

Analyse automatisée d'une bibliothèque cryptographique



Détection de failles par canal auxiliaire par analyse statique et symbolique

Duzés Florian

Master Cryptologie et Sécurité Informatique

4 septembre 2025

01

Introduction





Introduction

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.

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*

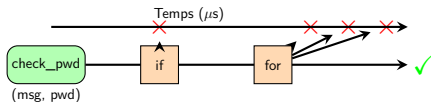
Une mesure précise du temps requis par des opérations sur les clés secrètes permettrait à un attaquant de casser le cryptosystème.

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

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

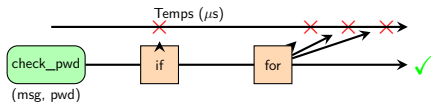
Petit exemple

```
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 }
```



Petit exemple

```
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 }
```

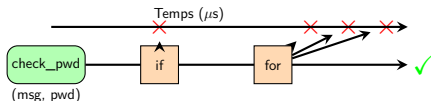


Petit exemple

```
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 }
```

Opérations influentes :

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

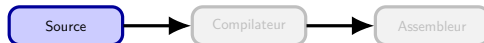




Axes de défenses contre les attaques par chronométrage



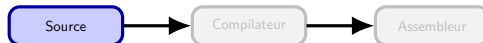
Travail à la source



Programmation en temps constant

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

Travail à la source



Programmation en temps constant

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

2024 - SCHNEIDER et al.

Breaking Bad : How Compilers Break Constant-Time Implementations

Travail avec le compilateur



Utilisation des compilateurs

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

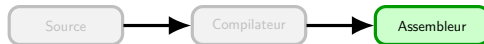
Travail avec le compilateur



Utilisation des compilateurs

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

Travail en assembleur



Écrire en assembleur

- + Efficace
- + Contrôle total

- Spécifique à chaque processeur qui exécute
- Beaucoup de connaissance spécifique au processeur ciblé
- Long à mettre en place
- Portabilité faible



Réalisation

Réalisation



Vérification de binaire

- Approche systématisée et automatique
- Analyse correcte et complète
- Processus d'intégration continue

Réalisation



Vérification de binaire

- Approche systématisée et automatique
- Analyse correcte et complète
- Processus d'intégration continue

Érysichthon

Premier outil de vérification complète de bibliothèque cryptographique face aux attaques temporelles.

Analyse HACL* :

	Prouvé	Attendu
(%) Fontions sécurisées	67,96%	62,39%
(%) Fontions attaquables	3,08%	27,61%
(%) Analyse interrompue	28,96%	0%




Sommaire

- 1. Introduction**
- 2. Outils de vérification**
- 3. Automatisme**
 1. Étude sur cas simple
 2. Contraintes et identification des limitations
- 4. Érysichthon**
 1. Conception générale
 2. Andhrímnir
- 5. Résultats**
- 6. Conclusion**
- 7. Annexes**

02

Outils de vérification





Choix de l'outil

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



Choix de l'outil

Outil	Cible	Techn.	Garanties
ctgrind [Lan10]	Binaire	Dynamique	▲
DATA [Wei+18] ; [Wei+20]	Binaire	Dynamique	▲
MicroWalk [Wic+18]	Binaire	Dynamique	▲
timecop [Nei18]	Binaire	Dynamique	▲
Binsec/Rel [DBR19]	Binaire	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



Choix de l'outil

Outil	Cible	Techn.	Garanties
Binsec/Rel [DBR19]	Binaire	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 [DBR19]

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

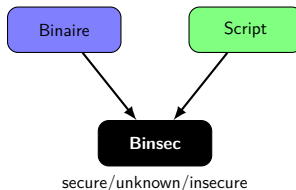
Il permet la recherche de vulnérabilités, la désobfuscation de logiciels malveillants et la vérification formelle de code binaire. Grâce à l'exécution symbolique, Binsec peut explorer et modéliser le comportement d'un programme pour détecter des erreurs ; cette détection est réalisée en association avec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

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

Fonctionnement

Détail :

