CHAPITRE 1

Analyse de programmes et méthodes de vérifications

Nous allons étudier les moyens à notre disposition pour réaliser une analyse pertinente et efficace d'un programme résistant aux attaques temporelles.

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 dont l'on souhaite se défendre. On parle parfois de *modèle de fuite*. Cette étape de synthèse et d'abstraction est importante pour identifier les risques encourus par le futur système, souvent en identifiant les points 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 modèles 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 matérielles permettant une défense contre ce modèle. Considérer un attaquant plus puissant, avec des accès à des ressources supplémentaires, potentiellement hypotétique, 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 connues, nous pouvons constituer un ensemble de règles qui vérifient 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 correcte 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'instructions binaire. Et une instruction est une action sur la mémoire. Cela nous permet de définir notre programme comme une suite de configurations (l, r, m) ; l la ligne d'instruction, r le dictionnaire de registre et m la mémoire. La configuration initiale est définie 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 registres 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 instructions. 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`

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

Nous pouvons retrouver l'ensemble des règles formelles en Annexe 5.1.

1.2 Analyse d'un programme

Nous avons conçu un modèle pour contrôler ou détecter les erreurs. Nous pouvons maintenant concevoir notre analyse pour vérifier ce modèle sur un programme. Plusieurs techniques de vérification 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 obtenus sont souvent approximés 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 obtenus et réduire le nombre de faux positifs nous pouvons 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 quelle entrée faible le programme ressort la même sortie faible peu importe les entrées hautes qui peuvent être précisées.

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

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 associée 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 employées pour que cette information se diffuse dans le compilateur et que des mesures adaptées soient effectuées au niveau du binaire. Cette approche est intéressante car elle permet une flexibilité plus importante lors de la production du code et permet de s'abstenir des contre-mesures décrites 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 approximée est sécurisée alors le programme est sécurisé. Cette approche se retrouve dans CacheAudit [Doy+13] : modélisation par un graphe de flot de l'état des caches, de la mémoire et des successions d'évènement.

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 utilisée 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 issues des évènements (accès mémoire, sauts,*etc*) rencontrés au fur et à mesure de l'exécution. Les approches diffèrent dans la collecte et la production de ces traces.

Trace unique Explorer tous les comportements d'un programme est coûteux en temps, et pour les besoins du développement il peut être préférable d'étudier quelques cas particuliers entièrement. Cette approche simplifie le modèle de l'attaquant et 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]).

1. Construction personnelle, le terme anglais est conservé.

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 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].

Ces modèles et ces approches permettent la mise en place d'outils d'analyse performants et innovants. Nous allons maintenant observer plus en détail leur fonctionnement.

CHAPITRE 2

Outils d'analyse et automatisme

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

Implémentations pour un usage industriel

Ce chapitre permet de présenter le raisonnement qui a motivé la conception d'un outil de détection automatique de failles par canal auxiliaire de type temporel.

3.1 Identification des besoins et spécificités

Nous avons pu voir grâce aux chapitres précédents que la conception et l'implémentation d'un système sécurisé est un problème difficile. Une première étape est de concevoir des primitives et des protocoles mathématiquement sécurisés. Une seconde étape est de s'assurer que leurs implémentations sont effectivement sécurisées, d'un point de vue :

— mathématique contre des attaques logiques (aspect fonctionnel : le code implémente correctement les bons concepts cryptographiques)
— matériel, contre des attaques très bas niveau (les attaques temporelles)

Avec l'objectif de concevoir un système sûr, il nous faut donc identifier toutes les tâches à réaliser pour arriver à bout de ce projet. En plus de ce travail de planification, l'identification et l'intégration d'outils déjà implémentés nous permettra d'avancer plus rapidement vers cet objectif.

Point de départ

En reprenant ces deux étapes, nous identifions les possibilités pour un développeur pour concevoir un système résistant à ces attaques temporelles.

La première étape de conception de primitives cryptologiques et de protocole n'est pas du ressort du développeur. Elle appartient aux cryptologues et aux chercheurs en sécurité mathématique. Ce sont eux qui conçoivent et maintiennent des librairies cryptographiques, des boîtes à outils qui proposent les briques de sécurité nécessaires aux systèmes sécurisés.

Plusieurs librairies existent [AHa98 ; Por16 ; Pol+20] et remplissent différents objectifs : rétrocompatiblité, politique temps constant, *etc*. Notre choix est à réaliser en fonction des spécificités du produits que nous cherchons à déployer.

La seconde étape est à distinguer en deux parties. Cette opération de vérification de la sécurité de l'implémentation peut être réalisée sur le produit fini et sur les librairies employés par le produit. Comme introduit, cette étape a pour objectif la vérification formelle du code du programme et la vérification matérielle au niveau assembleur.

Utiliser la bibliothèque **Hacl*** [Pol+20 ; Zin+17] permet d'avancer la première étape et la première partie de la seconde étape. Cette bibliothèque a été conçue formellement et vient avec les preuves mathématiques de la sécurité de son implémentation. Comme présenté en **??**, elle est programmée en F*. Le projet permet une exploitation en C et en assembleur [Zin+17].

En revanche, la seconde étape de la seconde partie nous demande une vérification au niveau de l'assembleur. Si certaine partie de cette librairie sont codées en assembleur, la majorité du projet reste du F* traduit vers C. Il faut réaliser une analyse. Dans le cadre de cette étude, l'outil d'analyse binaire retenu pour réaliser cette tâche est **Binsec**. Cet outil est implémenté en Ocaml et il est maintenu par une équipe de chercheurs et d'ingénieurs géographiquement proche de l'équipe PROSECCO Inria. Cet avantage permet des échanges plus directs et donc une facilité quant à la mise en place du projet.

L'objectif est donc d'analyser Hacl* dans son entièreté. Avec cette analyse complète, si elle est correcte, alors les deux étapes de réalisation d'un système sûr seront réalisées. Cela signifie qu'elle sera la première librairie cryptographique formellement sûre et vérifiée résistante aux attaques temporelles.

Objectifs à réaliser

Sans reprendre les explications du fonctionnement de Binsec, voir "ref vers fonctionement de Binsec", l'analyse se réalise sur un fichier binaire à l'aide d'instructions à adjoindre. Avec ce point de départ, nous commençons à construire notre carnet de spécifications.

Fichier binaire. Il faut donc des fichiers binaires à fournir à Binsec. Or comme chacun le sait, plus un binaire est imposant, plus son analyse est difficile. Et comme Binsec emploi l'analyse symbolique, explorer un binaire imposant a un coût de mémoire quadratique sur le parcours des instructions du binaire. L'idéal est donc d'analyser plein de petits fichiers binaires.

Analyse complète. Chaque fonction de Hacl* doit être analysée. En poursuivant la condition précédente, nous pouvons essayer de concevoir un binaire par fonction analysée. Nous distribuons ainsi l'analyse et nous parcourons ainsi toute les fonctions présentes dans la librairie.

