CHAPITRE 1

Outils et méthodes

chapitre sur les outils + moyens pour détecter

intro

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 que l'on souhaite cibler. On parle parfois de *modèle de fuite*. Cette étape de synthèse et d'abstraction est importante pour identifer les risques encourus par le futur système, souvent en identifiant les point 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 trace	Suivi des appels de fonctions, des accès réussis ou manqués à la mémoire.

Ces trois modèles sont notre source de méfiance et si on peut argumenter quand à l'inclusion de notre dernier modèle ; des travaux comme [Gau+23] portent directement sur des améliorations matériel pour contrecarrer ce type d'attaque. Considérer un attaquant plus puissant, avec des accès à des ressources supplémentaires, potentiellement hyptotétique, permet de concevoir un système plus sûr. Certains outils comme [HEC20 ; Wei+18] ou cette étude [Jan+21] exploitent cette mécanique pour attester de la sécurité d'un programme.

Puis, avec ces modèles et les contre-mesures connus, on peut constituer un ensemble de règles qui valident 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 que l'on souhaite 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 correct de programmes complexes (*i.e.* primitives cryptographiques asymétriques).

Formalisation de modèle

Si on regarde plus en détails les travaux nécessaires à la conception d'un tel modèle, on peut s'appuyer sur l'article "Secure Compilation of Side-Channel Countermeasures : The Case of Cryptographic "Constant-Time"" [BGL18].

On commence par définir un programme. Il s'agit d'une suite d'instruction binaire. Et une instruction est une action sur la mémoire. Cela nous permet de définir notre programme comme une suite de configuration (l, r, m) ; l la ligne d'instruction, r le dictionnaire de registre et m la mémoire. La configuration initiale est défini 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 registre 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 $c_0 \underset{f}{\rightarrow} c_1$, c_0 et c_1 deuc configurations successives, $\rightarrow$ la transition entre les deux et f une fuite émise par cette transition. Notons que certaines instructions ne produisent pas de fuites.

Une fois ce préambule installé on peut alors définir formellement le comportement d'une instruction. Si on regarde par exemple

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

$$\text{LOAD}\quad \frac{(l, r, m)\ e \vdash_t \mathrm{bv}}{(l, r, m)\ @\ e \vdash_{t\cdot[\mathrm{bv}]} m\ \mathrm{bv}}$$

- formalisation des regles de securite

The behavior of the program is modeled with an instrumented operational semantics taken from [69] in which each transition is labeled with an explicit notion of leakage. A transition from a configuration c to a configuration c0 produces a leakage t, denoted c → c0 . Analogously, the t evaluation of an expression e in a configuration (l, r, m), denoted (l, r, m) e 't bv, produces a leakage t. The leakage of a multistep execution is the concatenation of leakages produced by individual steps. We use → k with k a natural t number to denote k steps in the concrete semantics. An excerpt of the concrete semantics is given in Fig. 3 where leakage by memory accesses occur during execution of load and store instructions and control flow leakages during execution of dynamic jumps and conditionals. The full set of rules is given in Appendix A1.

[DBR19]

