CHAPITRE 1

Implémentations pour un usage industriel

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 **??**) 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 partagée 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.

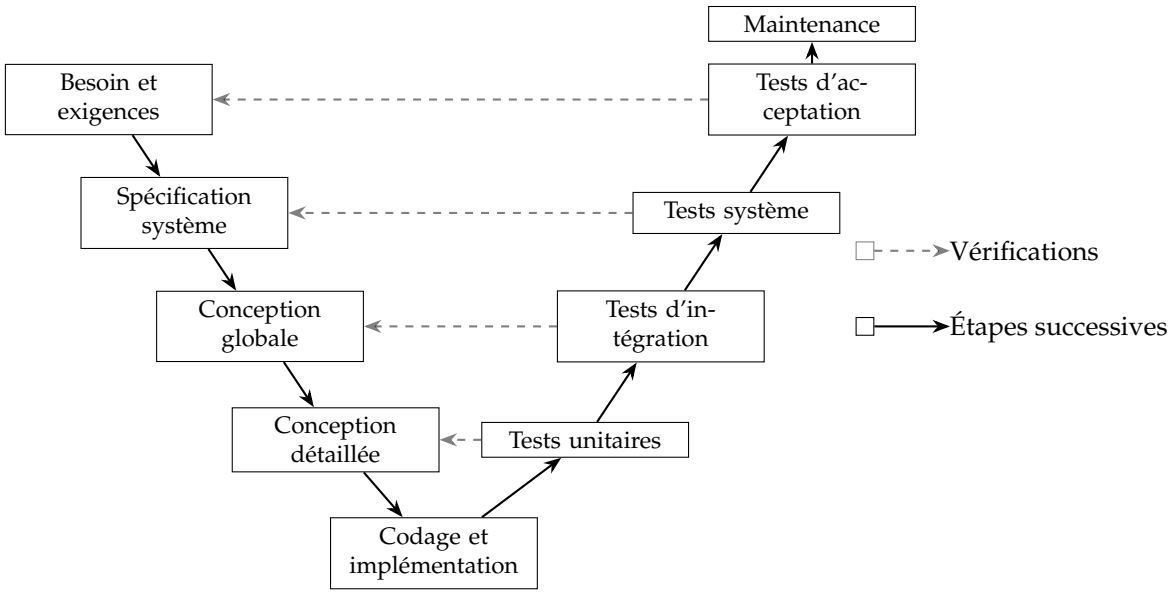
Couverture d'architectures. x86_64 et ARM sont les architectures matérielles les plus répandues dans le monde. Étendre l'analyse vers différentes plateformes et observer les différences qui émergent nous permettraient d'avancer dans la direction de la conception d'une librairie cryptographique universelle. 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`.

Cette phase permet de tester plusieurs fonctions de Hacl* et surtout 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 issu du tutoriel de Binsec et permet de réaliser une attaque sur un binaire issu d'une plateforme d'apprentissage à la sécurité informatique. 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.

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

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

secret global  r, cin, y, x

starting from <main>

with concrete stack pointer
halt at  0x0000000000000464
explore all

```

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 :

FIGURE 1.2 – Tableau de résultats d'analyse Binsec pour architecture ARMv7

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

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
-O2	✓	×	✓	×
-O3	✓	×	✓	×
-Os	✓	×	✓	×
-Oz	✓	×	✓	×

CHAPITRE 2

Érysichton à jamais affamé

- point histoire
- structure / schemas
- usage
- Andrihminir

usage de l'outil, comment ça rend