Analyse correcte. En se rappelant comment fonctionne les optimisations (voir le tableau 5.1) il nous faut être attentif avec certaines qui simplifient/modifient le code. Pour éviter des suppressions d'instructions, le fichier nécessite une légère contextualisation avant l'appel de la fonction analysée.

```c
#include <stdlib.h>

#include "Hacl_AEAD_Chacha20Poly1305_Simd128.h"

#define BUF_SIZE 16384
#define KEY_SIZE 32
#define NONCE_SIZE 12
#define AAD_SIZE 12
#define TAG_SIZE 16

uint8_t plain[BUF_SIZE];
uint8_t cipher[BUF_SIZE];
uint8_t aead_key[KEY_SIZE];
uint8_t aead_nonce[NONCE_SIZE];
uint8_t aead_aad[AAD_SIZE];
uint8_t tag[16];

int main (int argc, char *argv[])
{
Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt
    (cipher, tag, plain, BUF_SIZE, aead_aad, AAD_SIZE, aead_key, aead_nonce);
exit(0);
}
```

Code 1 – Code d'analyse de la fonction Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt, testé lors de la prise en main de Binsec et Hacl*

De même, comme nos fichiers analysés appartiennent à la librairie extérieur Hacl*, l'emploi de l'option `-static` est nécessaire pour prévenir la mise place de liens vers la librairie

partagée dans le fichier binaire. Cette option ne nuit pas à la qualité de l'analyse, elle permet en revanche d'avoir tous les éléments sous la main lorsque nous désassemblons un fichier binaire. Retirer cette option lors de la compilation, c'est s'ajouter des lourdeurs et rallonger le temps requis pour la vérification manuelle d'un fichier.

Couverture de compilateur. Les travaux de SCHNEIDER et al. [Sch+24] ont clairement mis en évidence que le choix du compilateur est à considérer. Nous allons pouvoir identifier quel compilateur nous permet d'avoir le plus de fichiers binaires sécurisés. Cette analyse nous permet d'identifier les limites de la pratique de la programmation en temps constant.

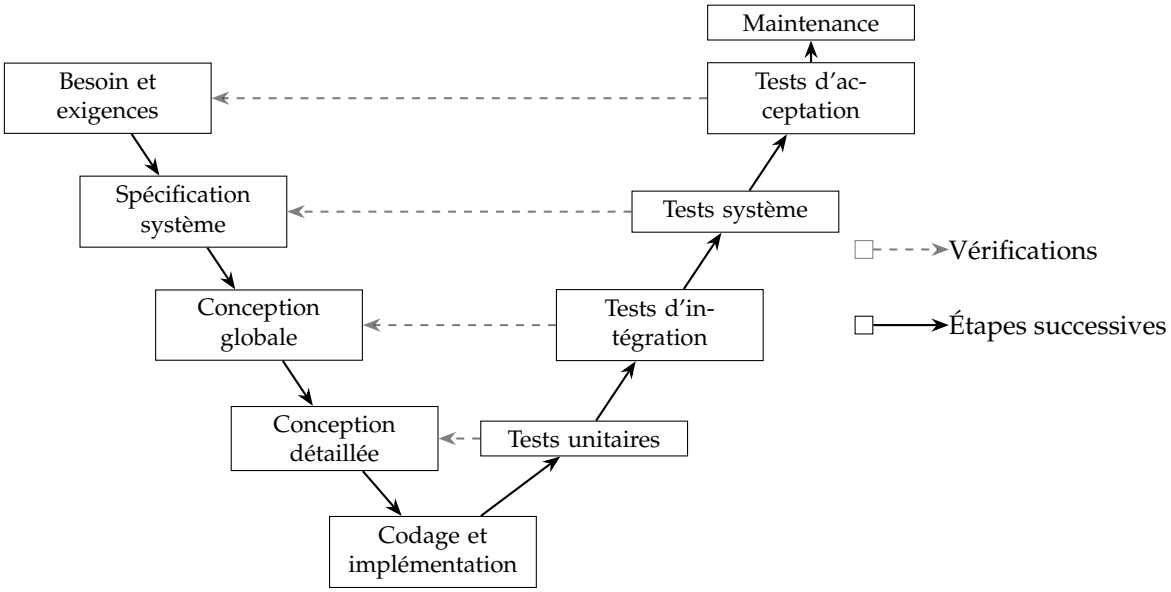
Couverture d'architectures. x86_64 et ARM sont les architectures matérielles les plus répandues dans le monde. Étendre l'analyse vers différentes plateformes et observer les différences qui émergent nous permettra d'avancer dans la direction de la conception d'une librairie cryptographique universelle. Nous pouvons aussi étendre cette analyse vers d'autres architectures comme PowerPC ou RiscV.

Automatisation. Faire cette analyse sur un fichier binaire, comme le code 1, avec trois axes de complexité (complétude, de la couverture d'architectures et des compilateurs) n'est pas envisageable à la main. Il faut absolument que cette analyse soit automatisée.

3.2 Initialisation et tests variés

Dans le cadre de la programmation sécuritaire, où sont développés les systèmes avec pour objectif d'un accident par siècle (métros automatiques, trains, avions…), les projets sont conçus selon le principe du cycle en V. Cette méthode au contraire de la méthode Agile permet de prévoir tous les cas de figure et d'usages, nous permettant de nous épargner les problèmes de correction de bogues.

FIGURE 3.1 – Cycle en V



Appliquer cette méthode à l'entièreté de ce projet n'est pas envisageable à cause du coût temporel qui est très élevé. Nous nous concentrons sur la réalisation d'une preuve de concept avec un produit minimal mais opérationnel. Le développement sera concentré sur l'objectif d'automatisation. Le développement d'outils permettant la réalisation des objectifs des couvertures nécessiteront un futur travail.

Identification des besoins et exigences

Nous avons déjà conçu notre carnet d'exigences. En revanche nous ne connaissons pas le comportement des outils que nous souhaitons employer. La première opération est donc de s'approprier le fonctionnement de ceux-ci. Le code 1 est un exemple de test réalisé dans cette phase du projet.

Binsec est un outil uniquement utilisable au travers d'un terminal. Il s'invoque avec son alias, le binaire à analyser et les options qui seront effectuées :

```
$ binsec -sse -sse-script $(BINSEC_SCRIPT) -checkct $(BINARY)
```

Code 2 – Commande Binsec basique

L'option `-sse` permet d'activer l'analyse par exécution symbolique, `-sse-script` associer à un fichier (ici `BINSEC_SCRIPT`) permet d'instruire notre analyse, préciser des stubs[1] et des initialisations. Enfin `-checkct` active la vérification de la politique temps constant au sein du fichier binaire indiqué par `BINARY`. Binsec renvoie dans le terminal le résultat de son analyse : [`secure, unknown, insecure`]. Le second est invoqué lorsque l'analyse est incomplète.