- Analyse au niveau binaire
- Supporte plusieurs architectures (x86_64, ARM, PowerPC, Risc-V)
- Configuration automatisable



```
$ binsec -sse -sse-script $(SCRIPT) -checkct $(BIN)
```


Usage

Code : Instructions pour trouver le mot d'un passe d'un crackme (61c8deff33c5d413767ca0ea)

```
1  starting from core
2  return_address<64> := @[rsp, 8]
3  replace <puts>, <printf> by
4      return
5  end
6  replace <__isoc99_scanf> by
7      assert @[rdi, 3] = "%s"z
8      len<64> := nondet           ,assume 0 < len < 50
9      all_printables<1> := true
10     for i<64> in 0 to len - 1 do
11         @[rsi + i] := nondet as key
12         all_printables := all_printables && " " <= key <= "~"
13     end
14     @[rsi + i] := "\x00"         ,assume all_printables
15     return
16 end
17 reach <printf> such that @[rdi, 16] = "[+] Correct key!"    then print ascii stream key
18 cut at <printf> if @[rdi, 18] = "[-] Incorrect key!"
19 halt at <__stack_chk_fail>, return_address
```

Usage

Code : Instructions pour trouver le mot d'un passe d'un crackme (61c8deff33c5d413767ca0ea)

Chargement des données

```

1  starting from core
2  return_address<64> := @[rsp, 8]
3  replace <puts>, <printf> by
4    return
5  end
6  replace <__isoc99_scanf> by
7    assert @[rdi, 3] = "%s"z
8    len<64> := nondet           ,assume 0 < len < 50
9    all_printables<1> := true
10   for i<64> in 0 to len - 1 do
11     @[rsi + i] := nondet as key
12     all_printables := all_printables && " " <= key <= "~"
13   end
14   @[rsi + i] := "\x00"         ,assume all_printables
15   return
16 end
17 reach <printf> such that @[rdi, 16] = "[+] Correct key!"   then print ascii stream key
18 cut at <printf> if @[rdi, 18] = "[-] Incorrect key!"
19 halt at <__stack_chk_fail>, return_address

```

Usage

Code : Instructions pour trouver le mot d'un passe d'un crackme (61c8deff33c5d413767ca0ea)

	Chargement des données
1 starting from core	
2 return_address<64> := @[rsp, 8]	
3 replace <puts>, <printf> by	Stubs
4 return	
5 end	
6 replace <__isoc99_scanf> by	
7 assert @[rdi, 3] = "%s"z	
8 len<64> := nondet ,assume 0 < len < 50	
9 all_printables<1> := true	Stubs
10 for i<64> in 0 to len - 1 do	
11 @[rsi + i] := nondet as key	
12 all_printables := all_printables && " " <= key <= "~"	
13 end	
14 @[rsi + i] := "\x00" ,assume all_printables	
15 return	
16 end	
17 reach <printf> such that @[rdi, 16] = "[+] Correct key!" then print ascii stream key	
18 cut at <printf> if @[rdi, 18] = "[-] Incorrect key!"	
19 halt at <__stack_chk_fail>, return_address	

Usage

Code : Instructions pour trouver le mot d'un passe d'un crackme (61c8deff33c5d413767ca0ea)

Chargement des données	
1	starting from core
2	return_address<64> := @[rsp, 8]
3	replace <puts>, <printf> by
4	return
5	end
6	replace <__isoc99_scanf> by
7	assert @[rdi, 3] = "%s"z
8	len<64> := nondet ,assume 0 < len < 50
9	all_printables<1> := true
10	for i<64> in 0 to len - 1 do
11	@[rsi + i] := nondet as key
12	all_printables := all_printables && " " <= key <= "~"
13	end
14	@[rsi + i] := "\x00" ,assume all_printables
15	return
16	end
17	reach <printf> such that @[rdi, 16] = "[+] Correct key!" then print ascii stream key
18	cut at <printf> if @[rdi, 18] = "[-] Incorrect key!"
19	halt at <__stack_chk_fail>, return_address



