CHAPITRE 1

Implémentations pour un usage industriel

intro

1.1 Identification des besoins et spécificités

On a pus voir grâce aux chapitres précédents que la conception et l'implémentation d'une système sécurisé est un problème difficile. Une première étape est de concevoir des primitives et protocoles mathématiquement sécurisés. Une seconde étape est de s'assurer que leurs implémentations sont effectivement sécurisées, d'abord d'un point de vue mathématique contre des attaques logiques (aspect fonctionnel : le code implémente correctement les bons concepts cryptographiques), mais aussi contre des attaques très bas niveau, les attaques temporelles.

Avec l'objectif de concevoir un sytème 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 implémenté nous permettra de d'avancer plus rapidement vers cet objectif.

Point de départ

En reprenant ces deux étapes, on va identifier quels sont nos leviers et nos possibilités pour un développeur pour avancer dans la conception de notre graal.

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 concoivent et maintiennent des librairies cryptographiques, des boîtes à outils qui propose les briques de sécurité nécessaires aux systèmes sécurisés.

Plusieurs librairies existent [AHa98 ; Por16 ; Pol+20] et remplissent différents objectifs : rétro-compatiblité, politique temps constant, *etc*. Notre choix est à réaliser en fonction des spécifités du produits que l'on cherche à déployer.

La seconde étape est à distinguer en deux parties. Cette opération de vérification de la sécurité de l'implémentation peut-être réaliser sur le produit fini et sur les librairies employés par le produit. Comme introduit, cette étape à 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 mathématique de la sécurité de son implémentation. Comme présenté en **??**, cette librairie 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**. Cette outil est implémenté en Ocaml et est maintenu par une équipe de chercheurs ingénieurs géographiquement proche de l'équipe PROSECCO Inria. Cet avantage permet des échanges plus directs et donc une facilité quand 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 que la première librairie cryptographique formellement sûre et résistante aux attaques temporelles sera conçus.

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'un carnet d'instructions à préciser. Avec ce point de départ, on peut commencer 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, on peut essayer de concevoir un binaire par fonction analysé. On distribue ainsi l'analyse et on parcourt ainsi toute les fonctions présentent dans la librairie.