Cette phase «Test et Identification des exigences» permet de confronter plusieurs fonctions de Hacl* et de se familiariser avec le langage d'instructions qu'admet l'option `-sse-script`. Un tutoriel complet est accessible pour comprendre le fonctionnement l'outil Binsec depuis sa page officielle[2].

```
1  starting from core with
2    argv<64> := rsi
3    arg1<64> := @[argv + 8, 8]
4    size<64> := nondet            # 0 < strlen(argv[1]) < 128
5    assume 0 < size < 128
6    all_printables<1> := true
7    @[arg1, 128] := 0
8    for i<64> in 0 to size - 1 do
9      @[arg1 + i] := nondet as password
10     all_printables := all_printables && " " <= password <= "~"
11   end
12   assume all_printables
13 end
14
15 replace <puts>, <printf> by
16 return
17 end
18
19 reach <puts> such that @[rdi, 14] = "Good password!"
20 then print ascii stream password
21
22 cut at <puts> if @[rdi, 17] = "Invalid password!"
23
24 halt at <printf>
```

Code 3 – Instructions permettant de trouver le mot d'un passe d'un binaire exercice

Ce code présenté ici est un exemple d'usage de Binsec et permet de réaliser une attaque sur un binaire issu d'une plateforme d'apprentissage à la sécurité logicielle[3]. L'exercice consiste à retrouver le mot de passe caché d'un binaire. Dans le cadre de notre exercice d'analyse de la politique temps constant, le script 4 est plus simple.

1. Terme anglais du lexique de la rétro-ingénierie ; module logiciel simulant la présence d'un autre.
2. https ://binsec.github.io/
3. https ://crackmes.one/

Ce script a pour objectif de vérifier les résultats apportés par [Sch+24] concernant une fuite présente sur la fonction «*FStar_UInt64_eq_mask*» et d'étendre cette analyse vers d'autres architectures. Dans une première démarche d'automatisation, ce code a été généré automatiquement par un script shell. Nous pouvons voir que l'analyse ne parcourt pas l'entièreté du binaire, seulement 8 sections sont chargées (sur 24). L'analyse commence à l'appel de la fonction `main` et se termine à la ligne 8 avec une adresse de fin. Cette adresse de fin est produite par le script shell pour attraper la fin de la fonction `main`.

```
1  load sections .plt, .text, .rodata, .data, .got, .got.plt, .bss from file
2
3  secret global  r, cin, y, x
4
5  starting from <main>
6
7  with concrete stack pointer
8  halt at  0x0000000000000464
9  explore all
10
```

Code 4 – Instructions permettant d'analyser le code **??** compilé vers RiscV-32

Ce modèle, qui nous servira de base pour la suite du développement, a permis une analyse rapide entre différents compilateurs et différentes architectures.

Application et observation entre architectures et compilateurs

FIGURE 3.2 – Tableau de résultats d'analyse Binsec pour architecture ARMv7 et ARMv8

opt\fonction analysée	**cmovznz4**				
Clang+LLVM	**14.0.6**	**15.0.6**	**16.0.4**	**17.0.6**	**18.1.8**
-O0	✓	✓	✓	✓	✓
-O1	✓	✓	✓	✓	✓
-O2	✓	✓	✓	✓	✓
-O3	✓	✓	✓	✓	✓
-Os	✓	✓	✓	✓	✓
-Oz	✓	✓	✓	✓	✓

✓ : *binary secure*

Nous comprenons, à la lecture du tableau 3.2, que la politique temps constant est considérée respectée par Binsec sur les versions testées ainsi que pour les différentes options de compilation. Ce résultat est encourageant pour la suite du projet.

FIGURE 3.3 – Tableau de résultats d'analyse Binsec pour architecture Risc-V

opt\fonction analysée	**cmovznz4** - 64 bits		**cmovznz4** - 32 bits	
Compilateur et architecture	gcc 15.1.0	clang 19.1.7	gcc 15.1.0	clang 19.1.7
-O0	~	×	~	×
-O1	✓	×	✓	×
-O2	✓	×	✓	×
-O3	✓	×	✓	×
-Os	✓	×	✓	×
-Oz	✓	×	✓	×

✓ : *binary secure* ; ~ : *binary unknown* ; × : *binary insecure*

Les résultats dans le tableau 3.3 sont indéniables : la version 19.1.7 de clang rend le code source perméable à des attaques temporelles.

Identification de défaut

Pour construire le tableau 3.3, plusieurs alertes se sont levées et ont permis de mettre en évidence un bug présent dans Binsec. Cette erreur dans l'analyse symbolique provoquait l'arrêt de l'exploration par explosion de l'usage de la mémoire. Les registres `ld` (*load*) et `sd` (*store*) étaient mal gérés. En particulier l'opérande `ld`, simulé par un tableau, n'était jamais vidé. Cette découverte a amené un correctif et une amélioration de Binsec. De par l'envergure de ce projet, il est possible que d'autres erreurs dues à Binsec soient découvertes. L'exploration de nombreuses et nouvelles ISA[a], surtout avec Risc-V qui est encore en développement et perfectionnement, permet de renforcer cet outil plus efficacement et rapidement que par la conception de tests manuels.

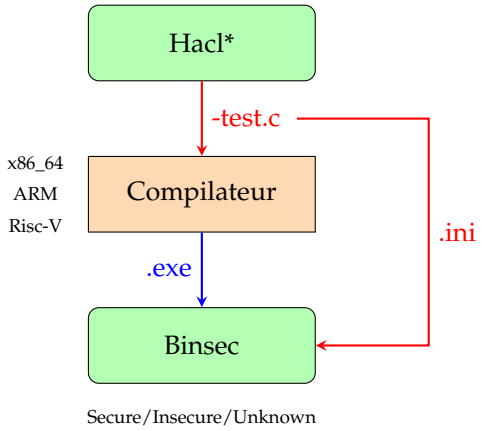
a. Acronyme anglais pour Architecture de Jeu d'Instruction, désigne l'ensemble des instructions assembleur associées à une architecture.

En explorant plus en avant le code binaire, nous découvrons que ces erreurs sont dues à l'opérande `beqz`[4]. L'ISA de Risc-V n'a pas à sa disposition un opérande comme `cmov` en X86_64 ou ARM. Donc l'application d'optimisation de compilation force l'usage de cette opérande qui n'est pas en temps constant. L'optimisation qui réalise ce changement se nomme «*InstCombinePass*».

Nous observons ici une manifestation indéniable des précédents résultats proposés par d'autres travaux de recherche. Une solution serait de modifier l'ISA pour permettre cette opération d'être en temps constant. Celle qui a été retenue, c'est d'employer un `pragma`, ici `# pragma clang optimise <off/on>`. Cette instruction, donnée dans le code source, indique au compilateur de désactiver ses optimisations pour le code contenu entre les deux balises `off`, `on`. Cette solution entraîne des pertes de performance et des ralentissements quant au temps de compilation et à l'usage des ressources. Il est donc préférable de l'utiliser avec parcimonie.