Retour de Binsec

```

1  $ binsec -sse -sse-script crackme.ini -sse-depth 10000 core.snapshot
2  [sse:result] Path 22 reached address 0x7ffff7e115b0 (<printf>) (0 to go)
3  [sse:result] Ascii stream key : "34407373373234353336"
4  [sse:info] SMT queries
5
6      Preprocessing simplifications      Satisfiability queries
7      total          20894                total          80
8      true           5243                  sat            41
9      false          10923                 unsat           39
10     constant enum  4728                  unknown         0
11                                     time            0.39
12                                     average          0.00
13
14     Exploration
15     total paths          22
16     completed/cut paths  19
17     pending paths        3
18     stale paths          0
19     failed assertions    0
20     branching points     20931
21     max path depth       7212
22     visited instructions (unrolled) 143268
23     visited instructions (static)   3720

```

Retour de Binsec

Commande

```

1 $ binsec -sse -sse-script crackme.ini -sse-depth 10000 core.snapshot
2 [sse:result] Path 22 reached address 0x7ffff7e115b0 (<printf>) (0 to go)
3 [sse:result] Ascii stream key : "34407373373234353336"
4 [sse:info] SMT queries
5     Preprocessing simplifications      Satisfiability queries
6         total          20894          total          80
7         true           5243           sat            41
8         false          10923          unsat          39
9         constant enum  4728           unknown        0
10                                time            0.39
11                                average          0.00
12 Exploration
13     total paths          22
14     completed/cut paths  19
15     pending paths        3
16     stale paths          0
17     failed assertions    0
18     branching points     20931
19     max path depth       7212
20     visited instructions (unrolled) 143268
21     visited instructions (static)   3720

```

Résultats

Retour de Binsec

Commande

```

1 $ binsec -sse -sse-script crackme.ini -sse-depth 10000 core.snapshot
2 [sse:result] Path 22 reached address 0x7ffff7e115b0 (<printf>) (0 to go)
3 [sse:result] Ascii stream key : "34407373373234353336" Clé
4 [sse:info] SMT queries
5     Preprocessing simplifications      Satisfiability queries
6         total          20894          total          80
7         true           5243           sat            41
8         false          10923          unsat          39
9         constant enum  4728           unknown         0
10                                time            0.39
11                                average          0.00
12 Exploration
13     total paths          22
14     completed/cut paths  19
15     pending paths        3
16     stale paths          0
17     failed assertions    0
18     branching points     20931
19     max path depth       7212
20     visited instructions (unrolled) 143268
21     visited instructions (static)   3720

```

Résultats

03

Automatisme





Étude sur cas simple

Codes : Test de la fonction *bn_cmovznz4*

```
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 }
```

Codes : Script Binsec associé

```
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
```



Analyse de compilateur

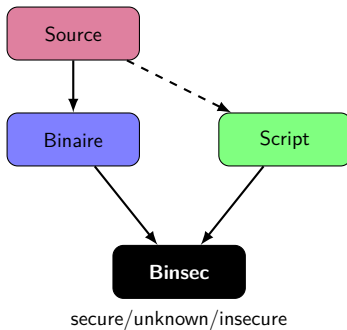
Table : Résultats d'analyse Binsec pour plusieurs compilateurs vers ARM

opt\fonction - architecture	cmovznz4 - 64b							cmovznz4 - 32b						
	gcc 15.1.0	clang 14.0.6	15.0.6	16.0.4	17.0.6	18.1.8	19.1.7	gcc 15.1.0	clang 14.0.6	15.0.6	16.0.4	17.0.6	18.1.8	19.1.7
-O0	~	✓	✓	✓	✓	✓	×	~	✓	✓	✓	✓	✓	×
-O1	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	×
-O2	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	×
-O3	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	×
-Os	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	×
-Oz	✓	✓	✓	✓	✓	✓	×	✓	✓	✓	✓	✓	✓	×