1.2 Analyse d'un programme

4.1 Static analysis Static analysis approaches attempt to derive security properties from the program without actually executing it, extracting formally defined guarantees on all possible executions through binary or source code analysis. As a formal exploration of every reachable state is unfeasible, program behavior is often approximated, making them prone to false positives. Static approaches were the first to be considered, as side-channel security is closely related to information flow policies [53]. 4.1.1 Logical reduction Non-interference is a 2-safety property stat- ing that two executions with equivalent public inputs and poten- tially different secret inputs must result in equivalent public outputs. This definition covers side channels by considering resource usage (e.g., address trace) as a public output. Approaches based on logical reduction to 1-safety transform the program so that verifying its side-channel security amounts to proving the safety of the trans- formed program. Self-composition [23] interleaves two executions of a program P with different sets of secret variables in a single self-composed pro- gram P ;P' . Solvers can then be used to verify the non-interference property. This approach was used by Bacelar Almeida et al. [15] to manually verify limited examples, relying on a large amount of code annotations. ct-verif instead runs the two copies in lockstep, while checking their assertion-safety [13]. It is able to verify LLVM programs, leveraging the boogie verifier. Sidetrail [19] reuses this to verify that secret dependent branches are balanced (assuming a fixed instruction cost and excluding memory access patterns), providing a counter-example when this verification fails. However, such approaches suffer from an explosion in the size of the program state space. Blazer [17] verifies timing-channel security on Java programs by instead decomposing the execution space into a partition on secret-independent branches. Proving 2-safety is thus reduced to verifying 1-safety on each trace in the partition improving scalability at the cost of precision. Themis [48] uses static taint analysis to automatically annotate secret-dependent Java code with Hoare logic formulas as pre- and post-conditions. An SMT solver then verifies that the post-condition implies execution time differences remain bounded by given constant. Both tools provide a witness triggering the vulnerability otherwise. 4.1.2 Type systems Approaches based on verifying type safety of a program differ from language-level countermeasures [12, 30], as CCS '23, November 26–30, 2023, Copenhagen, Denmark the developer only needs to type the secret values with annotations instead of rewriting the program. The type system then propagates this throughout the program, similarly to static taint analysis. Type systems were considered relatively early to verify non-interference properties [7] and offer good scalability but their imprecision makes them difficult to use in practice. VirtualCert [22] analyzes a modified CompCert IR where each instruction makes its successors explicit. The authors define seman- tics for that representation, building the type system on top of it. An alias analysis giving a sound over-approximated prediction of targeted memory address is needed to handle pointer arithmetic. While this approach is more suited to a strict verification task, it can also provide a leakage estimate. FlowTracker [107] introduces a novel algorithm to efficiently compute implicit information flows in a program, and uses it to apply a type system verifying constant-time. 4.1.3 Abstract interpretation As a program semantics is generally too complex to formally verify non-trivial properties, abstract in- terpretation [50] over-approximates its set of reachable states, so that if the approximation is safe, then the program is safe. CacheAudit [55] performs a binary-level analysis, quantifying the amount of leakage depending on the cache policy by finding the size of the range of a side-channel function. This side-channel function is computed through abstract interpretation, and the size of its range determined using counting techniques. It was later extended to support dynamic memory and threat models allowing byte-level observations [56] and more x86 instructions [91]. Blazy et al. [28] focus on the source code instead of the binary. Their tool is integrated into the formally-verified Verasco static an- alyzer, and uses the CompCert compiler. The analysis is structured around a tainting semantics that propagates secret information throughout the program. STAnalyzer [110] uses data-flow analysis to report secret-dependent branches and memory accesses. CacheS [126] uses an hybrid approach between abstract interpre- tation and symbolic execution. The abstract domain keeps track of program secrets—with a precise symbolic representation for values in order to confirm leakage—but keeps only a coarse-grain representation of non-secret values. To improve scalability, CacheS implements a lightweight but unsound memory model. 4.1.4 Symbolic execution Symbolic execution [81] (SE) denotes approaches that verify properties of a program by executing it with symbolic inputs instead of concrete ones. Explored execution paths are associated with a logical formula : the conjunction of condi- tionals leading to that path. A memory model maps encountered variables onto symbolic expressions derived from the symbolic in- puts and the concrete constants. A solver is then used to check whether a set of concrete values satisfies the generated formulas. Recent advances in SMT solvers have made symbolic execution a practical tool for program analysis [42]. CoCo-Channel [34] identifies secret-dependent conditions us- ing taint-analysis, constructs symbolic cost expressions for each path of the program uses SE and reports paths that exhibit secret- dependent timing behavior. Their cost model assigns a fixed cost per instruction, excluding secret-dependent memory accesses.CCS '23, November 26–30, 2023, Copenhagen, Denmark Several works use symbolic execution to derive a symbolic cache model and check that cache behavior does not depend on secrets. CANAL [117] models cache behaviors of programs directly in the LLVM intermediate representation by inserting auxiliary variable and instructions. It then uses KLEE [41] to analyze the program and check that the number of hits does not depend on secrets. Similarly, CacheFix [47] uses SE to derive a symbolic cache model supporting multiple cache policies. In case of a violation, CacheFix can synthesize a fix by injecting cache hits/misses in the program. CaSym [35] follows the same methodology and, to improve scala- bility, includes simplifications of the symbolic state and loop trans- formations, which are sound but might introduce false positives. SE suffers from scalability issues when applied to 2-safety prop- erties like constant-time verification. Daniel et al. [51] adapt its formalism to binary analysis, introducing optimizations to maxi- mize information shared between two executions following a same path. Their framework Binsec/Rel offers a binary-level CT analysis, performing a bounded exploration of reachable states and giving counterexamples for the identified vulnerabilities. Pitchfork [54] combines SE and dynamic taint tracking. It soundly propagates secret taints along all executions paths, reporting tainted branch conditions or memory addresses. Interestingly, Pitchfork can analyze protocol-level code by abstracting away primitives' im- plementations using function hooks, and analyzing them separately. ENCIDER [140] combines symbolic execution with taint analysis to reduce the number of solver calls. It also enables to specify information-flow function summaries to reduce path explosion. 4.2 Dynamic analysis Dynamic analysis groups approaches that derive security guar- antees from execution traces of a target program. Some form of dynamic binary instrumentation (DBI) is often used to execute the program and gather events of interest, such as memory accesses or jumps. Dynamic approaches differ in the events collected, and how traces are processed. They can be grouped depending on whether they reason on a single trace, or compare multiples traces together. 4.2.1 Trace comparison approaches Statistical tests. Statistical tests can be used to check if different secrets induce statistically significant differences in recorded traces. Cache Template [70] monitors cache activity to detect lines associ- ated with a target event, then finds lines correlated with the event using a similarity measure. A first pass using page-level observa- tions instead of lines can be used to improve scalability [112]. Shin et al. [114] use K-means clustering to produce two groups of traces for each line. The confidence in the partition indicates which line is likely to be secret-dependent. DATA [132] employs a Kuiper test then a Randomized Dependence Coefficient test to infer linear and non-linear relationships between traces and secrets. This was later extended to support cryptographic nonces as secrets [130]. Mutual information (MI) can be used to quantify the information shared between secret values and recorded traces, with a non-zero MI score giving a leakage estimation. MicroWalk [133] computes MI scores between input sets and hashed traces, with leakage lo- cation pinpointed using finer-grained instruction-level MI scores. MicroWalk-CI [134] optimizes this process by transforming the