Après avoir ciblé notre besoin, les exigences associées et effectué des tests pour comprendre le processus à automatiser, nous pouvons synthétiser la démarche avec la figure 3.4 : depuis Hacl*, nous extrayons une fonction qui sera testé, nous fabriquons le fichier de test en C ; nous identifions les paramètres secrets et nous concevons le script adéquat pour Binsec ; nous compilons le fichier C à notre guise et nous terminons par l'analyse Binsec.

FIGURE 3.4 – Flot de travail de l'outil d'analyse à concevoir



Nous allons maintenant nous pencher sur le procédé de conception de notre outil de détection automatique de failles temporelles.

4. Effectue un branchement si la valeur du registre consulté est zéro. Cette opérande est propre à Risc-V.

CHAPITRE 4

Érysichthon à jamais affamé

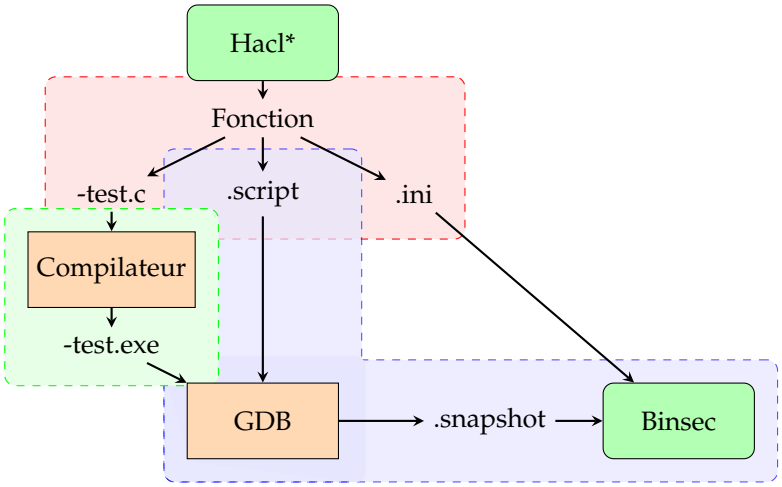
intro

4.1 Planification et préparations

Nous avons nos spécificités techniques et nous savons qu'elle forme notre outil doit avoir (fig **??**). Nous pouvons commencer par synthétiser les opérations nécessaires.

Nous allons donc concevoir des protocoles pour identifier les étapes nécessaires pour que Binsec analyse entièrement un fichier et nous renvoie un parmi [`secure, unknown, insecure`]. Le protocole `x86_64` est particulier. Depuis la version **0.5.0** de Binsec il est possible de fournir un «cliché mémoire» [1] pour accélérer l'analyse. Nous utilisons cet avantage pour l'intégrer à notre graphe d'éxécution. La machine sur laquelle le projet sera développé est sur une architecture x86_64, cela nous permet d'utiliser l'outil GDB pour la génération de cliché mémoire.

FIGURE 4.1 – Protocole pour analyser des fichiers compilés en x86_64

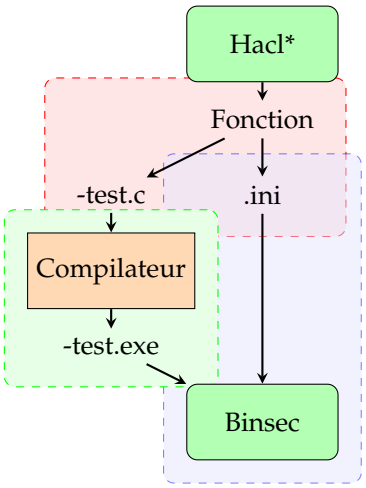


Ce graphe modélise la chaîne d'étapes nécessaire à l'obtention d'une analyse Binsec pour une fonction que nous ciblons. Plusieurs zones sont distinguées. La zone verte correspond à l'étape de compilation, la zone bleue à l'étape de préparation de l'analyse et la zone rouge à la synthèse de fichiers (de tests et d'instruction pour l'analyse de Binsec). Ce choix de couleur est adapté à la difficulté attendue de chaque étape. L'opération de compilation consiste en une commande. L'opération de préparation d'analyse consiste simplement en deux commandes : un appel à GDB avec le binaire puis un appel à Binsec avec le cliché mémoire et les instructions d'analyse.

1. Plus couramment 'Core dump', terme technique anglais désignant une copie de la mémoire vive et des registres d'un programme. Ce fichier sert à être analysé, généralement par un débogueur.

Avec ce graphe réalisé, nous pouvons le modifier pour préparer la voie à d'autres architectures. Dans un format plus générique voici comment se présente le protocole d'analyse :

FIGURE 4.2 – Protocole générique d'analyse



Dans ce contexte, une question se pose : est-ce que la conception des scripts pour Binsec (*.ini*) est automatisable ou est-ce qu'il faudra utiliser des émulateurs pour générer des clichés mémoire et revenir dans le cas de la figure 4.1 ?

En effet, l'importance de cette question se révèle lorsque nous changeons d'architecture et que nous devons nous passer de clichés mémoire. Sur notre machine en x86_64, si nous analysons un fichier compilé en ARM, alors nous pouvons rencontrer des appels à des fonctions systèmes : les `IFUNC`. Or la résolution de ces fonctions est gérée dynamiquement lors de l'exécution du programme. Cette mécanique permet d'utiliser des implémentations optimisées en fonction des configurations du système d'éxécution. Or comme Binsec réalise une analyse symbolique du programme, il faut lui spécifier quelles fonctions correspondent aux `IFUNC` qu'il peut croiser. pour illustrer ce point, observons le script nécessaire pour une vérification de la fonction «Hacl _AEAD _Chacha20Poly1305_Simd128_encrypt» compilé vers ARMv8.

```
load sections .plt, .text, .rodata, .data, .got, .got.plt, .bss from file

secret global input1, aad1

@[0x00000048f008 ,8] := <__memcpy_generic>
@[0x00000048f018, 8] := <__memset_generic>
@[0x00000048f030 ,8] := <__memcpy_thunderx2>

starting from <main>
with concrete stack pointer


halt at <exit>
explore all
```

Code 5 – Script d'instruction pour analyser un binaire compilé vers ARM

Les lignes 5 à 7 sont présentes pour indiquer les branchements à effectuer par Binsec lorsqu'il rencontre ces adresses. Cette opération automatiquement exécutée lors de l'initialisation de l'exécution doit ici être précisée avec les fonctions présentes dans le binaire. Automatiser ces affectations peut être difficile et nécessiter quelques outils d'analyse supplémentaires pour attraper les adresses qui ont besoin d'être réaffectées et leur attribuer les fonctions les plus adaptées.

