université de BORDEAUX

Analyse automatisée d'une bibliothèque cryptographique

Détection de failles par canal auxiliaire par analyse statique et symbolique

Mémoire de fin d'étude

Master *Sciences et Technologies*,
Mention *Informatique*,
Parcours *Cryptologie et Sécurité Informatique*.

Auteur
Florian Duzes <florian.duzes@u-bordeaux.fr>

Superviseur
Emmanuel Fleury <emmanuel.fleury@u-bordeaux.fr>

Tuteurs
Aymeric Fromherz<aymeric.fromherz@inria.fr>
Yanis Sellami<yanis.sellami@cea.fr>
Sebastien Bardin<sebastien.bardin@cea.fr>

version 1.5 – 22 août 2025

Introduction

Le développement sécurisé est une tâche ardue. Si nous portons notre regard vers le langage de programmation C, un guide [Can14] d'apprentissage du langage est complet en 133 pages, tandis qu'un guide pour le développement sécurisé[ANS20] produit par l'Anssi comprend 182 pages. Cette comparaison met en évidence la discipline requise par le développeur pour faire de la programmation sécurisée ; en plus des connaissances en cryptologie, en architecture matérielle et en programmation bas niveau nécessaires pour améliorer l'efficacité de son programme.

Malheureusement, malgré ces compétences, des erreurs peuvent être commises puis exploitées pour réaliser des attaques sur ces systèmes sécurisés. Il existe de nombreuses classes d'attaques, certaines exploitant les défauts de conception (type A) tandis que d'autres utilisent les caractéristiques matérielles (type B). Pour limiter ces effets ces risques, la pratique de la programmation formelle permet de contraindre le développeur et d'empêcher l'apparition de ces erreurs. La production de preuves formelles du code à l'issue de cet exercice permet d'avoir des garanties contre les attaques de type A.

En revanche, pour se défendre contre des attaques de type B, dépendantes du matériel supportant le programme, il est plus difficile d'avoir une méthode générique. Notamment à cause de toutes les variations existantes (*e.g.* : architectures variés, différentes versions de supports physiques, *etc*). Actuellement, la solution la plus courante consiste à identifier les attaques existantes afin d'ajouter les contre-mesures adéquates permettant d'obtenir un système sécurisé. Une sous-classe d'attaques continue malgré tout de résister à cette méthode : les attaques temporelles.

Découvertes par Paul Kocher en 1996 [Koc96], il les décrit comme «*une mesure précise du temps requis par des opérations sur les clés secrètes, permettrait à un attaquant de casser le cryptosystème*». Face à cette menace, l'enjeu d'avoir un code *achrognostique*[1] vient s'ajouter aux pratiques de programmation sécurisée. Et pourtant, si contre les attaques de type A nous arrivons à concevoir des preuves mathématiques de sécurité associées à nos systèmes sécurisés, les garanties contre les attaques de type B sont plus faibles ou inexistentes.

En 2024, les travaux de SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN [Sch+24] prouvent qu'un usage inadapté de compilateur sur un système sécurisé introduit des failles exploitables. Ces résultats, observables partiellement avec des travaux antérieurs (par exemple [DBR19]), montrent qu'un usage inadéquat d'options fournies au compilateur optimise un code prouvé sécurisé et retire les protections indiquées par le développeur. Cela nous amène à plusieurs questions de recherche (QR) auxquelles nous tenterons de répondre dans ce document.

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 ?

Les réponses à ces questions permettraient de développer des systèmes sécurisés, communs entre différents supports et architectures, ainsi que d'avoir de fortes garanties de sécurité. Pour la suite du document nous emploierons le terme de *binaire* ou *fichier binaire* pour nous référer aux fichiers compilés.

Ce mémoire se compose en deux parties. La première partie présente les attaques temporelles, leur impact, comment s'en protéger et les outils à disposition pour s'en prémunir.

1. Néologisme de Thomas Pornin dans son article *Constant-Time Code : The Pessimist Case* [Por25] pour désigner un code sans connaissance de temps

Dans la deuxième partie nous verrons les travaux nécessaires à la conception et à l'implémentation d'un outil de détection automatique de ces attaques temporelles, ainsi que les premiers résultats apportés par cette démarche novatrice.

Ce mémoire a été réalisé au sein du centre Inria de Paris en collaboration avec le centre CEA de Paris-Saclay.

Projets supports

Nous présentons succinctement les deux projets servant de fondations à la réalisation des travaux présentés dans ce mémoire.

HACL*

Acronyme pour "High assurance cryptography library"[2], lire *"HACL star"*. Il s'agit d'une bibliothèque cryptographique développée au sein du **Projet Everest**[3]. Initié en 2016, ce projet porté par des chercheurs de l'Inria (équipe PROSECCO[4]), du Centre de Recherche Microsoft et de l'Université Carnegie Mellon a pour but de concevoir des systèmes informatiques formellement sécurisés, appliqués à l'environnement HTTPS. Cette bibliothèque, écrite en F* ("F star"), implémente tous les algorithmes de cryptographie modernes et est prouvée mathématiquement sûre. Elle est ensuite transcrite en C pour être directement employée dans n'importe quel projet. HACL* est notamment utilisé dans plusieurs systèmes de production, notamment Mozilla Firefox, le noyau Linux, le VPN WireGuard,*etc*.

Binsec

Binsec (contraction de *Binary Security*)[5] est une plateforme open source développé pour évaluer la sécurité des logiciels au niveau binaire. Ce logiciel est développé et maintenu par une équipe du CEA List de l'Université Paris-Saclay, et accompagné de chercheurs de Verimag[6] et du LORIA[7]. Il notamment est 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 ; détection réalisée en association avec des outils de fuzzing et/ou des solveurs SMT.

2. `https://hacl-star.github.io/`
3. `https://project-everest.github.io/`
4. Équipe de recherche rattachée au centre Inria de Paris, focalisée sur les méthodes formelles et la recherche en protocoles cryptologiques. Pour atteindre ces objectifs, l'équipe développe des langages de programmation, des outils de vérification…
5. `https://binsec.github.io/`
6. Verimag est un laboratoire spécialisé dans les méthodes formelles pour une informatique sûre, avec des applications aux systèmes cyber-physiques. Fondé en 1993 au sein de l'Université Grenoble Alpes, puis rejoint par le CNRS, il a pour objectif la sécurité informatique dans les domaines des transports et de la santé.
7. Laboratoire lorrain de recherche en informatique et ses applications ; créé en 1997, c'est un centre de recherche commun au CNRS, à l'Université de Lorraine, à CentraleSupélec et à l'Inria.