Geimer et al. traces in call trees, and adds support for JavaScript and easy inte- gration in CI, following recommendations from [73]. CacheQL [141] reformulates MI into conditional probabilities, estimated with neu- ral networks. Leakage location is estimated by recasting the problem into a cooperative game solved using Shapley values. Contrary to other tools [20, 133], CacheQL does not assume uniform distribu- tion of the secret, nor deterministic executions traces. STACCO [136] targets control-flow vulnerabilities specifically in TLS libraries running on SGX, focusing on oracles attacks [11, 29]. Traces recorded under different TLS packets are represented as sequences of basic blocks and compared using a diff tool. Instead of recording traces, dudect [106] records overall clock cycles and compares their distribution with secret inputs divided in two classes (fix-vs-random). While this approach is simple and lightweight, it gives certainty that an implementation is secure up to a number of measurements. Contrary to other tools relying on an explicit leakage model, dudect directly monitors timings. Hence, vulnerabilities to other microarchitectural attacks like Hertzbleed might (in theory) be detected by dudect. Fuzzing. Fuzzing techniques can be used to find inputs maximiz- ing coverage and side-channel leakage. DifFuzz [95] combines fuzzing with self-composition to find side-channels based on in- struction count, memory usage and response size in Java programs. ct-fuzz [72] extends this method to binary executables and cache leakage. 4.2.2 Single trace Other approaches use only one trace to perform the analysis, sacrificing coverage for scalability. ctgrind [85] re- purposes the dynamic taint analysis of Valgrind to check CT by declaring secrets as undefined memory. This solution is easy to deploy and reuses familiar tools, but remains imprecise. ABSynthe [66] identifies secret-dependent branches using dy- namic taint analysis. It employs a genetic algorithm to build a sequence of instructions based on interference maps evaluating contention created by each x86 instructions. More precise approaches use SE to replay the trace with the secret as a symbolic value and check for CT violation. CacheD [127] applies this approach to memory accesses. Abacus [20] extends it to control-flow vulnerabilities, picking random values to check satisfiability instead of using a SMT solver. It also includes leakage estimation through Monte Carlo simulation. Finally, CaType [74] uses refinement types (i.e., types carrying a predicate restricting their possible values) on a trace to track constant bit values and improve precision. CaType also supports implementations that use blinding.

Transition

CHAPITRE 2

Automatisme et couverture

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

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

$\boxed{\textbf{Expr}}$

$$\text{CST}\ \frac{}{(l,r,m)\ \texttt{bv} \vdash \texttt{bv}}$$

$$\text{VAR}\ \frac{}{(l,r,m)\ \texttt{v} \vdash r\ \texttt{v}} \qquad \text{UNOP}\ \frac{(l,r,m)\ e \vdash \texttt{bv}}{(l,r,m)\ \blacklozenge_u e \vdash \blacklozenge_u \texttt{bv}}$$

$$\text{BINOP}\ \frac{(l,r,m)\ e_1 \vdash \texttt{bv}_1 \qquad (l,r,m)\ e_2 \vdash \texttt{bv}_2}{(l,r,m)\ e_1 \blacklozenge_b e_2 \vdash \texttt{bv}_1 \lozenge_b \texttt{bv}_2}$$

$$\text{LOAD}\ \frac{(l,r,m)\ e \vdash_t \texttt{bv}}{(l,r,m)\ \texttt{@}e \vdash_{t\cdot[\texttt{bv}]} m\ \texttt{bv}}$$