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

Points d'attention

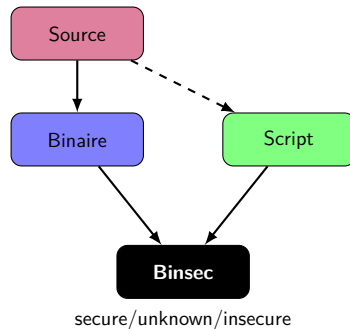
Dépendances des informations



Points d'attention

Dépendances des informations

- Variables secrètes
- Test correct





Cahier des charges

Objectifs du futur outil

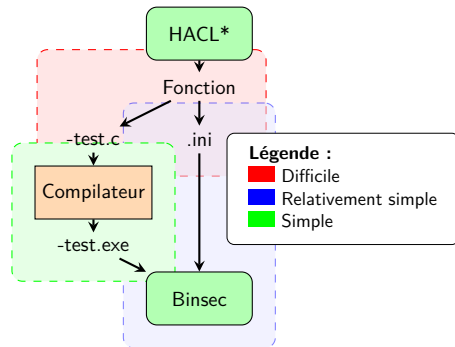
- Petits fichiers binaire
- Analyse correcte
- Analyse complète de la bibliothèque
- Automatique
- Couverture [*architectures*,
compilateurs]

