

Analyse automatisée d'une bibliothèque cryptographique



Détection de failles par canal auxiliaire par analyse statique et symbolique

Duzés Florian

Master Cryptologie et Sécurité Informatique

29 août 2025



Introduction - 1

Introduction - 1

HACL*

"**H**igh **A**ssurance **C**ryptography **L**ibrary"[Zin+17]^a est une bibliothèque cryptographique, écrite en F* ("F star"), implémentant tous les algorithmes de cryptographie modernes et est prouvée mathématiquement sûre.

HACL* est notamment utilisé dans plusieurs systèmes de production tels que Mozilla Firefox, le noyau Linux, le VPN WireGuard...

a. <https://hacl-star.github.io/>

Introduction - 2

1996 : Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*

Une mesure précise du temps requis par des opérations sur les clés secrètes permettrait à un attaquant de casser le cryptosystème.

Introduction - 2

1996 : Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*

Une mesure précise du temps requis par des opérations sur les clés secrètes permettrait à un attaquant de casser le cryptosystème.

2003 : BRUMLEY et BONEH *Remote Timing Attacks Are Practical*

Introduction - 2

1996 : Paul C. Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*

Une mesure précise du temps requis par des opérations sur les clés secrètes permettrait à un attaquant de casser le cryptosystème.

2003 : BRUMLEY et BONEH *Remote Timing Attacks Are Practical*

2011 : BRUMLEY et TUVERI *Remote Timing Attacks are Still Practical*



Introduction - 3

- QR1** Est-il possible de propager les garanties de sécurité pendant la compilation ?
- QR2** Est-il possible d'automatiser la détection de ces failles sur des fichiers compilés ?
- QR3** Est-il possible d'appliquer ces mécanismes pour assurer la vérification d'une bibliothèque cryptographique ?

Sommaire

1. Méthodes de protection et limitations
2. Outils de vérifications
3. Automatismes
4. Érysichthon
 1. Conception générale
 2. Andhrímnir
5. Résultats
6. Annexes

02

Méthodes de protection et limitations



État des lieux

Usage sécurisé

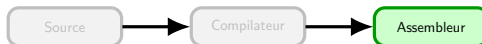


Analyse en remontée - 1

Écrire en assembleur

- + Efficace
- + Contrôle total

- Restreint l'architecture et les usages
- Beaucoup de connaissance spécifique au processeur ciblé



Analyse en remontée - 2

Utilisation des compilateurs

- Constantine - 2021
- Jasmin - 2017
- Raccoon - 2015
- CompCert - 2008 (2019)



Analyse en remontée - 2

Utilisation des compilateurs

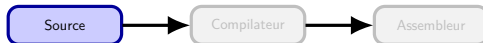
- Constantine - 2021
 - Jasmin - 2017
 - Raccoon - 2015
 - CompCert - 2008 (2019)
- Couverture des architectures supportée
 - Informations à transmettre
 - Spécifications ne sont plus respectées



Analyse en remontée - 3

Programmation en temps constant

- + Position haut niveau
- + Couverture d'architectures importantes
- Rigueur et conception particulière des actions
- Identification des points de fuites



Opérations dangereuses

Opérations influantes :

- Accès mémoire
- Décalage/rotation de valeurs
- Saut conditionnel
- Division/multiplication

Opérations dangereuses

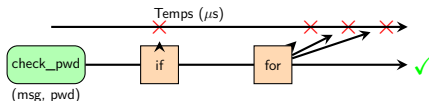
Opérations influantes :

- Accès mémoire
- Décalage/rotation de valeurs (caché)
- Saut conditionnel
- Division/multiplication

```

1      bool check_pwd(msg, pwd){
2          if (msg.length != pwd.length){
3              return False
4          }
5          for(int i = 0; i < msg.length; i++){
6              if(msg[i] != pwd[i]){
7                  return False
8              }
9          }
10         return True
11     }

```





Plus de problème ?



Plus de problème ?

Mauvaises nouvelles ?

2019 : DANIEL, BARDIN et REZK, *Binsec/Rel : Efficient Relational Symbolic Execution for Constant-Time at Binary-Level*



Plus de problème ?