Analyse correcte. Si on se rappel comment fonctionne les optimisations (voir le tableau 3.1) il faut faire attention avec certaines optimisations qui simplifient le code par soustraction d'oprations. Le fichier ne doit pas seulement contenir un appel de fonction, il faut une légère mise contexte.

```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'anlayse de la fonction Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt, testé lors de la prise main de Binsec et Hacl*

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

tagée dans le ficher 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 l'on déassemble un fichier binaire. Retirer cette option lors de la compilation, c'est se rajouter des lourdeurs et rallonger la 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. Il faut donc identifier quel compilateur nous permet d'avoir des fichiers binaires les plus sécurisés. On peut aussi identifier quels optimisations produisent la rupture de sécurité dans le binaire en étudiant plus en avant le comportement de ceux-ci.

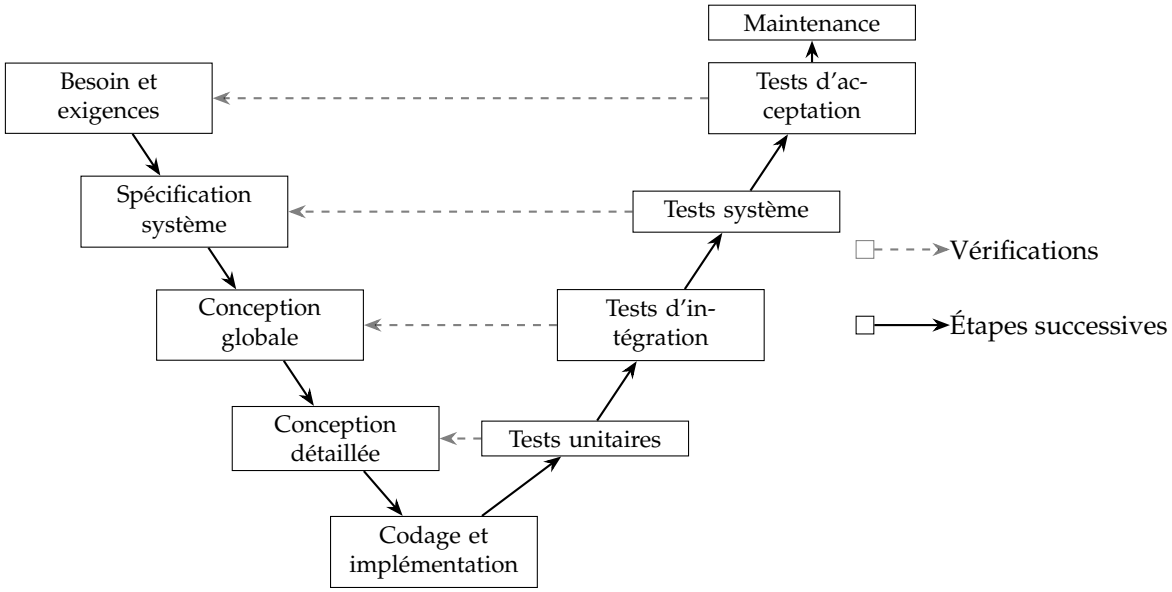
Couverture d'architectures. x86_64 et ARM sont les architectures matérielles les plus répandues dans le monde. Étendre l'analyse vers différentes plateformes et observer les différences qui émergent nous permettraient d'avancer dans la direction de la conception d'une librairie cryptographique universelle. On aussi étendre l'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 compléxité (complétude, de la couverture d'architectures et des compilateurs) n'est pas envisageable à la main. Il faut absolument que cette analyse soit automatisée.

1.2 Initialisation et tests variés

Dans le cadre de la programmation sécuritaire, où sont développés les systèmes avec pour objectif de un accident par siècle (métros automatiques, trains, avions…), les projets sont conçus selon le principe du cycle en V. Au contraire de la méthode Agile où on avance vers les problèmes en les résolvants au fur et à mesure, avec ce principe le développement est beaucoup plus long mais permet d'esquiver les problèmes qui, dans son contexte d'usage, peuvent entraîner des décés.

FIGURE 1.1 – Cycle en V



Appliqué cette méthode à l'entièreté de ce projet n'est pas envisageable à cause du coût temporel qui est très élévé. On va se concentrer sur la réalisation d'une preuve de concept et se concentrer sur la partie automatisation. La conception du produit sera minimale et le développement des couvertures sera soumis à un futur travail.

Identification des besoins et exigences

On a déjà conçus notre carnet d'exigence, en revanche on ne connais pas le comportement des outils que l'on souhaite employer. La première opération est de s'approprier le

fonctionnement des outils que l'on s'apprête à employer. 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 de l'analyse qui sera effectué :

```
$ 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 des propriétés 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'instruction 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 exempe d'usage de Binsec et permet de réaliser une attaque sur un binaire issu d'une plateforme d'apprentissage à la sécurité logiciel[3]. L'exerice consistant 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 été conçu avec pour objectif de vérifier les résultats apportés par [Sch+24] concernant une fuite présente sur la fonction «*FStar_UInt64_eq_mask*» et d'étendre l'analyse vers d'autres architectures. Dans une première démarche d'automatisation, ce code à été généré automatiquement par un script shell. On voit ici que l'analyse ne parcourt pas l'entièreté du binaire, seulement 8 sections sont chargés (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 addresse 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 1.2 – 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	✓	✓	✓	✓	✓

✓ : *binaire secure*

On comprend, à la lecture du tableau 1.2, que la politique temps constant est considérée respectée par Binsec sur les versions testé ainsi que pour les différentes options de compilation. Ce résultat est encourageant pour la suite du projet.

FIGURE 1.3 – 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	✓	×	✓	×

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

Les résultats dans le tableau 1.3 est indéniable : la version 19.1.7 de clang rend le code source perméable à des attaques temporelles.

Identification de défaut

Pour construire le tableau 1.3, plusieurs alertes se sont levées et on 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 géré. 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 part l'envergure de ce projet, il est possible que d'autre erreurs dû à Binsec soient découverte. 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 construction de tests manuels.

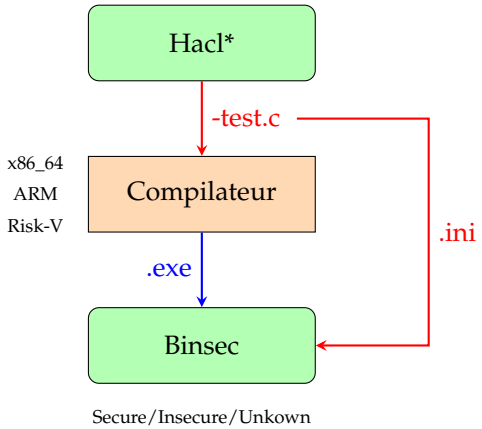
a. Ancronyme anglais pour Architecture de Jeu d'Instruction, désigne l'ensemble des instructions assembleur associées à une architecture.