CHAPITRE 1

Présentation, enjeux et attaques

Ce premier chapitre a pour but de présenter les enjeux de la sécurité informatique face aux attaques par canal auxiliaire et d'introduire les attaques temporelles. Nous distinguerons ces attaques en deux catégories, montrant ainsi la diversité et les potentiels dangers pour un système sécurisé ignorant de cette menace.

1.1 L'exécution du code est observable...

L'Informatique repose sur deux fondations que nous tendons à distinguer dans l'enseignement : le matériel et le logiciel. Pourtant, si nous gardions séparé ces deux domaines, nous aurions des tas de piles de métal et de plastiques d'un côté et des bibliothèques pleines d'idées intéressantes de l'autre. Au contraire, combiner les deux parties permet de réaliser des prouesses technologiques et scientifiques. Ainsi, lorsque nous concevons un système sécurisé, il nous faut prendre en compte ces deux composantes. Pour implémenter un système sécurisé, il ne faut pas seulement avoir un logiciel sécurisé, il est tout aussi important d'avoir un support physique sécurisé. Oublier ce détail, c'est oublier que programmer peut se résumer à manipuler de l'électricité.

Les attaques par canal auxiliaires consistent à exploiter les caractéristiques matériels du support pour gagner en connaissances sur un programme ciblé. Puis exploiter ces connaissances pour acquérir d'avantage d'informations privées : identifiants, clés secrètes, messages personnels. Nous leurs attribuons le terme "canal auxiliaire" car il ne s'agit pas de trouver une faille perdue dans les limites d'un logiciel ou d'exploiter une mécanique du logiciel pour sortir de l'espace prévu par le concepteur. Il s'agit de se positionner hors du cadre de développement. Voici quelques travaux présentant une attaque par canal auxiliaire et le canal exploité :

[KJJ99] Consommation d'énergie
[AKS06] Prédiction de branchement
[Mas+15] Variation de température
[Pes+16] Accès à la mémoire DRAM

Le point commun entre ces attaques est la nécessité d'avoir un point de contact avec la cible. Il faut que l'attaquant puisse récupérer le matériel informatique ou le programme qu'il souhaite attaquer pour ensuite poser des sondes/capteurs afin d'accumuler de la connaissance et monter son exploitation.

Une autre technique d'attaque consiste à venir introduire une erreur dans le déroulement normal d'un programme. Il s'agit d'une attaque par injection de faute. Originellement [Avi71] les fautes étaient "naturelles" : un défaut dans le code, un problème avec la transcription vers du code machine, un défaut d'un composant dans le système ou une interférence. Ces interférences sont causées par une irrégularité de l'alimentation électrique, des radiations électromagnétiques, une perturbation environnementale *etc* ... En 2004, BAR-EL, CHOUKRI, NACCACHE, TUNSTALL et WHELAN dans leur article *The Sorcerer's Apprentice*

Guide to Fault Attacks [Bar+04] effectuent un tour d'horizon des techniques, montrant l'efficacité de cette méthode sur RSA , NVM[1] , DES, EEPROM[2], JVM[3]. Nous retrouvons enfin une liste de contre-mesures et de méthodes de protection contre ces attaques.

Ainsi, donner un accès physique à un inconnu est une porte d'entrée pour un attaquant. Pourtant, penser que l'accès physique au support est une condition nécessaire et suffisante pour réaliser une attaque par canal auxiliaire est une erreur.

1.2 ...à distance

En effet, il est possible de réaliser des attaques à distance en exploitant d'autres failles de sécurité d'un programme ou d'autres caractéristiques matériels. L'attaque présentée par LIU, YAROM, GE, HEISER et LEE dans *Last-Level Cache Side-Channel Attacks are Practical* [Liu+15] repose sur la conception des services clouds où les machines virtuelles accèdent au même matériel. Tandis que la virtualisation crée l'illusion de compartimentation entre les sessions, en réalité, les adresses mémoire pointent vers une ressource physique partagée. Ainsi, l'exploitation du cache du dernier niveau (LLC) permet à un co-hôte de récupérer les clés secrètes d'un autre utilisateur. L'attaquant remplit le cache, puis mesure les temps d'accès vers ces registres, si des modifications apparaissent dans ces temps, cela signifie que la victime a accédé à ces registres. En répétant cette opération, l'attaquant peut reconstruire les clés secrètes de la victime.

D'autres attaques distantes comme celle de LIU, YAROM, GE, HEISER et LEE existent [YGH16 ; Mog+17 ; VPS18], mais nous observons rapidement que ces techniques emploient aussi la méthode de chronométrage. En effet, si nous ciblons un algorithme et que nous mesurons son temps d'exécution, si en fournissant différentes entrées (considérées secrètes) des variations sont observées entre les mesures, alors cela signifie que l'algorithme présente une dépendance à ces entrées. Une sous-fonction de cet algorithme est généralement responsable de ces variations. Cette classe d'attaque est appelée «*attaque temporelle*»[4].

Le lien entre temps et exécution de code est connu depuis le début de l'informatique. Le temps est le marqueur de performance, d'efficacité d'un programme. En revanche, l'idée d'exploiter cet indice pour réaliser une attaque est arrivée plus tardivement. KOCHER nous présente, le premier en 1996, comment monter une attaque en utilisant ce canal.

Ce lien entre le temps et l'exécution du code est connu, pourtant la mesure de l'ampleur de la fuite d'information transmise par ce canal n'est pas triviale, ni à son époque, ni aujourd'hui.

```
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 }
```

Code 1.1 – Exemple de code vulnérable à une attaque temporelle

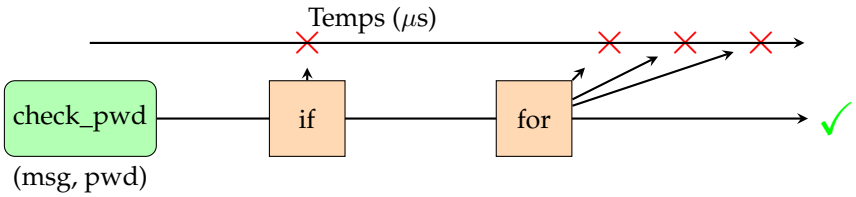
Si nous prenons le code présenté par le code 1.1, nous pouvons observer que la fonction `check_pwd` compare deux chaînes de caractères. Si elles sont de même longueur, elle les

1. Non Volatile Memory ou mémoire non volatile est un composant informatique qui conserve son contenu en l'absence d'électricité.
2. Electrically-Erasable Programmable Read-Only Memory ou mémoire morte effaçable électriquement et programmable.
3. Machine virtuelle qui exécute des programmes compilés en bytecode Java.
4. Le terme générique dans la recherche scientifique est «*time attack*». Une traduction plus précise serait «*attaque par chronométrage*». Nous choisissons ici d'utiliser le terme «*attaque temporelle*» car il est moins lourd et renvoie directement vers la faille exploitée plutôt que par la méthode employée.

compare caractère par caractère. Si elles sont de longueur différentes, la fonction retourne immédiatement `False`. Ainsi, si nous fournissons un mot de passe de longueur différente, le temps d'exécution sera constant et court. En revanche, si nous fournissons un mot de passe de même longueur, le temps d'exécution dépendra du nombre de caractères identiques consécutifs entre les deux chaînes. En effet, la fonction s'arrêtera dès qu'un caractère différent est trouvé. Ainsi, en mesurant le temps d'exécution pour différents mots de passe, un attaquant peut déduire des informations sur le mot de passe correct.