Nommer un outil

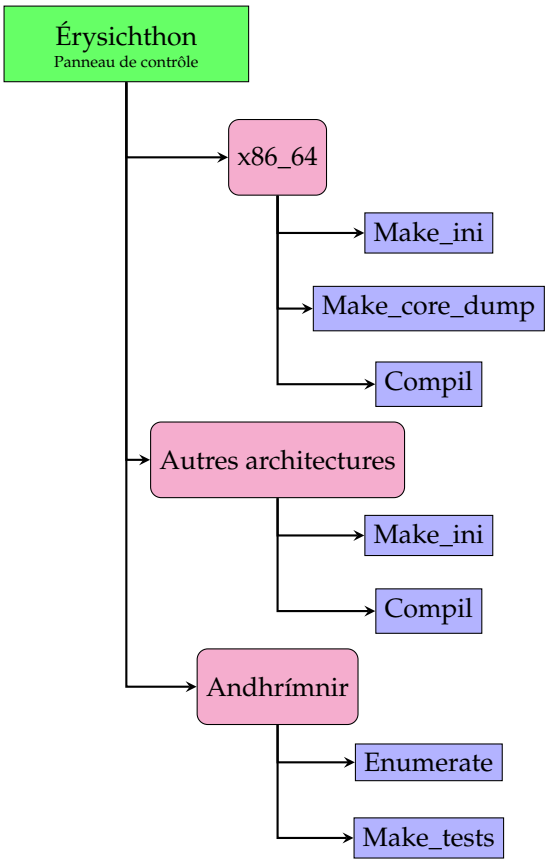
Rapidement il a fallu trouver un nom pour ce projet, l'appeler par "Notre outil… " devenait lourd et redondant entre les réunion hebdomadaire. En revanche trouver LE nom adéquat n'est pas une chose aisée, il peut être dû à une blague, une référence ou plus simplement lié au sens du projet. Dans notre cas, nous aimons la mythologie et le travail réalisé peut se résumer à "faut donner à manger à Binsec".

Érysichthon est un personnage de la mythologie grecque condamné à être affamé au point de se dévorer lui-même pour avoir détruit l'idole d'un dieu. Ce nom me plaît et sera retenu pour la suite du projet.

Conception d'Érysichthon

Nous avons vu les protocoles nécessaires pour construire une analyse complète. Nous avons fait des tests pour comprendre le fonctionnement de Binsec et comment doivent être déclarées les fonctions de Hacl*. Nous passons donc en phase de conception et construisons notre outil *Érysichthon*. Il sera une combinaison de script Python, script shell et de Makefile. Nous appelons module un ensemble de script qui réalise une tâche au sein d'*Érysichthon*. Nous vous présentons en figure **??** comment s'organisent les modules et les tâches qu'ils effectuent.

FIGURE 4.3 – Structure des modules d'Érysichthon



Nous retrouvons les différentes étapes de nos protocoles représentées par des rectangles symbolisant les modules associés depuis leurs noeuds respectifs : «`Make_ini`» pour la génération des scripts pour Binsec, «`Make_core_dump`» pour la génération des clichés mémoires et «`Compil`» pour les appels aux compilateurs. Le dernier noeud «`Andhrímnir`» est particulier et détaillé dans la section suivante.

Andhrímnir

Ce module de Érysichthon est particulier car il est lui aussi baptisé. Ce module consiste à produire les fichiers qui seront compilés puis analysés par Binsec. Son nom est celui du

cuisinier des dieux de la mythologie nordique, un travail répétitif et quotidien qu'il réalise ici pour notre outil.

Ce module est nommé car il constitue un projet dans le projet, sa conception seule a pris plus de la moitié du temps de développement total. C'est un outil qui, à partir d'un projet en C, est capable de générer automatiquement des tests qui compilent et peuvent ensuite être proposé à des outils d'analyse binaire. Ce module est aggrégé à Érysichthon mais peut être porté vers d'autre projets. À la différence des logiciels qui produisent des tests unitaires (uniquement sur des projet Java, Haskell ou C# et souvent associé à des offres payantes), il y a ici une garantie quant à la complétude des tests produit. Toutes les fonctions présentes dans le projet C analysé auront un test associé.

Ce module, comme son grand frère, est fonctionnel et abouti. En revanche, il nécessite quelques opérations manuelles supplémentaires et quelques améliorations pour pouvoir supporter n'importe quel projet C. Additionnellement il possède quelques optimisations propres à Hacl* permettant d'accélérer la mise en service d'Érysichthon.

4.2 Conception et usages

Nous commençons par le petit frère, Andhrímnir. Il fonctionne avec une phase d'initialisation «`Enumerate`» et une phase de production de tests «`Make_tests`», elle-même découpée en plusieurs étapes. La génération de **548** fichiers de tests est réalisé en moins de deux secondes.

Enumerate

Cette étape, de réalisation très simple, consiste à identifier toutes les fonctions qui auront un fichier test généré. Comme Hacl* génère automatiquement son code C, nous exploitons cette particularité pour lister efficacement nos fonctions. L'opération actuellement réalisée est de lister l'ensemble des fichiers "**.h**" contenu dans le répertoire cible. Ensuite un parcours et une lecture de ceux-ci nous donne toutes les fonctions de l'API d'Hacl* et d'avoir une couverture complète du projet.

C'est lors de cette étape que nous spécifions les fonctions à tester (ou nous pouvons aussi retirer des fonctions de la chaîne de production). Ce garde-fou permet d'accélérer l'obtention des résultats finaux et d'aider grandement lorsque nous souhaitons déboguer.

Make_test

Le modèle de fichier que nous construisons est similaire aux tests minimaux préalablement réalisés. Des paramètres, une déclaration de fonction et un appel à la fonction `exit`, c'est notre recette pour une analyse simple. Binsec réalise une analyse symbolique, il ignore donc la valeur réelle des entrées. Notre objectif avec notre recette est de concevoir un test qui compile et qui contient toutes les instructions assembleurs qui pourront être analysées. L'exemple 4.4 illustre comment sont générés nos fichiers de tests.

Une première partie initie le fichier. Cette partie contient les appels inclusifs de la bibliothèque standard C, l'invocation de la bibliothèque Hacl* au travers du fichier d'en-tête de référence (ici `Hacl_EC_K256`) et la signature de fabrication en commentaire. L'utilisation de la bibliothèque standard permet d'utiliser la fonction `exit`. Avec cet appel, nous construisons nos scripts Binsec avec une interruption sur cette fonction. Cet arrêt précoce permet d'accélérer l'analyse du binaire de la cible (ici `Hacl_EC_K256_felem_sqr`) et nous garantit que cette analyse soit complète.

La deuxième partie contient tous les éléments déclaratifs nécessaires à l'invocation de la fonction. Puis se termine avec le corps du fichier C qui contient l'appel de la fonction, notre balise de fin avec la fonction `exit`. Cette construction est standardisée entre les fichiers et permet de mettre en place quelques optimisations.

Comme illustré par la figure 4.5, la génération des tests est effectuée lors de la lecture des fichiers d'en-tête. Une phase de lecture d'un fichier de données "json" est ensuite réalisée pour avoir toutes les informations nécessaires à la constitution du fichier test. Une fois ces étapes réalisées, Andhrímnir commence sa préparation. Or il se trouve que parfois, certaines fonctions de Hacl* font appel à des structures propres à la bibliothèque qui ont une instanciation particulière. Le module «`Détection appel auxiliaire`» permet de vérifier ce cas de figure.