En explorant plus en avant le code binaire, on découvre que ces erreurs sont dus à l'opérande `beqz`[4]. L'ISA de Risc-V n'a pas à sa dispostion 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*».

On a pus observer ici une manifestation indéniable des précédents résulats 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 donné dans le code source indique au compilateur de désactiver ses optmisation pour le code contenu entre les deux balise `off,on`. Cette solution entraîne des pertte de performance et des ralentissements quand au temps de compilation et à l'usage des ressources, il vaut mieux l'utiliser avec parcimonie que désactiver toutes les optimisations de compilations.

Nous avons notre besoin, nous avons nos exigences, nous avons fait des tests pour comprendre le processus que l'on va devoir automatser. Ces idées peuvent être illustrées par la figure 1.4 : depuis Hacl*, on extrait une fonction que l'on veut tester en un fichier de test en C, on identifie les paramètres secrets et on conçois le script pour Binsec, on compile notre fichier C à notre guise, on lance l'analyse Binsec.

FIGURE 1.4 – Flot de travail de l'outil d'analyse à concevoir



Transition

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

CHAPITRE 2

Érysichthon à jamais affamé

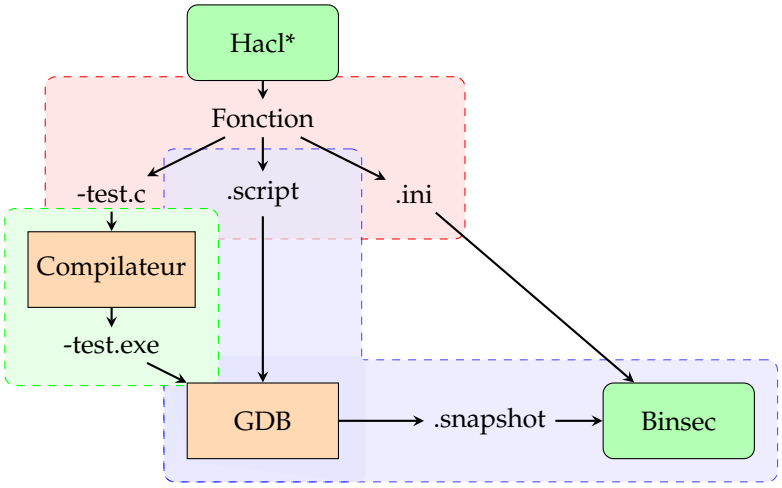
intro

2.1 Planification et préparations

Nous avons nos spécificités technique et nous savons qu'elle forme notre outil doit avoir. 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. On va se servir de cet avantage pour l'intégrer 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 2.1 – Protocole pour analyser des fichiers compiler en x86_64

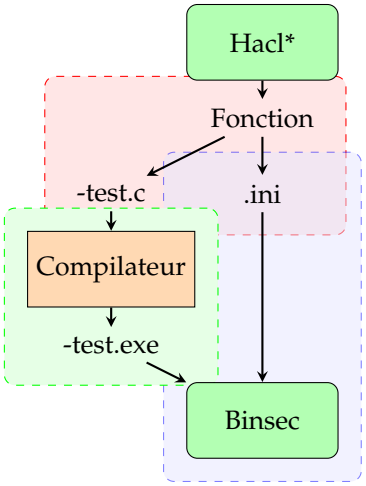


Ce graphe présente la chaîne d'étape à avoir pour obtenir une analyse Binsec depuis une fonction que l'on cible. Plusieurs zones sont distinguées. La zone verte correspond à l'étape de compilation, la zone bleue à l'étape de préparation de l'analyse et la zone rouge à la synthèse des fichiers de tests et d'instruction pour l'analyse. Ce choix de couleur est adapté à la difficulté attendu de chaque étape. L'opération de compilation consiste en une commande. L'opération de préparation d'analyse consiste aussi en deux commandes : un appel à GDB avec le binaire puis un appel à Binsec avec le cliché mémoire et les instruction 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 analyser, généralement par un débogueur.