Cahier des charges

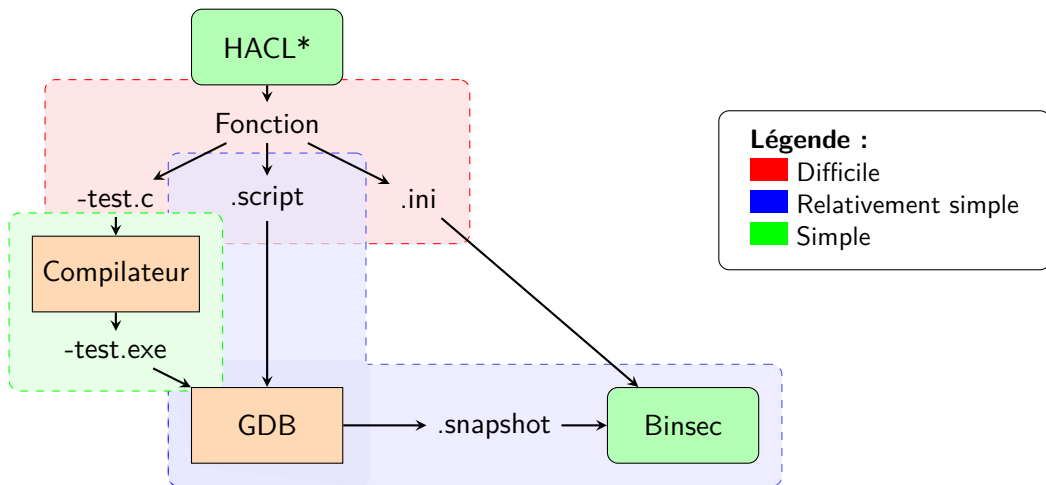
Objectifs du futur outil

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

Figure : Flot de travail de l'outil




Adaptation architecturale

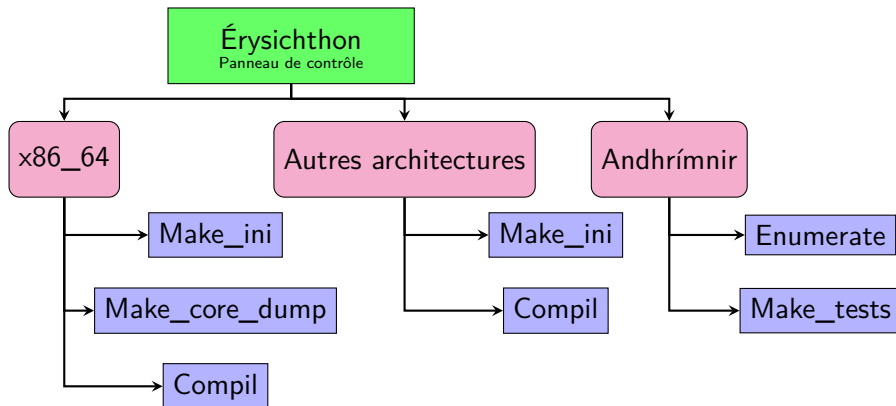


04

Érysichthon



Construction en modules





Andhrímnir - 1

Module indépendant

- Réalise des tests automatiquement
- Réalise des tests corrects



Andhrímnir - 1

Module indépendant

- Réalise des tests automatiquement
- Réalise des tests corrects

Module adapté

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

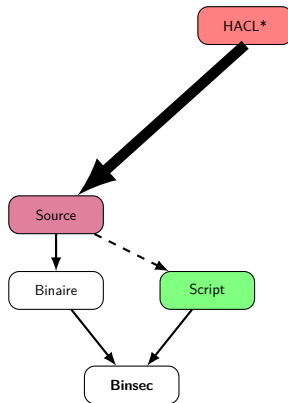
Andhrímnir - 1

Module indépendant

- Réalise des tests automatiquement
- Réalise des tests corrects

Module adapté

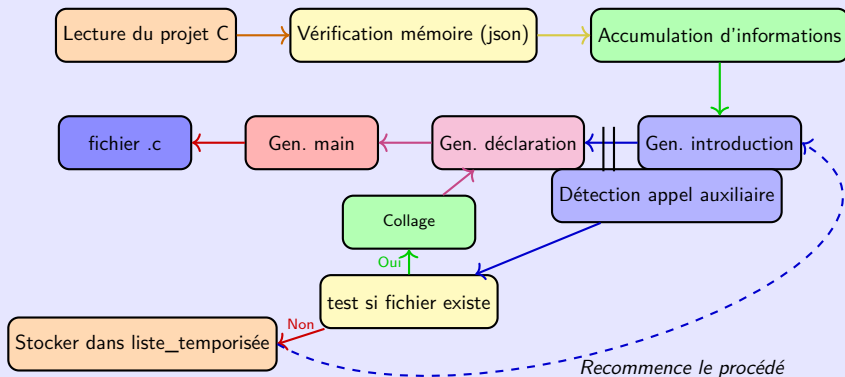
- Optimisation pour HACL*
- Communications avec Érysichthon
- Adaptées pour l'analyse symbolique avec Binsec



Andhrímnir - 2

Andhrímnir

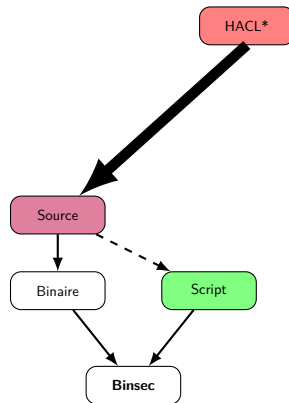
Génère les tests





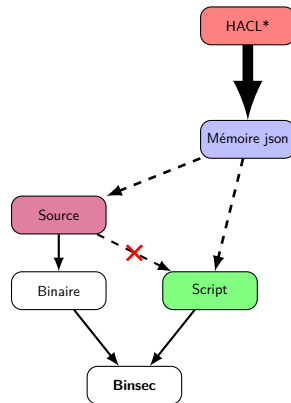
Pourquoi utiliser une mémoire

```
{
  "Meta_data":{
    "build" : "13-06-2025",
    "version" : "0.2.0"
  },
  "Hacl_AEAD_Chacha20Poly1305_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
  },
  .:.
}
```



Pourquoi utiliser une mémoire

```
{  
  "Meta_data": {  
    "build" : "13-06-2025",  
    "version" : "0.2.0"  
  },  
  
  "Hacl_AEAD_Chacha20Poly1305_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  
  },  
  ...  
}
```