FIGURE 4.4 – Test de la fonction Hacl_EC_K256_felem_sqr

```
//
// Made by
// ANDHRÍMNIR - 0.3.0
// 09-07-2025
//

#include <stdlib.h>
#include "Hacl_EC_K256.h"
```

Phase introductive : 8 lignes

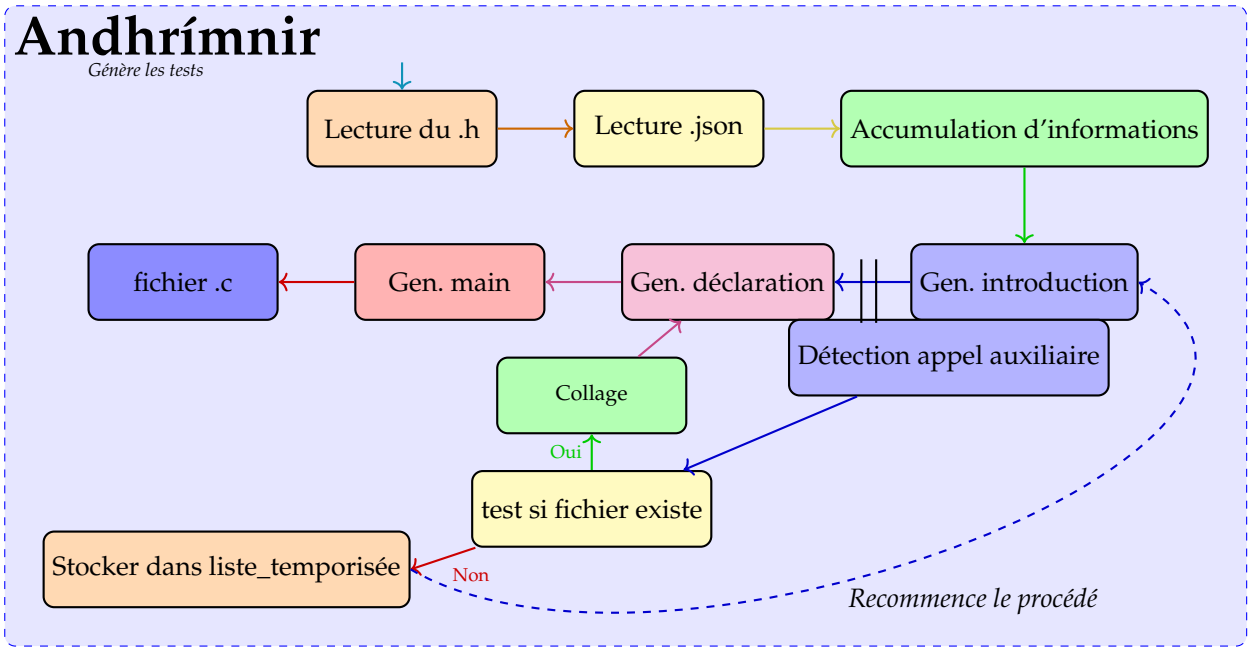
```
#define BUFFER_SIZE 5
uint64_t a[BUFFER_SIZE];
uint64_t out[BUFFER_SIZE];
```

Phase déclarative

```
int main (int argc, char *argv[]){
Hacl_EC_K256_felem_sqr(a, out);
  exit(0);
}
```

Phase principale

FIGURE 4.5 – Schéma de conception d'Andhrímnir



Dans le cas où aucun appel n'est détecté, Andhrímnir continue sa préparation avec les étapes successives illustrées par la figure 4.4 : génération des déclarations puis génération du `main`.

À l'inverse où un appel est détecté, il est possible que la fonction soit déclarée dans un autre fichier d'en-tête. Si c'est le cas, alors Andhrímnir doit déterminer quel fichier contient les informations requises pour compléter les informations nécessaires pour produire un fichier de test correct. La solution qui nous est venue est de temporiser le problème. Andhrímnir prépare des tests pour toutes les fonctions. Donc s'il a besoin d'une fonction qu'il a déjà préparé, nous pouvons accéder aux informations contenues dans le fichier de test associé. Au contraire, s'il a besoin d'une fonction qu'il n'a pas encore préparé, alors il peut la mettre de côté et retravailler dessus une fois qu'il a fini son premier passage sur toutes les fonctions d'Hacl*. Ce procédé est récursif pour pallier le problème d'appels

en cascade.

En réalité Andhrímnir ne recharge pas les informations d'une fonction dont il a besoin, il effectue cette opération de «`collage`». Elle consiste à une instruction shell qui vient ajouter (coller) au fichier en cours de conception la partie déclaration du fichier. Cette astuce permet d'éviter une nouvelle étape d'accumulation d'informations.

La phase de lecture dans les fichiers de données json existe afin d'accélérer le développement et la mise en service d'Andhrímnir. Cela permet de venir ajouter manuellement des instructions de haut niveau pour la conception des tests. Le code en annexe 6 illustre ce point : certaines fonctions ont besoin que les paramètres déclarés respectent certaines conditions. Cet exemple est accompagné du fichier json associé et du fichier de test final **??**.

Make_core_dump et Compilation

Les opérations de production de clichés mémoire et de compilation sont un assemblage de commandes shell et script pour GDB qui sont concevables sans problèmes. L'élément difficile à cette étape est la compilation de la bibliothèque Hacl*. Cette étape est nécessaire pour correctement compiler nos fichiers tests qui appellent Hacl*. Or cette gestion de la compilation est réalisée par le projet Hacl* lui-même et a besoin d'être améliorée pour permettre une compilation croisée vers d'autres architectures.

Une modification du script de compilation «`configure`» a été proposé et modifié sur le dépôt officiel du projet Hacl*.

Make_ini

Ce module consiste à concevoir les fichiers d'instructions pour Binsec. Il doit spécifier les variables secrètes associées à la fonction analysée. À la suite des exemples cités précédemment, le code 9 illustre comment ces instructions s'organisent. Il est adapté pour l'architecture x86_64 et exploite la mécanique des clichés mémoires.

Un premier temps initie le chargement des données, ensuite l'étiquette «`secret`» est accrochée aux variables à suivre durant l'analyse. Des commandes de gestion d'instructions particulières : des appels systèmes, des vérifications de registres inconnus de Binsec ; permettent que l'analyse ne s'interrompe pas et nous donne un résultat pertinent (`secure`, `insecure`). Enfin nous indiquons notre arrêt d'exploration sur la fonction «`exit`» et nous donnons notre feu vert avec la commande d'exploration totale.

Dans le cadre d'autres architectures, comme ARM, le code 5 montre que la différence à considérer est cette affectation manuelle des «`IFUNC`». Pour le moment, la solution en place qui gère une affectation correcte est conçue en fonction du support matériel sur lequel l'outil est activé.