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

FIGURE 2.2 – Protocole générique d'analyse



Dans ce contexte, une question se pose : est-ce que la conception des script 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 2.1 ?

Cette question est importante parce que lorsqu'on analyse un fichier ARM par exemple, si le fichier source contient des appels à des fonctions systèmes, les `IFUNC`, celles-ci sont exécutées en fonction de l'architecture qui exécute le programme. Or Binsec n'a pas ces informations, il faut les fournir à la main. Voici le scipt 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>

lr<64> := 0xdeadbeef as return_address

starting from <main>
with concrete stack pointer

halt at return_address
halt at <_IO_puts>
explore all
```

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

Les lignes 5 à 7 sont présente pour indiquer les branchement à effectuer par Binsec lorsque qu'il rencontre ces adresses. Cette opération automatiquement exécutée lors de l'initialisation de l'exécution doit ici être précisée avec les fonction présente 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 réaffecté et leur attribuer les fonctions les plus adaptées.

Nommer un outil

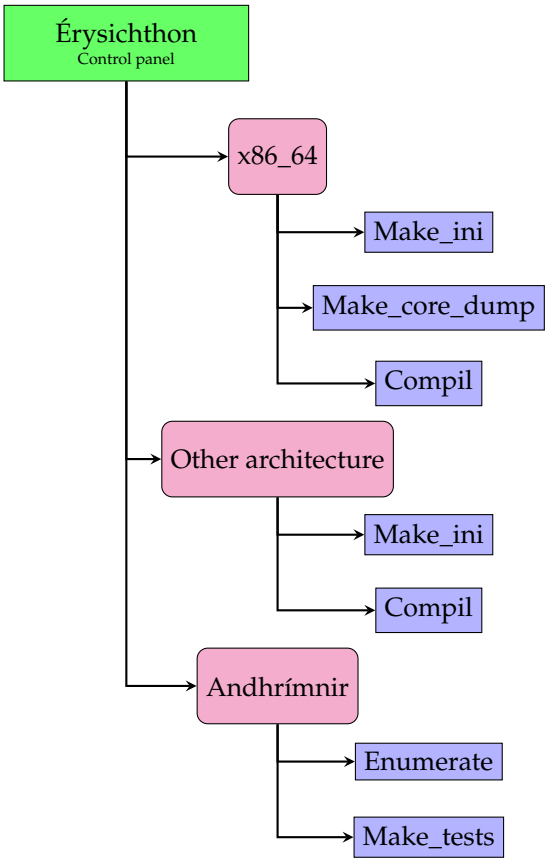
Rapidement il a fallu trouver un nom pour ce projet, l'appeler par "Notre outil… " devenait lourd et redondant entre les réunion hebdomadaire. En revanche trouver LE nom adéquat n'est pas une chose aisée, il peut être dû à une blague, une référence ou plus simplement liés au sens du projet. Dans notre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

Érysichthon est une 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

On a des protocoles. On a des expériences sur des fonctions de Hacl*. On a un nom. Il faut maintenant passer à l'échelle et concevoir un outil complet qui intéragit et exécute nos instructions. L'outil sera en somme une combinaison de script python, shell et de Makefile. Il faut donc les organiser et adapter leurs intéractions pour arriver au bout de nos peines.

FIGURE 2.3 – Structure des modules d'Érysichthon



On retrouve les différentes étapes de nos protocoles représentées par des modules au sein de leur noeuds respectifs : «`Make_ini`» pour les 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. Premièrement c'est le seul qui est aussi son nom. Ce module consiste a produire les fichiers qui seront compilés puis analysés par Binsec. Ce module est nommé d'après le cuisinier de la mythologie nordique, car comme lui, il prépare sempiternellement des mets pour son maître.