Mauvaises nouvelles !

2019 : DANIEL, BARDIN et REZK, *Binsec/Rel : Efficient Relational Symbolic Execution for Constant-Time at Binary-Level*

2024 : SCHNEIDER et al., *Breaking Bad : How Compilers Break Constant-Time Implementations*

03

Outils de vérifications



Spécialisations

Outil	Cible	Techn.	Garanties
ctgrind [Lan10]	Binaire	Dynamique	▲
ABPV13 [Alm+13]	C	Formel	●
VirtualCert [Bar+14]	x86	Formel	●
ct-verif [Bar+16]	LLVM	Formel	●
FlowTracker [RPA16]	LLVM	Formel	●
Blazer [Ant+17]	Java	Formel	●
BPT17 [BPT17]	C	Symbolique	▲
MemSan [Tea17]	LLVM	Dynamique	▲
Themis [CFD17]	Java	Formel	●
COCO-CHANNEL [Bre+18]	Java	Symbolique	●
DATA [Wei+18] ; [Wei+20]	Binaire	Dynamique	▲
MicroWalk [Wic+18]	Binaire	Dynamique	▲
timecop [Nei18]	Binaire	Dynamique	▲
SC-Eliminator [Wu+18]	LLVM	Formel	●
Binsec/Rel [DBR19]	Binaire	Symbolique	▲
CT-WASM [Wat+19]	WASM	Formel	●
FaCT [Cau+19]	DSL	Formel	●
haybale-pitchfork [Dis20]	LLVM	Symbolique	▲

Liste d'outils de vérification

Source : [Jan+21]

Cible

[C, Java] Code source

Binaire Binaire

DSL Surcouche de langage

Trace Trace d'exécution

WASM Assembleur web

Techn.

Formel Programmation formelle

[*] type d'analyse

Garanties (attaques temporelles)

● = Analyse correcte, ▲ = Limitée

L'outil idéal

Binsec

Binary Security^a est une plateforme open source développée pour évaluer la sécurité des logiciels au niveau binaire.

Il est notamment utilisé pour la recherche de vulnérabilités, la désobfuscation de logiciels malveillants et la vérification formelle de code binaire. Grâce à l'exécution symbolique, Binsec peut explorer et modéliser le comportement d'un programme pour détecter des erreurs ; cette détection est réalisée en association avec des outils de fuzzing et/ou des solveurs SMT.

a. <https://binsec.github.io/>

04

Automatismes



Premiers scripts

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

```

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,      all_printables<1> := true
6  @[arg1, 128] := 0
7  for i<64> in 0 to size - 1 do
8    @[arg1 + i] := nondet as password
9    all_printables := all_printables && " " <= password <= "~"
10  end
11  assume all_printables
12  end
13
14  replace <puts>, <printf> by return end
15
16  reach <puts> such that @[rdi, 14] = "Good password!"
17  then print ascii stream password
18
19  cut at <puts> if @[rdi, 17] = "Invalid password!"

```


Simplification

Codes : Instructions permettant d'analyser le code précédent compilé vers Risc-V 32bits

```
1  #include <stdlib.h>
2  #include <stdint.h>
3  #include "Hacl_P256.h"
4
5  #define SIZE 4
6  uint64_t cin;
7  uint64_t x[SIZE]; uint64_t y[SIZE]; uint64_t r[SIZE];
8
9  int main(){
10     bn_cmovznz4(r, cin, x, y);
11 }
```

Adaptation

Codes : Instructions permettant d'analyser le code précédent compilé vers Risc-V 32bits

```
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 0x00000000000000464
9 explore all
10
```

Premiers pas vers l'automatisation

Table : 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* ; ~ : *binary unknown* ; × : *binary insecure*