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

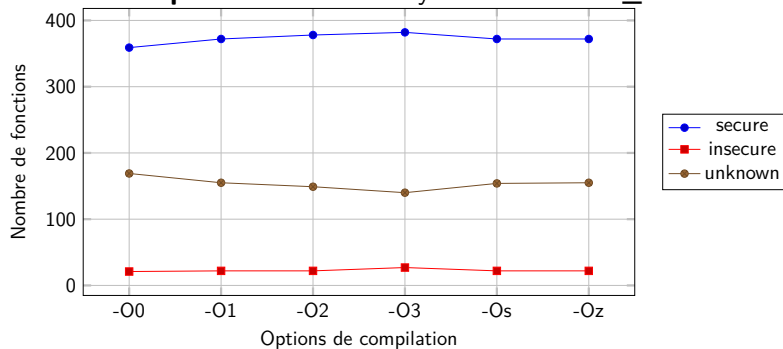
05

Résultats



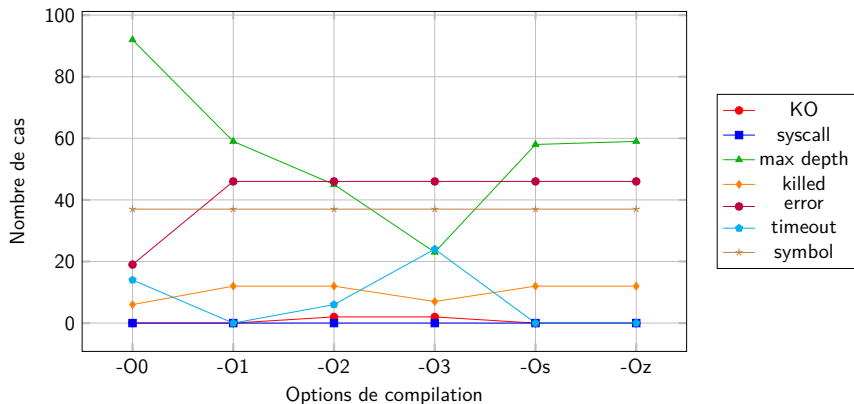
Premières passes

Graph : Résultats d'Érysichthon en x86_64



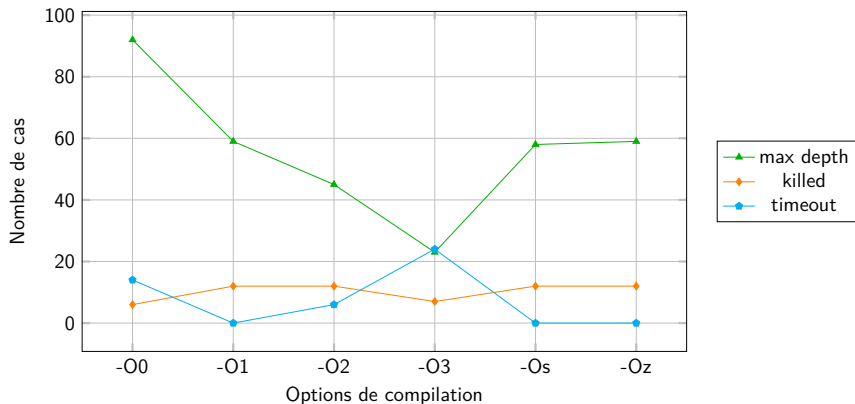
Analyses

Graphes : Détail des erreurs interrompant l'analyse Binsec



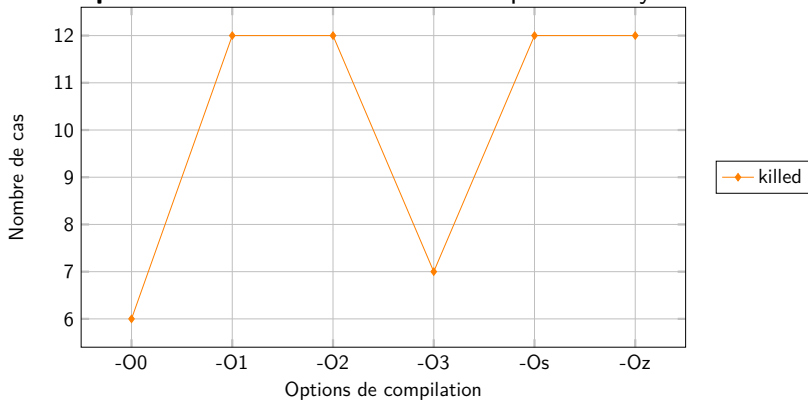
Analyses

Graphes : Détail des erreurs interrompant l'analyse Binsec



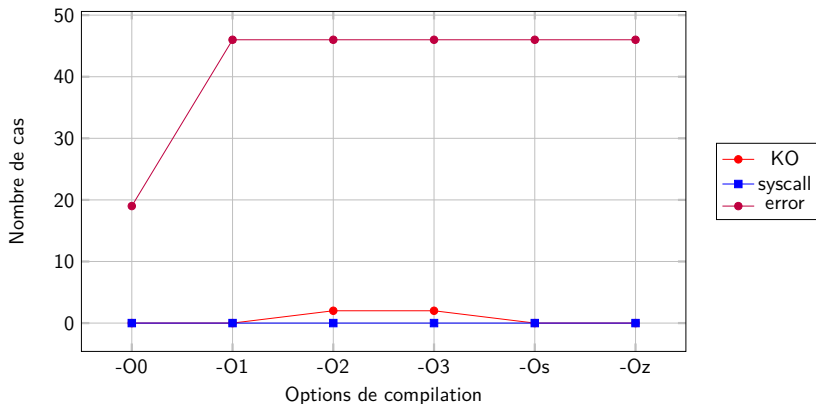
Analyses

Graphes : Détail des erreurs interrompant l'analyse Binsec



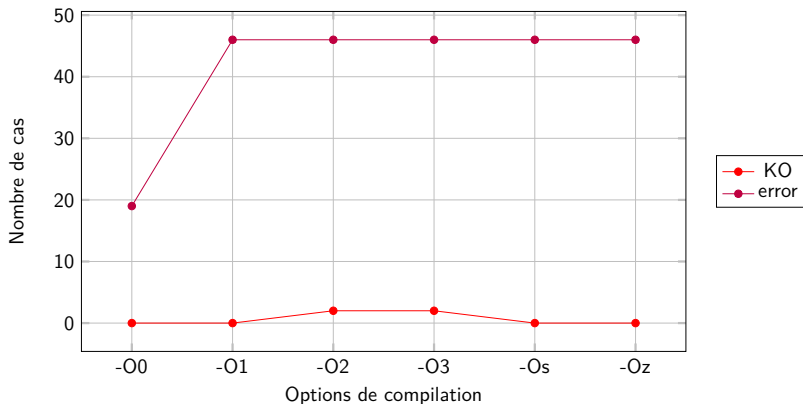
Analyses