Nous pouvons synthétiser les exécutions de la fonction `check_pwd` en un graphe comme celui présenté par la figure 1.1. Chaque interruption de la fonction peut être observée et mesurée, permettant ainsi de régénérer le mot de passe. Bien sûr, la connaissance du protocole ciblé est requise, ou alors il faut réaliser un travail de rétro-ingénierie pour calibrer l'attaque.

FIGURE 1.1 – Suivi du temps d'exécution pour différents mots de passe



Cette méthode est plus efficace qu'une attaque par force brute. En effet, si le mot de passe est composé de 8 caractères de l'alphabet latin, alors il y a 256 possibilités par caractère, pour un total de $256^8 = 2^{64}$ possibilités. En revanche, si nous utilisons la méthode de l'attaque temporelle, le nombre de possibilités est réduit à $8 + 8 \times 256 = 2056$ possibilités. En effet, nous cherchons dans un premier temps à identifier la longueur du mot de passe, puis nous identifions ensuite caractère après caractère pour trouver le mot de passe. Des temps d'exécution courts correspondent à des cas d'échec, tandis qu'un allongement du temps d'exécution nous permet de déterminer une bonne piste.

Les attaques temporelles présentent la particularité d'être génériques. Tandis que les attaques décrites précédemment nécessitent des conditions d'accès ou d'initialisation plus importantes, cette classe d'attaque présente l'avantage d'être réalisable sur tous les types de systèmes, et notamment les systèmes accessibles par internet. La connaissance de cette menace est donc primordiale pour l'implémentation et la mise en service de produits sur Internet.

Par la suite du document, le terme "fuite" sera utilisé pour désigner un extrait du programme qui peut être exploité pour réaliser une attaque temporelle. Si nous reprenons le code 1.1, les branchements conditionnels lignes [4, 6] sont des fuites d'informations. C'est grâce à ces instructions que l'attaque décrite précédemment est réalisable.

Nous allons maintenant nous intéresser aux moyens et méthodes à notre disposition pour se protéger contre les attaques temporelles.

CHAPITRE 2

Protection

Ce deuxième chapitre montre les innovations nécessaires pour se protéger des attaques temporelles. Nous y découvrons les bonnes pratiques de programmation, les premiers outils automatiques de vérification de code ainsi que les limitations auxquelles est confronté le développeur qui souhaite être résistant à ces attaques.

2.1 Bonnes pratiques et usages

Face à la menace des attaques temporelles, quelles solutions peuvent être mises en place pour protéger nos systèmes informatiques ? Cette attaque a besoin d'un accès au système et d'un chronomètre. Comme nous sommes dans un contexte de systèmes accessibles par internet, altérer ou retirer l'accès signifie perdre en qualité ou supprimer le service proposé. Il faut donc que notre approche cible plutôt l'utilisation du chronomètre.

Il faut donc programmer de telle sorte que sur toutes les entrées possibles de notre système informatique aucune variation de temps ne peut être observée entre les exécutions. Trois méthodes existent pour pallier ce problème.

Programmation en temps constant

La programmation en temps constant ou «*Constant-Time Programming*», est une pratique de programmation qui vise à résoudre exactement ce problème. Directement lié à la complexité algorithmique, cette pratique modifie et adapte les algorithmes pour que toutes les opérations effectuées aient un temps d'exécution identique.

PORNIN [Por16] présente tous les éléments à adapter pour configurer un code respectant la politique de programmation en temps constant. Si les opérations élémentaires respectent "naturellement" cette politique ; les **accès mémoires**, les **sauts conditionnels**, les **opérations de décalages/rotations** et les **divisions/multiplications** sont les opérations à adapter en fonction de la plateforme cible. Les descriptions rapportées ci-dessous sont issues de [Por16].

Accès mémoire

Un chargement depuis la mémoire d'une information est une source de variation. Nous avons vu précedemment [Liu+15 ; Pes+16] que l'usage d'un cache mémoire est un canal d'accès pour réaliser une attaque. En effet, l'utilisation d'un cache permet de distinguer les appels entre les données déjà mises en mémoire ou pas. De plus, les changements de valeur dans celui-ci peuvent aussi être observés après exécution.

Décalage et rotation

Ces opérations binaires sont ou ne sont pas en temps constant en fonction des CPU sur lesquels le code est exécuté. Certains ont un "barrel shifter" qui permet d'effectuer directement les instructions correspondantes. Cela impacte directement les algorithmes dépendants de décalages logiques comme le chiffrement RC5.

Saut conditionnel

Les saut conditionnels sont des instructions qui, comme pour les accès mémoire, demandent de charger les adresses des instructions suivantes. Or, comme un compilateur tend à précharger les instructions suivantes, il va charger les deux côtés du saut conditionnel puis défausser la branche inutile ; ce qui entraîne un léger ralentissement. En revanche, il est important de noter que si le branchement est indépendant d'une variable secrète, il n'est pas nécessaire de le modifier. Par exemple si j'ai un compteur et que mon programme doit terminer après un certain nombre d'itérations, aucune fuite ne sera observée.

Division

Certaines architectures ont des instructions de divisions spécifiques qui permettent d'accélérer le calcul, les autres emploient des sous-programmes dédiées souvent optimisés en opération de masquage et de décalage. La norme C entraîne elle aussi de la confusion car elle impose $(-1)/2 = 0$; il faut donc être familier avec les spécificités du processeur pour affiner l'usage de cette opération.

Multiplication

Enfin, la multiplication, elle aussi dépendante des variables d'entrées, présente une fuite d'information importante. Mais les CPU les plus récents (rédigé en 2016) ont implémenté cette opération en temps constant. Cela suit l'évolution des compilateurs et des processeurs qui tendent à accélérer les opérations et réduire le nombre d'instructions total.