4.3 Résultats

L'éxécution c'est réalisé sur une machine équipée d'un processeur *Intel Xeon E5-2620v4* avec 32 Gio de mémoire. Le temps nécéssaire pour une analyse complète en x86_64 avec `GCC` est de 4h07. Une analyse complète comprend la compilation de Hacl*, la génération des fichiers de tests, la compilation des-dits fichiers et l'analyse individuelle de chacun par Binsec. Nous avons ajouter un module pour synthétiser et générer des rapports d'analyse.

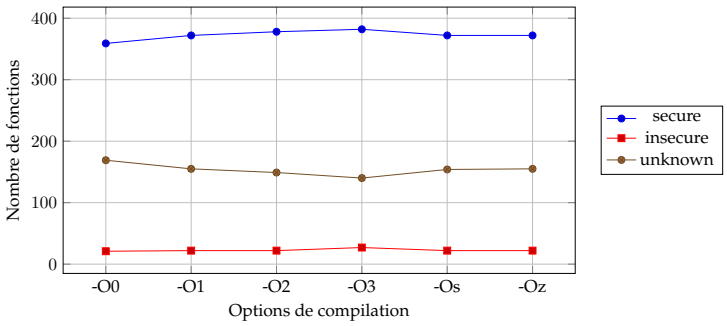


FIGURE 4.6 – Graphes des résultats d'Érysichthon en x86_64

Les données détaillées peuvent être consulté en annexe 6.1. En l'état, l'analyse rapporte entre 139 et 168 fichiers dont l'analyse n'a pus se terminer. Il faudrait observer plus en détails ces fichiers pour connaître les causes de ces arrêts.

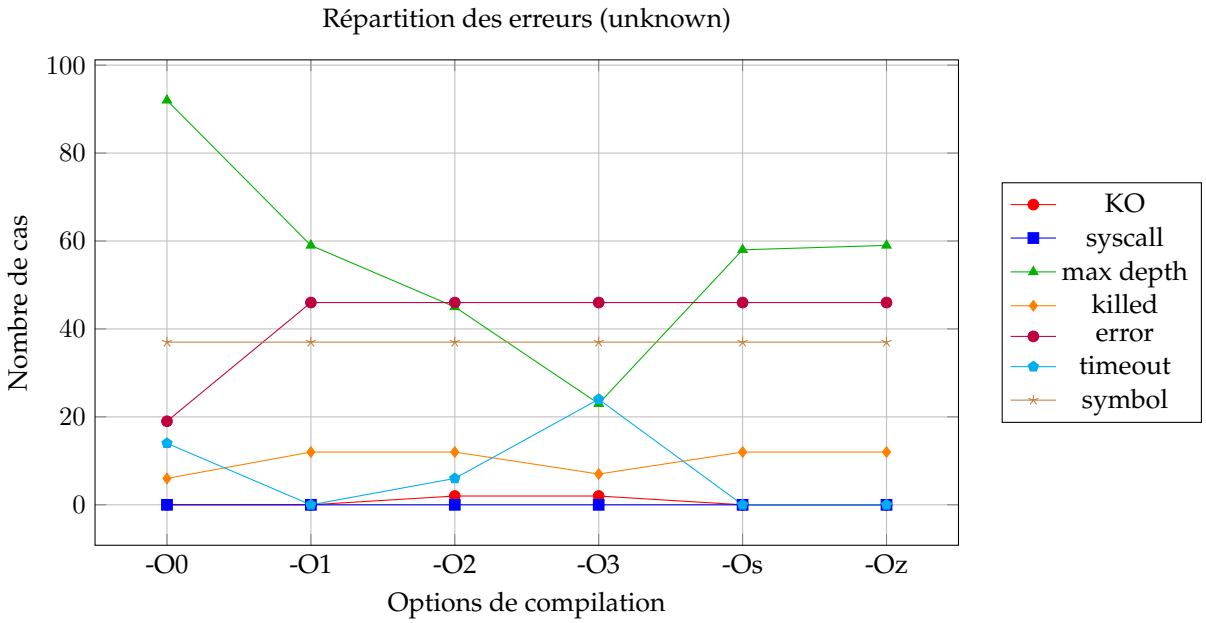


FIGURE 4.7 – Graphes détaillant les erreurs interrompant l'analyse Binsec

Le détails des valeurs est rapporté en annexe 6.2.

On peut voir avec cette figure 4.7 que les erreurs sont dues à :

`max depth` Arrêt par limitations du nombre d'instruction à analyser, cela permet de réduire la profondeur des branchement conditionnels à explorer et limiter le risque de parcour infini.

`timeout` Comme le précédent, limitation par le temps.

`error` Instruction inconnue de Binsec, il a besoin que le script d'instruction soit corrigé.

`symbol` Comme le précédent, mais peut-être que le fichier de test a besoin d'être modifié.

`KO` Instruction inconnue de Binsec, il a besoin d'être amélioré.

`killed` Consommation excessive des ressources, processus terminé.

Ces résultats indiquent les corrections à apporter à Érysichthon.

Sécurité de Hacl*

Revenons sur les resultats présentés sur la figure 4.6 et ignorons les fonctions marquées `unknown`. Les fichiers non sécurisés sont les plus nombreux avec l'option `-O3` (27) et le moins avec l'option `-O0` (21). Cette expérimentation remet en évidence les travaux de SCHNEIDER et al. mais nous pouvons nous intéroger quant à la réalité des fuites présentés ici. Comme nous analysons toute la bibliothèque Hacl* certaines fonctions peuvent être non sécurisées mais cela n'impacte pas la sécurité globale car elles ne sont besoin d'être sécurisé