Ce module est nommé car il constitue un projet dans le projet, sa seule conception pris plus de la moitié du temps de développement total. C'est un outil qui, à partir d'un pro-

jet 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 porter 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), on a ici une couverture complète des fonctions portés par le projet C.

Ce module, comme son grand frère, fonctionne et est abouti, par contre il nécessite quelques opérations manuelles et peut être améliorer pour pouvoir supporter d'autre projet C. Actuellement il possède quelques optimisation associées à Hacl* pour accélérer la réalisation de la preuve de concept.

2.2 Conception et usages

On va commencer par le petit frère, Andhrímnir. Il fonctionne avec une phase d'initialisation «`Enumerate`» et une phase de production de tests «`Make_tests`», elle même découpé en plusieurs étapes. La génération de **548** fichiers de tests est réalisé moins de 2 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, on peut exploiter cette particularité pour lister rapidement nos fonctions. L'opération actuellement réalisé 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 l'on peut spécifier ou retirer des fichiers ou plus précisément des fonctions de la chaîne de production. Ce garde fou permet d'accélérer l'obtention des résultats finaux et d'aider grandement lorsque l'on souhaite débuguer.

Make_test

Le modèle de fichier de l'on veut construire est similaire aux tests minimaux que l'on a pus effectuer auparavant. Des paramètres, une déclaration de fonction et un appel à la fonction `exit`, c'est la recette pour une analyse simple. Binsec réalise une analyse symbolique, il ignore donc la valeur réelle des entrées. L'objectif est donc de concevoir un test qui compile pour avoir ensuite les bonnes instructions assembleurs à analyser. L'exemple 2.4 illustre comment sont générer nos fichiers de tests.

FIGURE 2.4 – 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

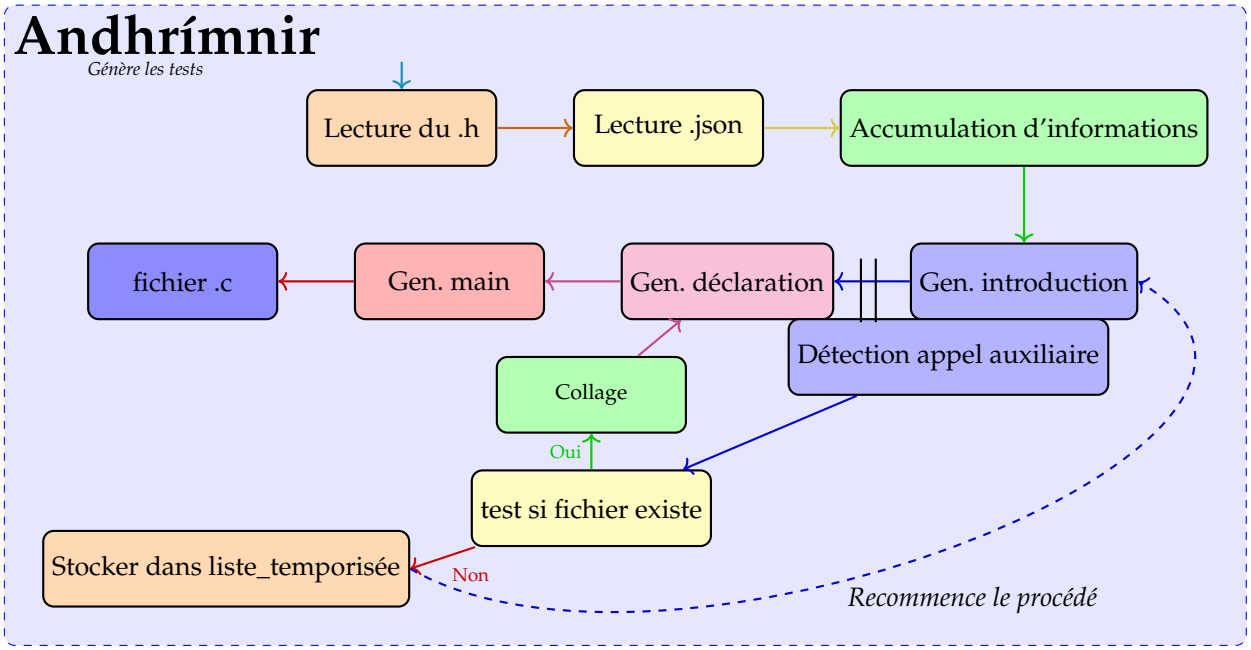
```
int main (int argc, char *argv[]){
Hacl_EC_K256_felem_sqr(a, out);
  exit(0);
}
```

Phase principale

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