En reprenant ces règles, nous pouvons modifier notre exemple de code 1.1 et appliquer des modifications sur les lignes que nous avons déjà ciblées comme fuites d'informations. Les modifications sont libres au choix du concepteur. Voici une correction qui peut être réalisée :

```
1  bool check_pwd(msg,pwd) {
2    // Hachage
3      char msg_hash[SHA256_DIGEST_LENGTH]; sha256_hash_string(msg, msg_hash);
4      char pwd_hash[SHA256_DIGEST_LENGTH]; sha256_hash_string(pwd, pwd_hash);
5
6      // Comparaison
7      bool equal = true;
8      for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
9          equal &= msg[i] != pwd[i]
10     }
11     return equal;
12 }
```

Code 2.1 – Exemple de correction pour rendre un code résistant aux attaques temporelles

Nous voyons que le premier branchement a été remplacé par un hachage des paramètres d'entrées. Cette opération est considérée ici en temps constant mais peut ne pas

l'être. Il faut être vigilant sur toutes les briques d'algorithme que nous souhaitons utiliser. Enfin, le second branchement conditionnel est purement supprimé, le parcours des tableaux se fait entièrement.

Avec cette modification, nous avons un code 2.1 qui ne présente plus de fuite de données. Pourtant, nous pouvons avoir un doute sur l'usage de la fonction "*sha256_hash_string*". Si cette fonction n'est pas elle même implémentée selon la politique temps constant, nous avons alors introduit une nouvelle surface de fuite d'informations. Il faut vérifier notre code pour supprimer ce doute.

Outils de garanties

Plusieurs outils existent et peuvent être utilisés tous au long du processus de développement d'un système sécurisé. Cela peut être durant la phase de conception du code source, au moment de la compilation ou encore en vérification de la compilation.

Une solution légère est de ce servir du système libre «**Compiler Explorer**[1]». Avec à disposition un éditeur de texte, il est possible de voir comment sera généré le code assembleur. En reprenant une partie du code 1.1, nous pouvons voir sur la figure 2.1 que le choix du compilateur, ici sa version, introduit une légère modification. Ce changement n'est pas perceptible sans observation directe, il se perçoit directement grâce à la petite taille du code observé. Un autre exemple 2.2 est présenté à la fin du chapitre.

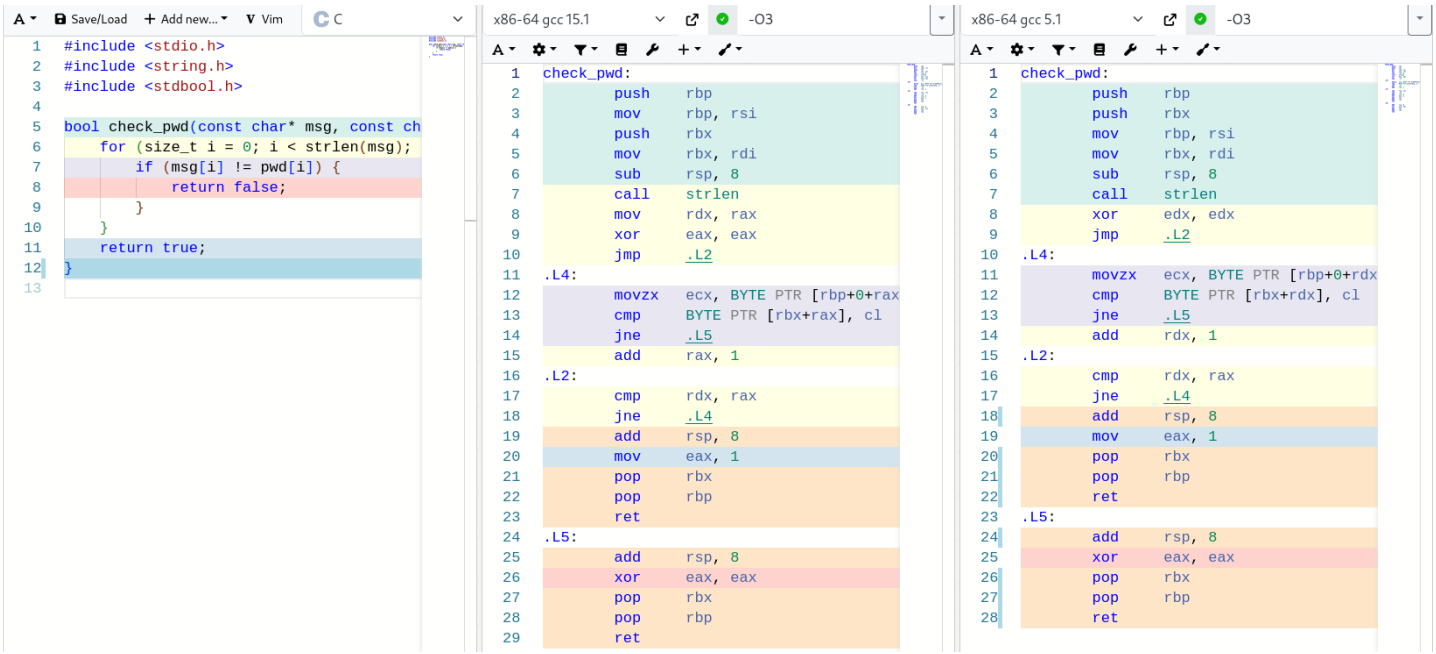


FIGURE 2.1 – Capture d'écran de comparaison de code assembleur x86_64 entre `GCC 5.1` et `GCC 15.1`

Si nous souhaitons faire une analyse à l'échelle d'un projet, ce parcours à la main des fonctions ou de morceaux de fonctions est réellement fastidieux. Il faut mieux déléguer ce travail à un outil conçu pour vérifier la présence de fuite.

Plusieurs articles référencent l'ensemble des outils existants [Jan+21 ; Gei+23] pour réaliser ce travail. Le tableau 2.1 de JANCAR, FOURNÉ, BRAGA, SABT, SCHWABE, BARTHE, FOUQUE et ACAR liste 24 outils en libre accès conçus pour détecter des failles par canal auxiliaire.

Ils sont listés alphabétiquement et ont précisé le type de fichier analysé (*Cible*), la méthode d'analyse réalisée (*Techn.*) et les garanties attendues de ces analyses (*Garanties*). Nous reviendrons plus en détail sur ces méthodes et leurs fonctionnements dans le chapitre 3.

1. `https://godbolt.org/`

TABLE 2.1 – 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, [Symbolique, Dynamique, Statistique] = type d'analyse.

Garanties (*Sécurité face aux attaques temporelles*) : ● = Analyse correcte, ▲ = Correcte mais avec des limitations, ○ = Aucune garantie, ★ = Vérification d'autres propriétés.