Recherche de failles

Table : 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*



Cahier des charges

Objectifs du futur outil

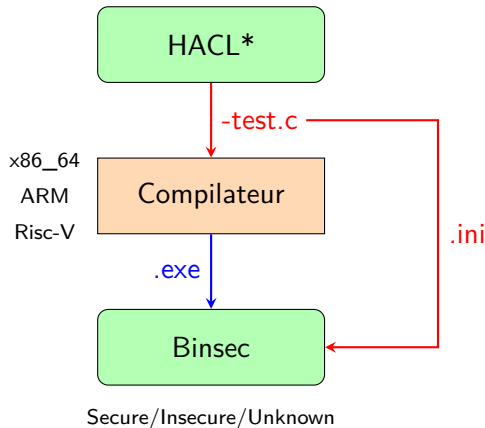
- Petits fichiers
binaire
- Analyse complète
de la bibliothèque
- Analyse correcte
- Automatique

Cahier des charges

Objectifs du futur outil

- Petits fichiers binaire
- Analyse complète de la bibliothèque
- Analyse correcte
- Automatique
- Couverture [architectures, compilateurs]

Figure : Flot de travail de l'outil

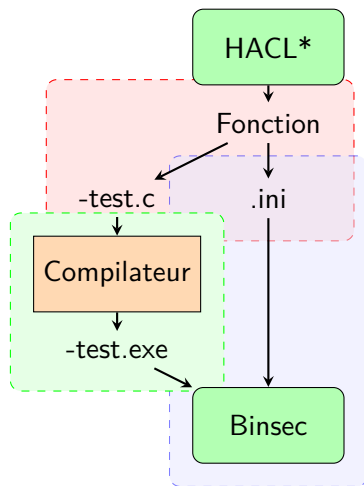


05

Érysichthon



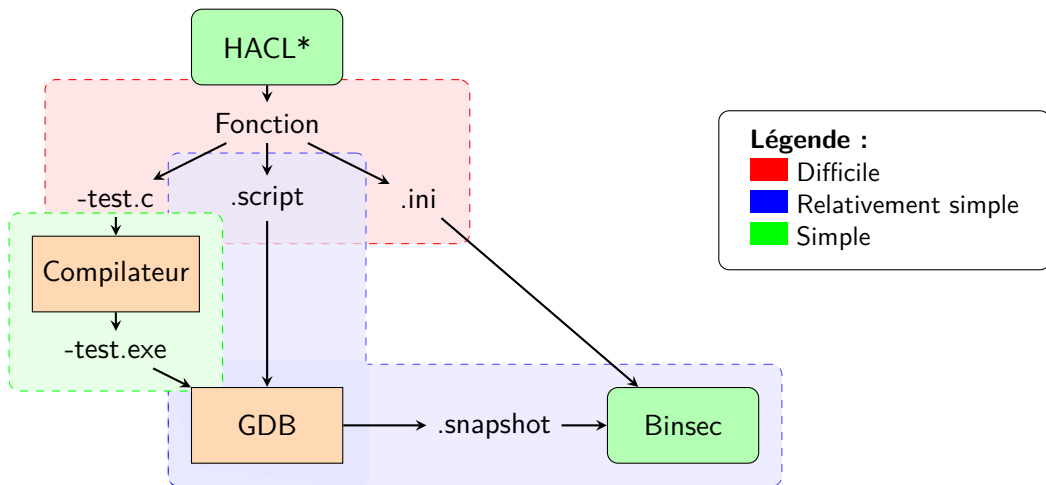
Conception générale



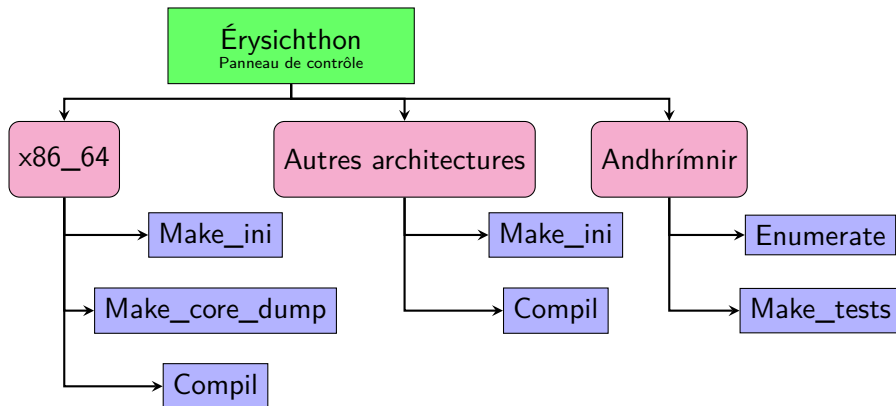
Légende :