Graphes : Détail des erreurs interrompant l'analyse Binsec




Analyses

Graphes : Détail des erreurs interrompant l'analyse Binsec



06

Conclusion



Conclusion

Point d'amélioration

- Réduire le nombre d'interruption
- Étendre l'analyse à d'autres compilateur
- Établir dans un processus d'intégration continue

Analyse Hacl*

	Prouvé	Attendu
(%) Fontions sécurisées	67,96%	62,39%
(%) Fontions attaquables	3,08%	27,61%
(%) Analyse interrompue	28,96%	0%

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/>.

08

Annexes



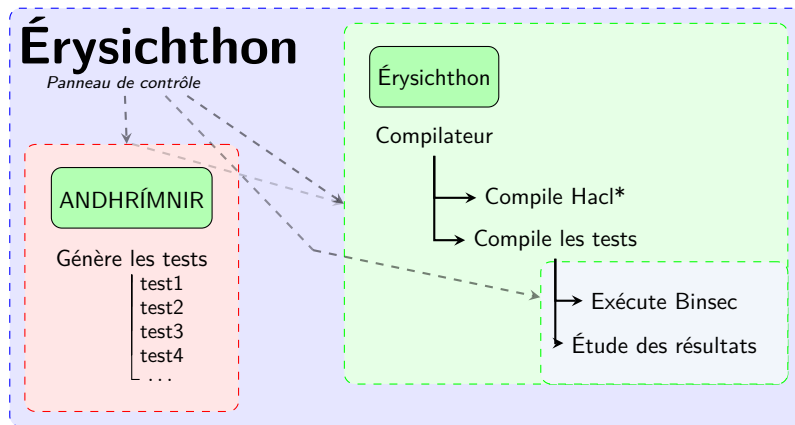
Options de compilations

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.