CHAPITRE 5

Références

TABLE 5.1 – 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

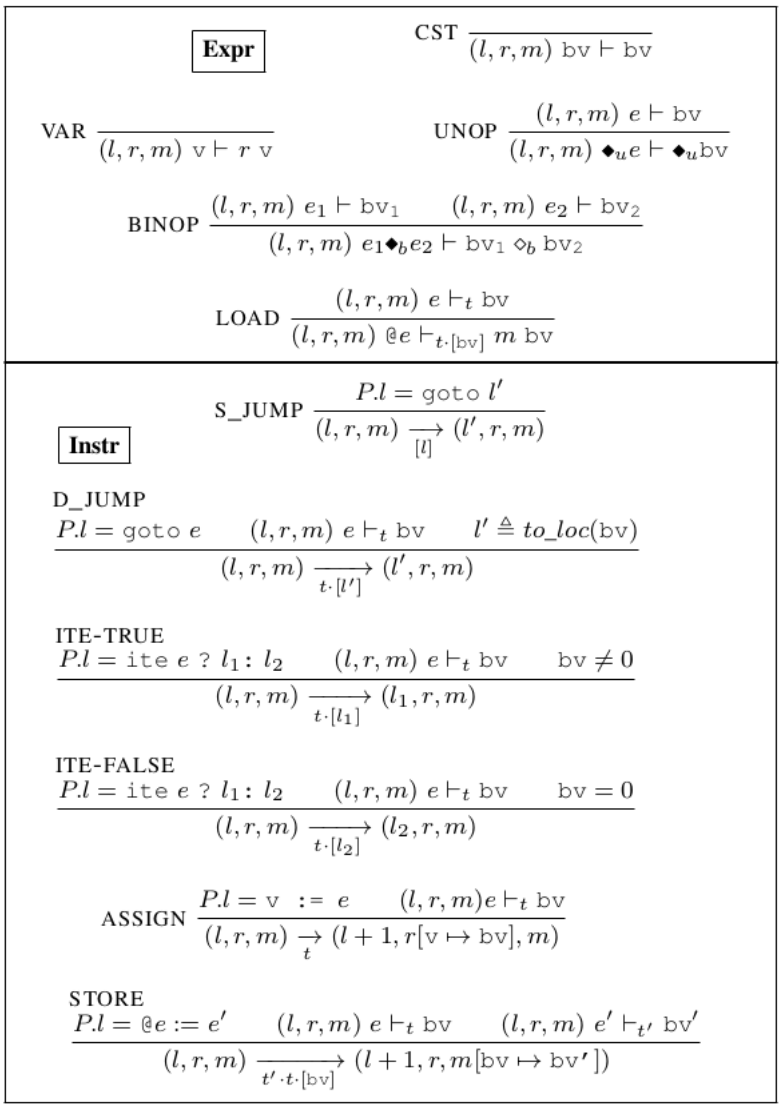


FIGURE 5.1 – Ensemble d'instructions définis formellement par [DBR19]

CHAPITRE 6

Érysichthon, structure et exemples

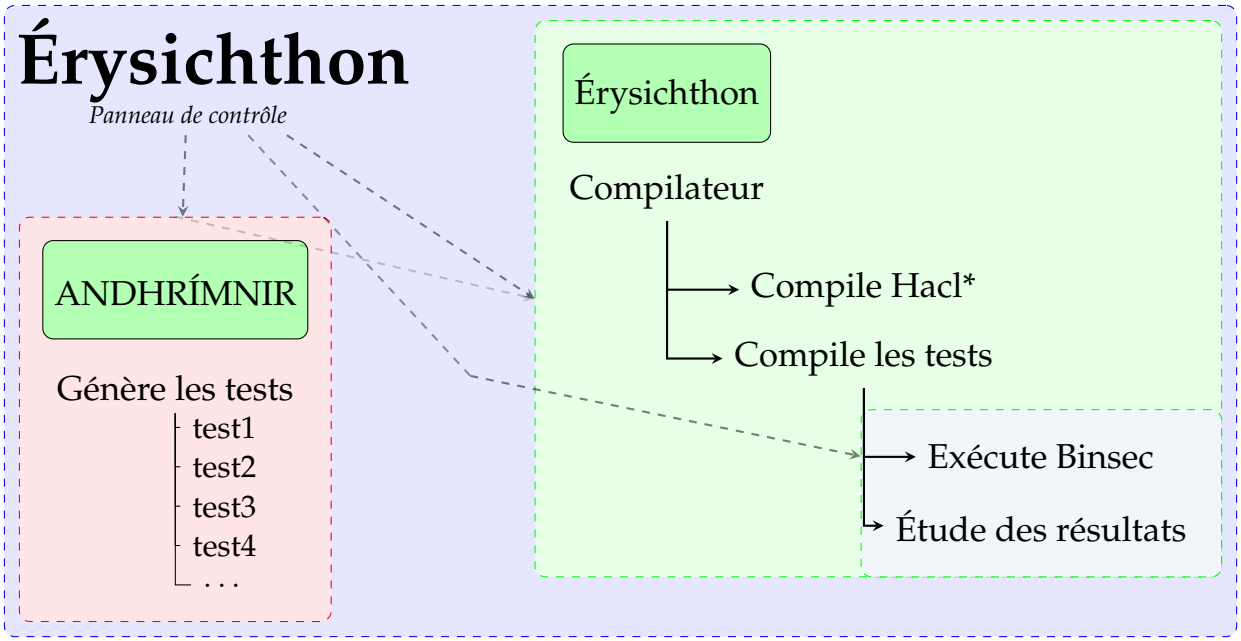


FIGURE 6.1 – Structure d'Érysichthon, schéma du point de vue de l'usager
Les flèches grises indiquent tous les éléments actionnables individuellement.

```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 6 – 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 7 – 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 8 – 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 9 – Instruction Binsec générée automatiquement, Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.ini

Optimisation	Secure	Unknown	Insecure
-O0	359	168	21
-O1	372	154	22
-O2	378	148	22
-O3	382	139	27
-Os	372	154	22
-Oz	373	153	22

TABLE 6.1 – Résultats d'Érysichthon en x86_64

Erreur / Option	-O0	-O1	-O2	-O3	-Os	-Oz
KO	0	0	2	2	0	0
syscall	0	0	0	0	0	0
max depth	92	59	45	23	58	59
killed	6	12	12	7	12	12
error	19	46	46	46	46	46
timeout	14	0	6	24	0	0
symbole	37	37	37	37	37	37

TABLE 6.2 – Tableau détaillant les erreurs interrompant l'analyse Binsec

Fonctions	Options concernées					
Hacl_EC_Ed25519_point_eq	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_ecdh	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_ecdsa_verify_hashed_msg	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_ecdsa_verify_sha256	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_is_public_key_valid	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_public_key_compressed_from_raw	O0					
Hacl_K256_ECDSA_public_key_uncompressed_to_raw	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_is_signature_normalized	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_signature_normalize	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_verify_hashed_msg	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_verify_sha256	O0	O1	O2	O3	Os	Oz
Hacl_NaCl_crypto_box_open_detached_afternm	O0	O1	O2	O3	Os	Oz
Hacl_NaCl_crypto_secretbox_open_detached	O0	O1	O2	O3	Os	Oz
Hacl_P256_compressed_to_raw	O0					
Hacl_P256_dh_responder	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_p256_sha2	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_p256_sha384	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_p256_sha512	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_without_hash	O0	O1	O2	O3	Os	Oz
Hacl_P256_uncompressed_to_raw	O0	O1	O2	O3	Os	Oz
Hacl_P256_validate_public_key	O0	O1	O2	O3	Os	Oz
Hacl_FFDHE_ffdhe_shared_secret_precomp		O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_sign_hashed_msg		O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_sign_sha256		O1	O2	O3	Os	Oz
Hacl_NaCl_crypto_box_beforenm				O3		
Hacl_NaCl_crypto_box_detached				O3		
Hacl_NaCl_crypto_box_easy				O3		
Hacl_NaCl_crypto_box_open_detached				O3		
Hacl_NaCl_crypto_box_open_easy				O3		

TABLE 6.3 – Détails des fonctions non sécurisées en fonction des optimisations entrées, éxecution d'Érysichthon en x86_64