- Difficile
- Relativement simple
- Simple

Adaptation architecturale



Construction en modules



Andhrímnir - 1

Module indépendant

- Réalise des tests complet d'un projet C
- Réalise des tests automatiquement



Andhrímnir - 1

Module indépendant

- Réalise des tests complet d'un projet C
- Réalise des tests automatiquement

Module adapté

- Optimisation pour HACL*
- Communications avec Érysichthon
- Adaptées pour de l'analyse symbolique



Exemple de test

Code : 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"

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

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

Exemple de test

Code : Test de la fonction `Hacl_EC_K256_felem_sqr`

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

Phase introductive : 8 lignes

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

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

Phase déclarative

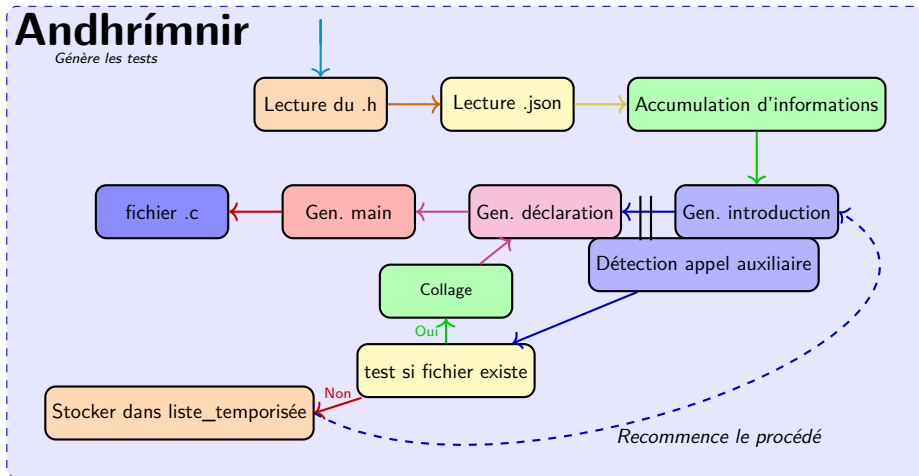
```
int main (int argc, char *argv[]){  
    Hacl_EC_K256_felem_sqr(a, out);  
    exit(0);  
}
```