Outil	Cible	Techn.	Garanties
ABPV13 [Alm+13]	C	Formel	●
Binsec/Rel [DBR19]	Binaire	Symbolique	▲
Blazer [Ant+17]	Java	Formel	●
BPT17 [BPT17]	C	Symbolique	▲
CacheAudit [Doy+13]	Binaire	Formel	★
CacheD [Wan+17]	Trace	Symbolique	○
COCO-CHANNEL [Bre+18]	Java	Symbolique	●
ctgrind [Lan10]	Binaire	Dynamique	▲
ct-fuzz [HEC20]	LLVM	Dynamique	○
ct-verif [Bar+16]	LLVM	Formel	●
CT-WASM [Wat+19]	WASM	Formel	●
DATA [Wei+20 ; Wei+18]	Binaire	Dynamique	▲
dudect [RBV17]	Binaire	Statistique	○
FaCT [Cau+19]	DSL	Formel	●
FlowTracker [RPA16]	LLVM	Formel	●
haybale-pitchfork [Dis20]	LLVM	Symbolique	▲
KMO12 [KMO12]	Binaire	Formel	★
MemSan [Tea17]	LLVM	Dynamique	▲
MicroWalk [Wic+18]	Binaire	Dynamique	▲
SC-Eliminator [Wu+18]	LLVM	Formel	●
SideTrail [Ath+18]	LLVM	Formel	★
Themis [CFD17]	Java	Formel	●
timecop [Nei18]	Binaire	Dynamique	▲
VirtualCert [Bar+14]	x86	Formel	●

Une dernière solution serait d'utiliser un compilateur spécialisé qui produit un code assembleur sans fuite [Bor+21 ; RLT15] ou d'utiliser un compilateur formel comme *CompCert* [Bar+19]. Cette solution rencontre en pratique de nombreux problèmes que nous conservons pour la section 2.2 Limitations.

Écriture en code assembleur

Enfin, la dernière méthode pour obtenir un code sécurisé et sans fuite c'est de programmer directement en assembleur. De cette manière, nous avons un contrôle total sur le flot d'exécution de notre programme, nous pouvons insérer des optimisations qu'un compilateur pourrait ignorer. Écrire en assembleur requiert de connaître la plupart des opérandes disponibles pour l'architecture ciblée et les modèles des composants présents sur le support. Cela nous amène directement aux limitations induites par cette solution.

2.2 Limitations

Écrire en assembleur c'est écrire spécifiquement pour une architecture de processeur. Il faut connaître les instructions adéquates, les potentielles optimisations qui existent sans parler de la syntaxe particulère qui rend son développement plus lent. Travailler en assembleur c'est limiter la portabilité du code proposé. Or l'objectif derrière le développement d'une bibliothèque sécurisée est de pouvoir être employée par le plus de configurations possibles pour se protéger d'attaques.

Face à cette situation, nous pouvons choisir d'utiliser un compilateur spécialisé ([Bor+21 ; RLT15]). Et comme un serpent qui se mord la queue, nous voici à nouveau limités. Ces compilateurs peuvent ne pas supporter l'ensemble du jeu d'instruction d'une architecture, ils peuvent avoir besoin d'instructions supplémentaires (des annotations de code) pour réali-

ser leur compilation, ils peuvent ne pas implémenter les optimisations présentent sur les processeurs les plus récents.

Donc, si les compilateurs spécialisés ne sont pas envisageables, nous nous retrouvons à utiliser les compilateurs courants `GCC` et `LLVM+Clang` pour notre solution sécurisée. Nous devons donc programmer en respectant la politique temps constant. Et si cette pratique semble être notre solution, nous pouvons lire dans l'article de présentation de l'outil d'analyse Binsec *Binsec/Rel : Efficient Relational Symbolic Execution for Constant-Time at Binary-Level* :

Conclusion - [DBR19]

Nous avons découvert que `gcc -O0` et des optimisations de `clang` introduisent des infractions à la politique temps constant indétectées par les outils antérieurs

Cette, annonce discrète au sein du document, a ensuite été prise en compte par SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN qui a mené une enquête sur les bibliothèques cryptographiques sécurisées et résistantes aux attaques temporelles : [Sch+24]. La conclusion principale est que les compilateurs modernes sont devenus assez performants pour voir à travers les astuces employées et qu'une mauvaise utilisation d'optimisation implique l'introduction de faille de sécurité.

Voici un exemple communiqué par SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN auprès des chercheurs de HACL*. Nous pouvons voir deux fonctions dans le code 2.2, «*cmovznz4*» et «*FStar_UInt64_eq_mask*». La première appelle la seconde pour générer un masque qui sera ensuite appliqué aux paramètres de «*cmovznz4*». Nous avons ici une fonction qui agit comme un branchement conditionnel. Si `cin` vaut 1, alors $r = x$ sinon $r = y$.

```c
#include <stdint.h>

static inline uint64_t FStar_UInt64_eq_mask(uint64_t a, uint64_t b)
{
  uint64_t x = a ^ b;
  uint64_t minus_x = ~x + (uint64_t)1U;
  uint64_t x_or_minus_x = x | minus_x;
  uint64_t xnx = x_or_minus_x >> (uint32_t)63U;
  return xnx - (uint64_t)1U;
}

void cmovznz4(uint64_t cin, uint64_t *x, uint64_t *y, uint64_t *r)
{
  uint64_t mask = ~FStar_UInt64_eq_mask(cin, (uint64_t)0U);
  uint64_t r0 = (y[0U] & mask) | (x[0U] & ~mask);
  uint64_t r1 = (y[1U] & mask) | (x[1U] & ~mask);
  uint64_t r2 = (y[2U] & mask) | (x[2U] & ~mask);
  uint64_t r3 = (y[3U] & mask) | (x[3U] & ~mask);
  r[0U] = r0;
  r[1U] = r1;
  r[2U] = r2;
  r[3U] = r3;
}
```

