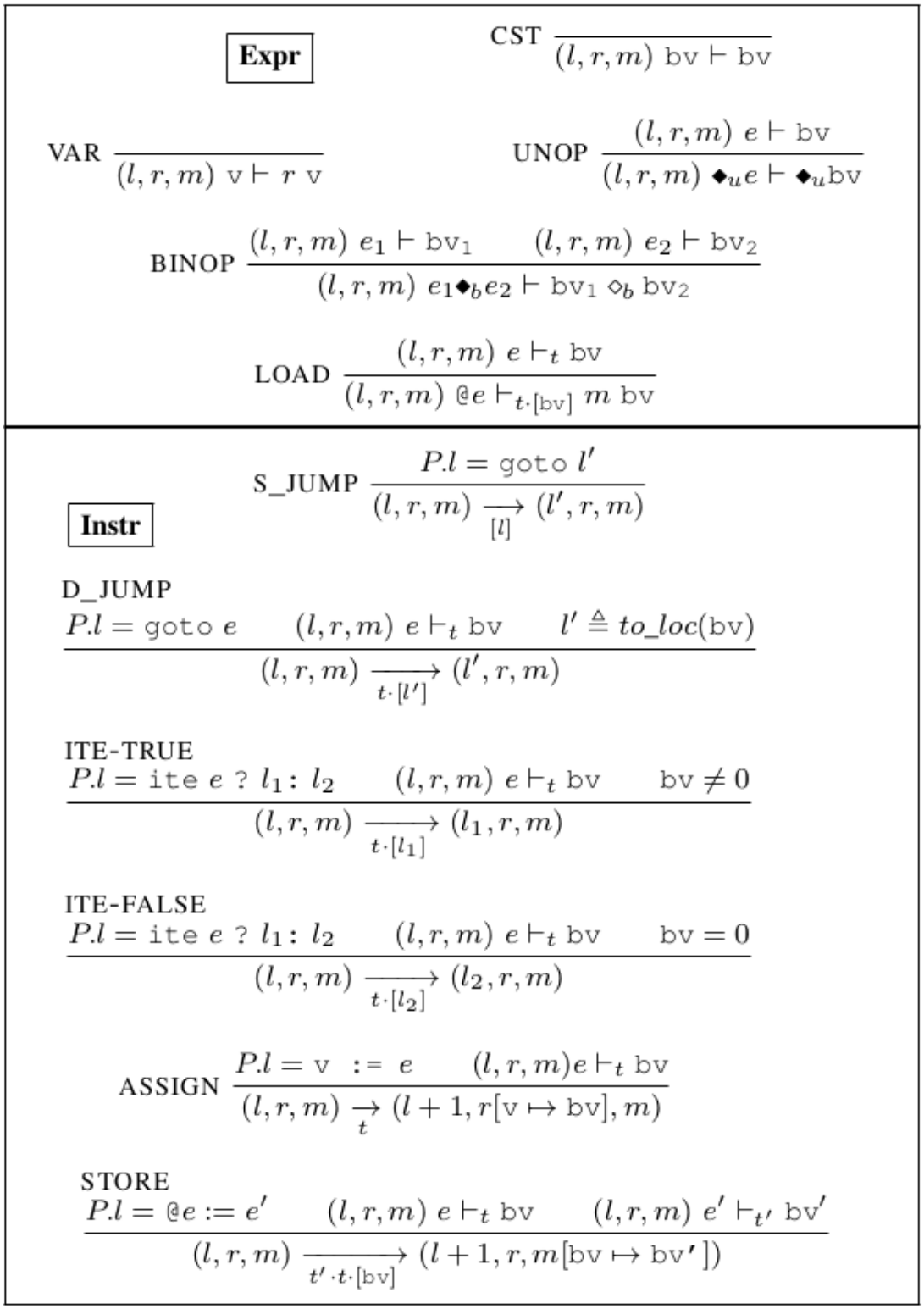


FIGURE 3.1 – Ensemble d'instructions définis formellement par [DBR19]