Phase principale




Andhrímnir - 2

Andhrímnir - 2

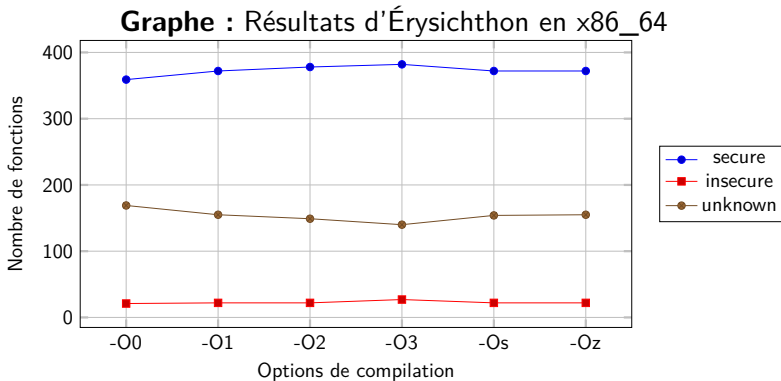


06

Résultats

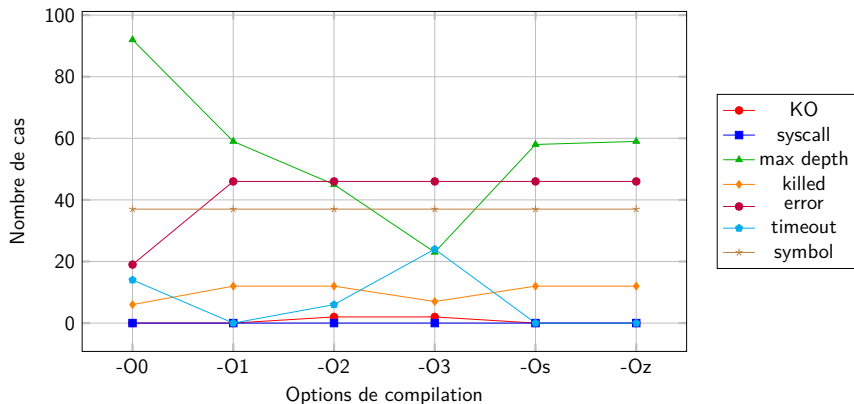


Premières passes



Analyses

Graphes : Détail des erreurs interrompant l'analyse Binsec



Conclusion





Références

- [BT11] Billy Bob BRUMLEY et Nicola TUVERI. *Remote Timing Attacks are Still Practical*. Cryptology ePrint Archive, Paper 2011/232. 2011. URL : <https://eprint.iacr.org/2011/232>.
- [BB03] David BRUMLEY et Dan BONEH. *Remote Timing Attacks Are Practical*. Washington, D.C., août 2003. URL : <https://www.usenix.org/conference/12th-usenix-security-symposium/remote-timing-attacks-are-practical>.
- [DBR19] Lesly-Ann DANIEL, Sébastien BARDIN et Tamara REZK. *Binsec/Rel : Efficient Relational Symbolic Execution for Constant-Time at Binary-Level*. 2019. arXiv : 1912.08788. URL : <http://arxiv.org/abs/1912.08788>.
- [Sch+24] Moritz SCHNEIDER et al. *Breaking Bad : How Compilers Break Constant-Time Implementations*. 2024. arXiv : 2410.13489 [cs.CR]. URL : <https://arxiv.org/abs/2410.13489>.
- [Zin+17] Jean-Karim ZINZINDOHOUE et al. *HACL* : A verified modern cryptographic library*. 2017. URL : <https://hacl-star.github.io/>.

09

Annexes



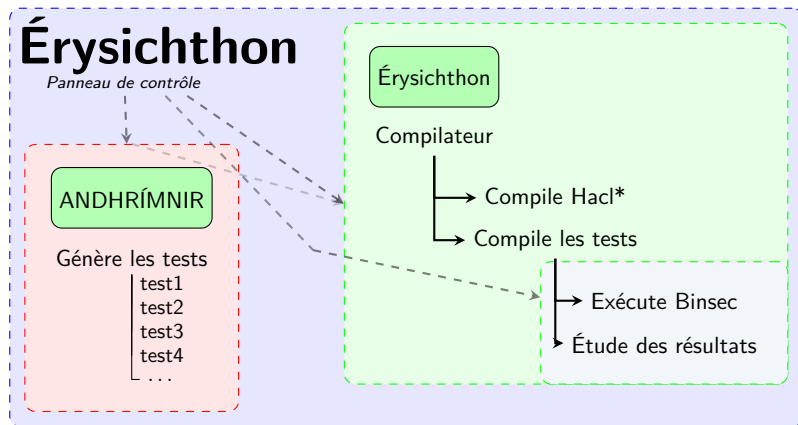
Options de compilations

Tableau : Liste des options de compilations et leurs effets (non exhaustive) ¹

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

1. <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

Construction en vue depuis l'utilisateur



Fin de stage

Liste des freins au temps constant

- Mécanisme de Pipeline
- Micro instructions
- Renommage des registres
- Prédiction de branches
- Exécution désordonnée
- In silicium JIT

Constant-Time Code: The Pessimist Case

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

6 March, 2025

Abstract. This note discusses the problem of writing cryptographic implementations in software, free of timing-based side-channels, and many ways in which that endeavour can fail in practice. It is a pessimist view: it highlights why such failures are expected to become more common, and how constant-time coding is, or will soon become, infeasible in all generality.