Code 2.2 – Fonction de masquage issu de *HACL**

Avec le compilateur `RISC-V rv64gc clang 15.0.0`, si nous précisons les options de compilation `-O0` ou `-O1`, nous pouvons observer différents résultats. Le plus notable ici est l'apparition de l'instruction `beqz`, qui est un branchement conditionnel, ainsi que la suppression de la fonction de masquage «*FStar_UInt64_eq_mask*». Les optimisations appelées par l'option `-O1` identifient le masquage effectué et modifient le code pour accélérer son exécution. L'optimisation 2.2a suit les instructions précisées par le code source, de cette manière le compilateur réalise une compilation rapide. Au contraire de l'optimisation 2.2b qui réalise une analyse plus longue du code source, et donc a une compilation plus lente,

mais grâce à l'ajout des branchements successifs (les instructions `beqz`) permet une exécution plus rapide. Les options de compilation sont indiquées en annexe A.1[2].

```
cmovznz4:
        ...
        li      a1, 0
        call    FStar_UInt64_eq_mask
        not     a0, a0
        sd      a0, -56(s0)
        ld      a0, -40(s0)
        ld      a0, 0(a0)
        ld      a2, -56(s0)
        and     a0, a0, a2
        ld      a1, -32(s0)
        ld      a1, 0(a1)
        not     a2, a2
        and     a1, a1, a2
        or      a0, a0, a1
        sd      a0, -64(s0)
        ...
        ret

FStar_UInt64_eq_mask:
        addi    sp, sp, -64
        sd      ra, 56(sp)
        sd      s0, 48(sp)
        addi    s0, sp, 64
        sd      a0, -24(s0)
        sd      a1, -32(s0)
        ld      a0, -24(s0)
        ld      a1, -32(s0)
        xor     a0, a0, a1
        sd      a0, -40(s0)
        ld      a1, -40(s0)
        li      a0, 0
        sub     a0, a0, a1
        sd      a0, -48(s0)
        ld      a0, -40(s0)
        ld      a1, -48(s0)
        or      a0, a0, a1
        sd      a0, -56(s0)
        ld      a0, -56(s0)
        srli    a0, a0, 63
        sd      a0, -64(s0)
        ld      a0, -64(s0)
        addi    a0, a0, -1
        ld      ra, 56(sp)
        ld      s0, 48(sp)
        addi    sp, sp, 64
        ret
```

(a) Option `-O0`

```
cmovznz4:
        mv      a5, a1
        beqz    a0, .LBB0_2
        mv      a5, a2
.LBB0_2:
        beqz    a0, .LBB0_5
        addi    a6, a2, 8
        bnez    a0, .LBB0_6
.LBB0_4:
        addi    a4, a1, 16
        j       .LBB0_7
.LBB0_5:
        addi    a6, a1, 8
        beqz    a0, .LBB0_4
.LBB0_6:
        addi    a4, a2, 16
.LBB0_7:
        ld      a7, 0(a5)
        ld      a5, 0(a6)
        ld      a6, 0(a4)
        beqz    a0, .LBB0_9
        addi    a0, a2, 24
        j       .LBB0_10
.LBB0_9:
        addi    a0, a1, 24
.LBB0_10:
        ld      a0, 0(a0)
        sd      a7, 0(a3)
        sd      a5, 8(a3)
        sd      a6, 16(a3)
        sd      a0, 24(a3)
        ret
```

(b) Option `-O1`

FIGURE 2.2 – Comparaison du code 2.2 en fonction de différentes options de compilation données au compilateur, réalisée avec l'aide de *Compiler Explorer*.

Avec ces solutions applicables, il nous faut maintenant étudier leurs impacts. Nous allons voir quels moyens permettent de vérifier la sécurité d'un programme.

2. `https://gcc.gnu.org/`

CHAPITRE 3

Analyse de programmes et méthodes de vérifications

Nous allons étudier les moyens à notre disposition pour réaliser une analyse pertinente et efficace d'un programme résistant aux attaques temporelles.

3.1 Modélisation d'une attaque

En sécurité informatique, la première étape, essentielle avant de développer une solution, c'est de produire un modèle du danger contre lequel nous souhaitons se défendre. On parle parfois de *modèle de fuite*. Cette étape de synthèse et d'abstraction est importante pour identifier les risques encourus par le futur système, souvent en identifiant les points de fuite employés par les attaques déjà publiées. SCHNEIDER, ZEITSCHNER, KLOOS, LEMKE-RUST et IACONO [Sch+25] nous donne trois modèles d'adversaires à considérer lorsqu'on souhaite se défendre contre les attaques temporelles :

TABLE 3.1 – 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 de plusieurs caches mémoires.
Par récupération de traces	Suivi des appels de fonctions, des accès réussis ou manqués à la mémoire.

Ces types d'attaques forment une base pour la conception de nos modèles d'attaquant. Considérer le mode opératoire «récupération de traces» induit un modèle plus puissant. Des travaux comme ceux de GAUDIN, HATCHIKIAN-HOUDOT, BESSON, COTRET, GOGNIAT, HIET, LAPOTRE et WILKE [Gau+23] portent directement sur des améliorations matérielles permettant une défense contre ce modèle. Considérer un hypothétique attaquant plus puissant, avec des accès à des ressources supplémentaires, permet de concevoir un système plus sûr. Certains outils comme [HEC20 ; Wei+18] ou cette étude [Jan+21] exploitent notamment cette mécanique pour attester de la sécurité d'un programme.

Puis, avec ces modèles et les contre-mesures connues, nous pouvons constituer un ensemble de règles qui vérifient 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) accédée.
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 à sécuriser. C'est ainsi 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 correcte de programmes complexes (*i.e.* primitives cryptographiques asymétriques).

Formalisation de modèle - [DBR19]

Si nous voulons concevoir un modèle formel, nous pouvons nous appuyer sur l'article *Secure Compilation of Side-Channel Countermeasures : The Case of Cryptographic "Constant-Time"* [BGL18].

Nous commençons par définir un programme. Il s'agit d'une suite d'instructions binaires. Et une instruction est une action sur la mémoire. Cela nous permet de définir notre programme comme une suite de configurations (l, r, m) ; l la ligne d'instruction, r le dictionnaire de registre et m la mémoire. La configuration initiale est définie 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 registres 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 $c_0 \xrightarrow[t]{} c_1$, c_0 et c_1 sont deux configurations successives, $\rightarrow$ la transition entre les deux et t une fuite émise par cette transition. Notons que certaines instructions ne produisent pas de fuites.

Une fois ce préambule installé nous définissons formellement le comportement de nos instructions. Regardons par exemple comment se formalise un chargement :

FIGURE 3.1 – Instruction `chargement`

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

Ici, l'évaluation de l'expression `e` sur une configuration (l, r, m) produit une fuite de la valeur bv. En haut nous retrouvons la notation de l'opération effectuée et au-dessous la formalisation de la fuite : $t \cdot [bv]$ signifie que la valeur bv s'ajoute à la liste des fuites. Ce second exemple 3.2 présente une opération de branchement en fonction de e vers les instructions l_1 et l_2. Nous voyons que la valeur est différente de zéro, ce qui nous produit une fuite vers l'instruction l_1. Cette fuite est à ajouter à notre liste t.

FIGURE 3.2 – Instruction `branchement`

$$\text{T-ITE} \quad \frac{P.l{=}\mathit{ite}\ e\ ?\ l_1 : l_2 \qquad (l,r,m) \vdash_t \text{bv} \qquad \text{bv} \neq 0}{(l,r,m) \underset{t\,[l_1]}{\longrightarrow} (l_1,r,m)}$$

Nous pouvons retrouver l'ensemble des règles formelles en Annexe A.1.