CHAPITRE 4

# Érysichthon, structure et exemples

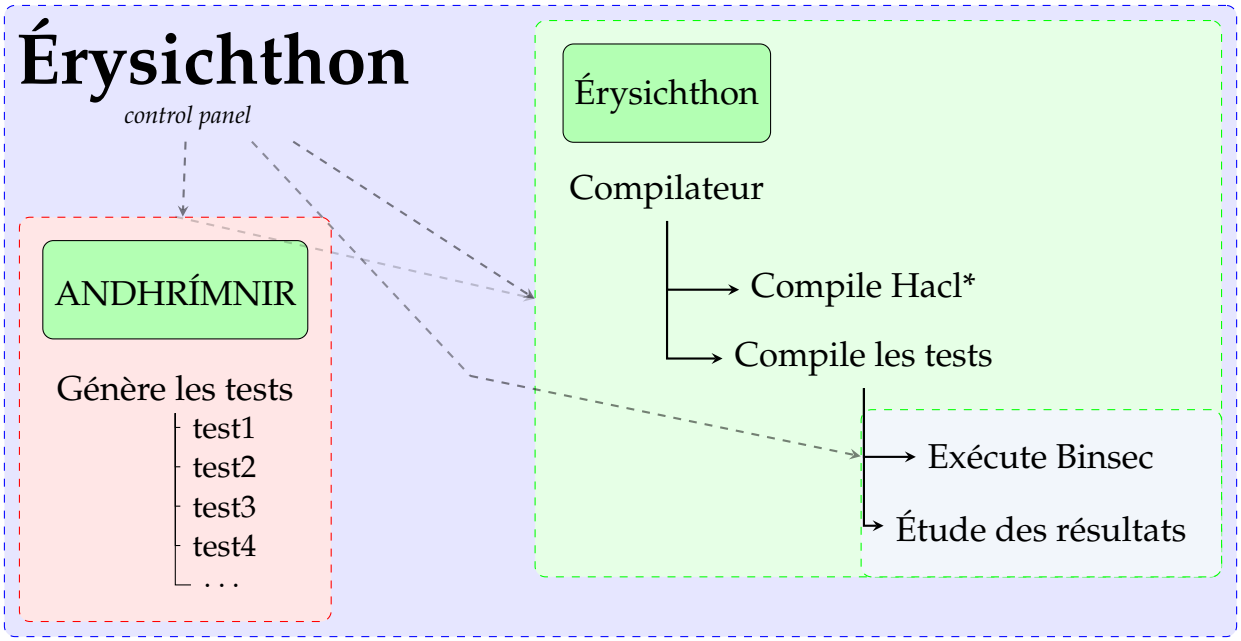


FIGURE 4.1 – Structure d'Érysichthon, schéma du point de vue de l'usager
Les flèches grises indiquent tous les éléments actionnables individuellement.

```c
/**
Encrypt a message `input` with key `key`.

The arguments `key`, `nonce`, `data`, and `data_len` are same in encryption/decryption.
Note: Encryption and decryption can be executed in-place, i.e.,
`input` and `output` can point to the same memory.

@param output Pointer to `input_len` bytes of memory where the ciphertext is written to.
@param tag Pointer to 16 bytes of memory where the mac is written to.
@param input Pointer to `input_len` bytes of memory where the message is read from.
@param input_len Length of the message.
@param data Pointer to `data_len` bytes of memory where the associated data is read from.
@param data_len Length of the associated data.
@param key Pointer to 32 bytes of memory where the AEAD key is read from.
@param nonce Pointer to 12 bytes of memory where the AEAD nonce is read from.
*/
void
Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt(
  uint8_t *output,
  uint8_t *tag,
  uint8_t *input,
  uint32_t input_len,
  uint8_t *data,
  uint32_t data_len,
  uint8_t *key,
  uint8_t *nonce
);
```

Code 1 – Déclaration de la fonction **encrypt** dans le fichier d'en-tête Hacl_AEAD_Chacha20Poly1305_Simd256.h

```json
{
"Meta_data":{
    "build" : "13-06-2025",
    "version" : "0.2.0"
}

,"Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt": {
    "*output":"BUF_SIZE"
   ,"*input":"BUF_SIZE"
   ,"input_len":"BUF_SIZE"
   ,"*data":"AAD_SIZE"
   ,"data_len":"AAD_SIZE"
   ,"*key":"KEY_SIZE"
   ,"*nonce":"NONCE_SIZE"
   ,"*tag":"TAG_SIZE"
   ,"BUF_SIZE":16384
   ,"TAG_SIZE":16
   ,"AAD_SIZE":12
   ,"KEY_SIZE":32
   ,"NONCE_SIZE":12
 }
}
```

Code 2 – Extrait du fichier Hacl_AEAD_Chacha20Poly1305_Simd256.json

```c
//
// Made by
// ANDHRÍMNIR - 0.5.4
// 12-08-2025
//

#include <stdlib.h>
#include "Hacl_AEAD_Chacha20Poly1305_Simd128.h"

#define BUF_SIZE 16384
#define TAG_SIZE 16
#define AAD_SIZE 12
#define NONCE_SIZE 12
#define KEY_SIZE 32
uint8_t output[BUF_SIZE];
uint8_t tag[TAG_SIZE];
uint8_t input[BUF_SIZE];
uint32_t input_len_encrypt = BUF_SIZE;
uint8_t data[AAD_SIZE];
uint32_t data_len_encrypt = AAD_SIZE;
uint8_t key[KEY_SIZE];
uint8_t nonce[NONCE_SIZE];


int main (int argc, char *argv[]){
Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt(output, tag, input, input_len_encrypt,
 data, data_len_encrypt, key, nonce);
   exit(0);
}
```

Code 3 – Code généré du fichier test Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.c

```
starting from core

secret global output, input, data, key, nonce, tag
replace opcode 0f 01 d6 by
zf := true
end
replace opcode 0f 05 by
  if rax = 231 then
    print ascii "exit_group"
    print dec rdi
    halt
  end
  print ascii "syscall"
  print dec rax
  assert false
end
halt at <exit>
```

Code 4 – Instruction Binsec générée automatiquement,
Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.ini