$\boxed{\textbf{Instr}}$

$$\text{S\_JUMP}\ \frac{P.l = \texttt{goto}\ l'}{(l,r,m) \xrightarrow[{[l]}]{} (l',r,m)}$$

D_JUMP
$$\frac{P.l = \texttt{goto}\ e \qquad (l,r,m)\ e \vdash_t \texttt{bv} \qquad l' \triangleq \mathit{to\_loc}(\texttt{bv})}{(l,r,m) \xrightarrow[t\cdot[l']]{} (l',r,m)}$$

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

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

$$\text{ASSIGN}\ \frac{P.l = \texttt{v}\ \ := \ e \qquad (l,r,m) e \vdash_t \texttt{bv}}{(l,r,m) \xrightarrow[t]{} (l+1, r[\texttt{v} \mapsto \texttt{bv}], m)}$$

STORE
$$\frac{P.l = \texttt{@}e := e' \qquad (l,r,m)\ e \vdash_t \texttt{bv} \qquad (l,r,m)\ e' \vdash_{t'} \texttt{bv}'}{(l,r,m) \xrightarrow[t'\cdot t\cdot[\texttt{bv}]]{} (l+1, r, m[\texttt{bv} \mapsto \texttt{bv}'])}$$

FIGURE 3.1 – Ensemble d'instructions définis formellement par [DBR19]

CHAPITRE 4

Érysichthon, structure et exemples

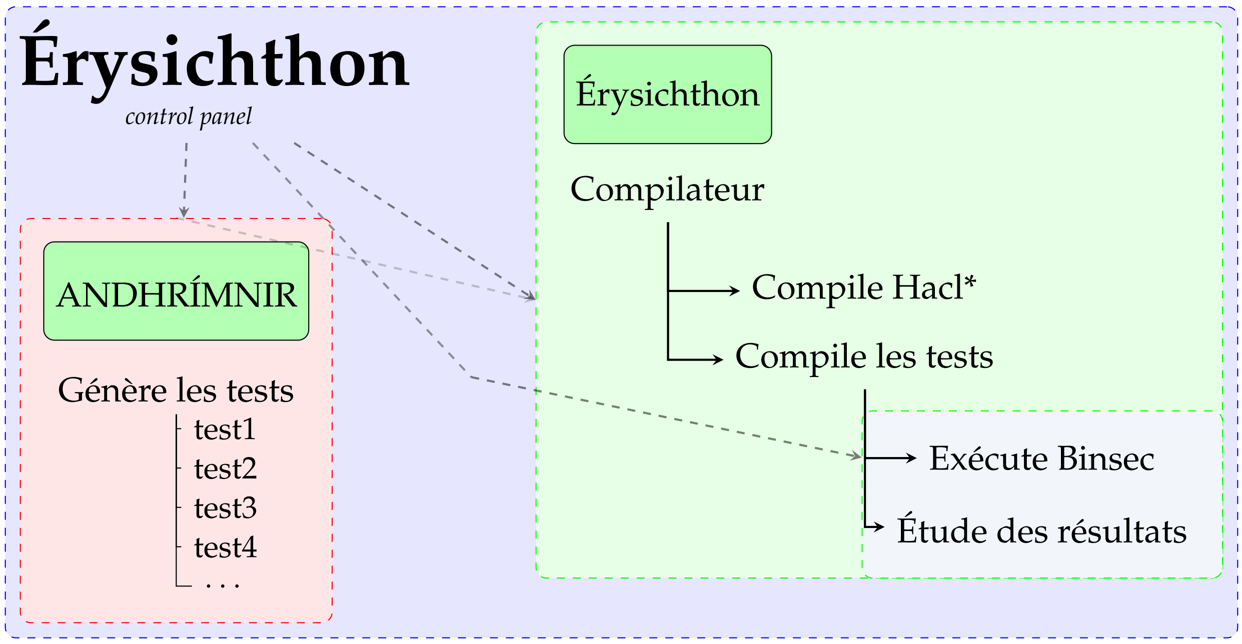

FIGURE 4.1 – Structure d'Érysichthon, schéma du point de vue de l'usager
Les flèches grises indiquent tous les éléments actionnables individuellement.

```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 1 – 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 2 – 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 3 – 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 4 – Instruction Binsec générée automatiquement,
Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.ini