3.2 Analyse d'un programme

Nous savons concevoir un modèle pour contrôler ou détecter les erreurs. Nous pouvons maintenant concevoir notre analyse pour vérifier ce modèle sur un programme. Plusieurs techniques de vérification existent et nous allons les passer en revue : [Gei+23].

Analyse statique

Cette méthode consiste à déduire le fonctionnement d'un programme. Nous souhaitons vérifier que son fonctionnement respecte les propriétés de sécurité définies préalablement. Cette analyse sans exécution réalise une simulation du programme en explorant les chemins d'exécution possibles. De fait, les résultats obtenus sont souvent approximés car une exploration totale peut se révéler irréalisable. Historiquement il s'agit de la première méthode étudiée/employée et depuis elle a été dérivée en plusieurs approches.

Non-interférence Pour renforcer les résultats obtenus et réduire le nombre de faux positifs nous pouvons vérifier la propriété de non-interférence. Cette propriété est inhérente aux programmes. Un programme a des entrées et des sorties. Celles-ci peuvent être classées *faibles* (peu importantes) ou *hautes* (données secrètes, sensibles). Un programme est non-trois modèles d'adversaires à considérer lorsqu'on souhaite se interférent si et seulement si pour n'importe quelle entrée faible le programme ressort la même sortie faible peu importe les valeurs des entrées hautes qui peuvent être précisées.

Appliqué à une analyse statique pour la vérification de programme, la mesure des ressources employées par l'ordinateur permet d'avoir une sortie faible pour comparer le comportement d'un programme en fonction de ses entrées (ici considérées secrètes).

Self-Composition[1] La self-composition consiste à entrelacer deux exécutions d'un programme P avec différents ensembles de variables secrètes dans un seul programme autocomposé $P; P'$. Des solveurs peuvent alors être utilisés pour vérifier la propriété de non-interférence. Cette approche a été utilisée par ALMEIDA, BARBOSA, PINTO et VIEIRA [Alm+13] pour vérifier manuellement des exemples limités, nécessitant de nombreuses annotations pour limiter l'explosion (quadratique) des états à comparer et explorer. [DBR19] emploie cette approche associée à des solveurs SMT pour vérifier uniquement les propriétés définies dans leur modèle. La restriction aux propriétés temps constant permet l'exploitation de cette méthode sans le contrecoup de l'augmentation de la taille des formules.

Systèmes de types Cette approche diffère des précédentes car elle nécessite un travail supplémentaire du développeur. Il doit ajouter la spécification `secret` aux valeurs employées pour que cette information se diffuse dans le compilateur et que des mesures adaptées soient effectuées au niveau du binaire. Cette approche est intéressante car elle permet une flexibilité plus importante lors de la production du code et permet de s'abstenir des contre-mesures décrites au chapitre 2. En revanche elle nécessite un compilateur spécialisé et aucune vérification sur le binaire produit n'est effectuée.

Interprétation abstraite Un programme est (généralement) trop complexe pour être entièrement formellement vérifié, donc il y a une sur-approximation des états atteignables par l'analyse. Ainsi, si l'analyse approximée est sécurisée alors le programme est sécurisé. Cette approche se retrouve dans CacheAudit [Doy+13] : modélisation par un graphe de flot de l'état des caches, de la mémoire et des successions d'évènement.

Exécution symbolique L'exécution symbolique consiste à exécuter le programme avec des entrées symboliques. Les chemins explorés sont associés à une formule logique, et un solveur vérifie si un ensemble de valeurs concrètes satisfait les formules générées. Cette méthode est utilisée pour vérifier l'absence de dépendance aux secrets dans les comportements temporels ou mémoire du programme.

Analyse dynamique

L'analyse dynamique emploie la preuve par l'exemple pour garantir la sécurité du programme cible. Nous exécutons le programme et nous collectons sa trace : informations issues des évènements (accès mémoire, sauts,*etc*) rencontrés au fur et à mesure de l'exécution. Les approches diffèrent dans la collecte et la production de ces traces.

Trace unique Explorer tous les comportements d'un programme est coûteux en temps, et pour les besoins du développement il peut être préférable d'étudier quelques cas particuliers entièrement. Cette approche simplifie le modèle de l'attaquant et réalise sa vérification plus rapidement. `ctgrind` [Lan10] réutilise l'analyse dynamique de Valgrind pour vérifier les propriétés temps constant. Pour ajouter de la précision, il est possible d'utiliser l'exécution symbolique pour rejouer la trace avec le secret comme valeur symbolique et vérifier la violation du temps constant (CacheD [Wan+17]).

1. Construction personnelle, le terme anglais est conservé.

Comparaison de traces Les tests statistiques peuvent vérifier si différents secrets induisent des différences significatives dans les traces enregistrées. Des outils comme DATA [Wei+20] ou MicroWalk [Wic+18] utilisent diverses méthodes statistiques ou d'apprentissage pour détecter et localiser les fuites. D'autres outils comme dudect [RBV17] enregistrent simplement le nombre total de cycles d'horloge et comparent leur distribution selon les secrets.

Le fuzzing peut aussi être utilisé pour trouver des entrées maximisant la couverture et la fuite via canal auxiliaire, comme dans ct-fuzz [HEC20].

3.3 Outils de vérifications

Nous avons présenté rapidement de nombreux outils capable d'effectuer de la vérification de binaire (voir la table 2.1) et tous au long de la section précédentes nous avons présenté les méthodes employés par ceux-ci.

Ils sont tous conçus pour analyser spécialement type de fichier (C, Java, LLVM, *etc*). Dans notre cas nous ciblons des binaires. Nous pouvons donc nous tourner vers ces candidats :

- Binsec
- CacheAudit
- ctgrind
- DATA
- dudect
- KMO
- MicroWalk
- timecop

Outils obsolète ou inadéquat

`ctgrind` et `timecop` sont tous les deux bâtis sur *Valgrind*. L'analyse se base sur l'outil de détection d'erreur mémoire propre à *Valgrind* : *Memcheck*. Celui-ci détecte les branchement conditionnels et les accès mémoire calculés vers des régions non initialisée. Les vulnérabilités sont trouvées en marquant les variables secrètes comme non définies, au travers d'une annotation de code spécifique. Puis, durant l'exécution, *Memcheck* associe chaque bit de données manipulées par le programme avec un bit de définition qu'il propage tout au long de l'analyse et qu'il 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.