La deuxième partie contient tous les éléments déclaratifs nécessaire à l'invocation de la fonction. Puis on 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 standardisé entre les fichiers et permet de mettre en places quelques optimisations.

FIGURE 2.5 – Schéma de conception d'Andhrímnir



Comme illsutré par la figure 2.5, 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" ensuite est réalisé pour avoir toutes les informations nécessaire à la constitution du fichier test. Une foix ces étapes réalisés, Andhrímnir commence sa préparation. or il se trouve que quelques foix, certaines fonctions de Hacl* font appel à des structures propres à la librairie qui ont une instanciation particulière. Le module «`Détection appel auxiliaire`» permet de vérifier ce cas de figure.

Dans le cas où aucun appel n'est détecté, Andhrímnir continue sa préparation avec les étapes successives illustrées par la figure 2.4 : génération des déclaration puis du main.

À l'inverse où un appel est détecté, il est possible la fonction soit déclaré 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 et de temporiser le problème. Andhrímnir prépare des tests pour toute les fonctions. Donc s'il a besoin d'une fonction qu'il a déjà préparé, on peut accéder aux informations contenues dans le fichier de test associé ; 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 foix qu'il a finis son premier passage sur toute les fonctions d'Hacl*. Ce procédé est récursif pour pallier le problème d'appels en cascade.

En réalité Andhrímnir ne va pas recharger les informations d'une fonction qu'il a besoin, il va faire 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 donées json existe pour accélerer le développement de Andhrímnir. Cela permet de venir ajouter manuellement des instructions haut niveau pour la conception des tests. Le code en annexe 6 illustre ce point. Une fonction a besoin de que les éléments qui lui sont données en paramètres respecte ses attentes. Cet exemple est accompgnée du fichier json associé et du fichier de tests final **??**.

Make_ini

Transition

CHAPITRE 3

Références

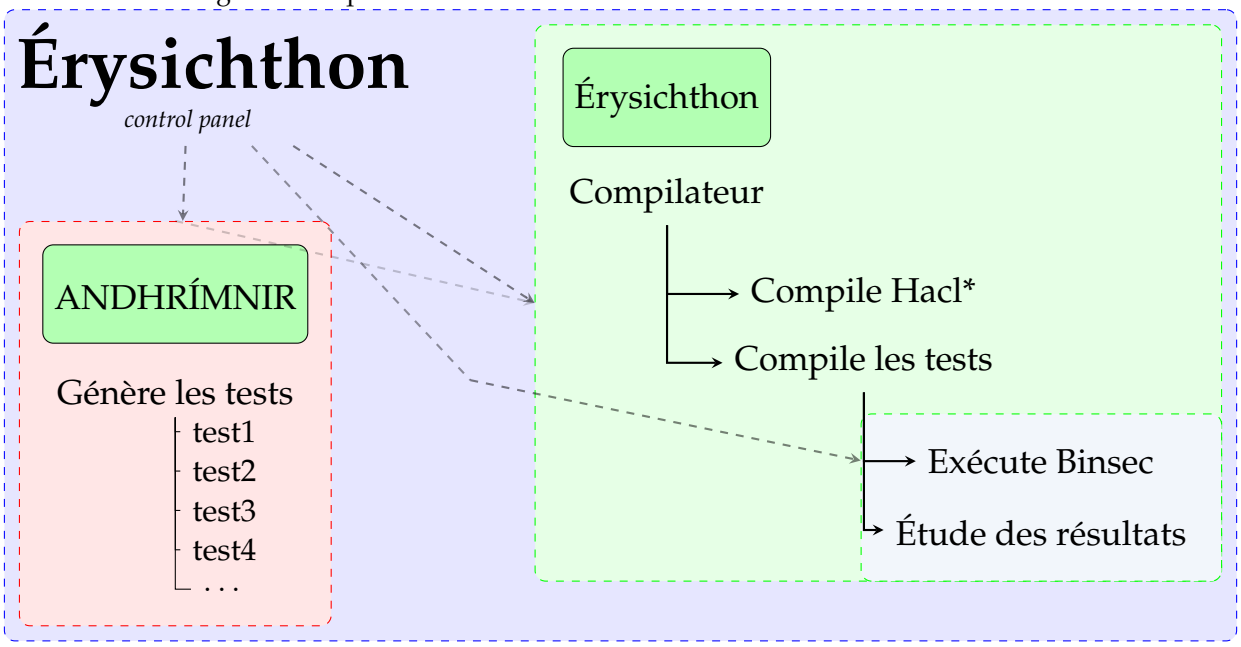
TABLE 3.1 – Liste des options de compilations et leurs effets (non exhaustive), `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

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

CHAPITRE 4

Conception de tests

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 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 du fichier test Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.c