`KMO` détecte les fuites d'information pour des attaques qui ciblent le cache. Cet outil développé en 2012 et plus maintenu utilisé une représentation interne du binaire pour compter les bits d'informations qui peuvent fuiter. En initialisation, l'outil fait deux estimations (borne supérieure et inférieure) de la quantité d'informations qu'un attaquant peut extraire en observant les adresses mémoires présentes dans le cache après l'exécution d'un programme. Grâce à l'observation qu'une estimation haute du nombre d'états atteignables par le programme est aussi une estimation haute du nombre de bits que peut connaître un attaquant, on peut déterminer de la sécurité d'un programme.

Outils statistiques

`dudect` détecte les fuites temporelles par des mesures répétées du temps d'exécution et une comparaison à l'aide d'un test statistique. Le binaire est exécuté sous deux classes différentes d'entrées secrètes : un ensemble avec des valeurs constantes et un ensemble avec des valeurs sélectionnées aléatoirement avant chaque mesure. Les temps d'exécution de la fonction cible sont alors enregistrés en sondant les compteurs de cycles CPU. Si les deux distributions sont distinguables alors une fuite est rapportée uniquement si la valeur dépasse un seuil (prédéfini). Les garanties apparaissent à partir d'un grand nombre de mesures.

`MicroWalk` enregistre plusieurs exécutions de la fonction cible avec différentes entrées. La trace enregistrée contient les cibles des branches et les adresses mémoire rencontrées pendant l'exécution. Des scores d'information mutuelle (MI) sont ensuite calculés entre la trace de fuite et l'ensemble des entrées, fournissant une quantification du nombre de bits d'entrée divulgués. `MicroWalk` propose différents compromis entre granularité du score MI et performance, allant de l'information mutuelle sur l'ensemble du programme (quantifiant grossière des fuites) jusqu'à des instructions individuelles (pour localiser précisément les fuites).

`DATA` identifie les vulnérabilités par canaux auxiliaires basées sur les adresses grâce à une analyse dynamique. Premièrement, une phase de collecte des traces d'adresses. En comparant ces traces, il identifie des différences au niveau des adresses, indiquant d'éventuelles fuites. Cependant, de nombreuses différences proviennent des entrées publiques et ne sont donc pas critiques. Pour filtrer ces faux positifs, `DATA` utilise des tests statistiques. La deuxième phase teste si les différences dépendent de la clé privée en comparant des traces générées à partir d'une clé fixe avec des traces générées à partir de clés variables. Ce test entre fixe et aléatoire nécessite un contrôle sur la variable secrète. Enfin il y a une phase de classement des fuites pour détecter les relations entre les traces d'adresses et le secret.

Outils formels

`CacheAudit` prend en entrée un binaire de programme et une configuration de cache, et il en déduit des garanties de sécurité formelles et quantitatives adaptés à un jeu de modèles d'attaquants par canaux auxiliaires (observation des états de cache, des traces d'utilisation de caches et de chronométrage des temps d'exécution). Le principale défaut est l'usage limité à l'architecture x86_64.

`Binsec` est une plateforme d'outils. On peut concevoir des scripts ou des extensions pour diriger les analyses qu'il effectue. Il a notamment une extension qui permet d'exploiter l'exécution symbolique pour vérifier les propriétés de temps constant d'un binaire. `Binsec` détecte l'architecture du binaire puis convertit les instructions en une représentation interne (RI) codée en DBA. À partir de la RI, il réalise une exécution symbolique avec une vérification de la non-interférence et vérifie qu'aucune dépendance aux entrées secrètes n'existe.

Choix de notre outil d'analyse

Au regard du tour d'horizon des outils que nous venons d'effectuer, seul `Binsec` peut être retenu pour la suite du projet. L'analyse n'est pas statistique, nous permet de couvrir plusieurs architectures, nous pouvons l'adapter à tous les binaires qui peuvent être issus d'une bibliothèque cryptographique. Un défaut que nous pouvons observer est sa complexité. `Binsec` est une platefome d'outils (27) et prendre en main toutes ses possibilités n'est pas évident.

Ce première partie ce conclut avec des réponses pour nos deux premières questions de recherche. Conserver les garanties de sécurité tout au long de la compilation est réalisable mais ne permet pas d'avoir une service utilisables sur différentes architectures. D'un autre côté nous avons pu mettre la main sur un outil d'analyse de binaire qui nous permettra d'effectuer la vérification d'une bibliothèque cryptogaphique. Nous allons maintenant nous pencher vers la réalisation d'un processus d'automatisation de la vérification d'un binaire.

CHAPITRE 4

Besoins et contraintes, préavis de conception

Nous allons maintenant présenter les raisonnements en amont de la conception d'un outil de détection automatique de failles par canal auxiliaire de type temporel pour l'analyse d'une bibliothèque cryptographique.

4.1 Identification des besoins et spécificités

Nous avons pu voir grâce aux chapitres précédents que la conception et l'implémentation d'un système sécurisé est un problème difficile. Une première étape est de concevoir des primitives et des protocoles mathématiquement sécurisés. Une seconde étape est de s'assurer que leurs implémentations sont effectivement sécurisées, d'un point de vue :

— mathématique contre des attaques logiques (aspect fonctionnel : le code implémente correctement les bons concepts cryptographiques)
— matériel, contre des attaques très bas niveau (les attaques temporelles)

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

Point de départ

En reprenant ces deux étapes, nous identifions les possibilités pour un développeur pour concevoir un système résistant à ces attaques temporelles.

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 conçoivent et maintiennent des bibliothèques cryptographiques, des boîtes à outils qui proposent les briques de sécurité nécessaires aux systèmes sécurisés.

Plusieurs bibliothèques existent [AHa98 ; Por16 ; Pol+20] et remplissent différents objectifs : rétrocompatibilité, politique temps constant, *etc*. Notre choix est à réaliser en fonction des spécificités du produit que nous cherchons à déployer.

La seconde étape est à distinguer en deux parties. Cette opération de vérification de la sécurité de l'implémentation peut être réalisée sur le produit fini et sur les bibliothèques employés par le produit. Comme introduit, cette étape a pour objectif la vérification formelle du code du programme et la vérification logicielle au niveau binaire.

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 mathématiques de la sécurité de son implémentation. Comme présenté en Projets supports, elle est programmée en F*. Le projet permet une exploitation en C et directement avec des binaires [Zin+17].

En revanche, la seconde étape de la seconde partie nous demande d'effectuer une vérification au niveau binaire. Si certaines parties de cette bibliothèque 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**. Cet outil est implémenté en Ocaml et il est maintenu par une équipe de chercheurs et d'ingénieurs du CEA-List de Saclay.

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 qu'elle sera la première bibliothèque cryptographique formellement sûre et vérifiée résistante aux attaques temporelles.

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'instructions à adjoindre. Avec ce point de départ, nous commençons à construire notre carnet de spécifications.

Fichier binaire. Il faut donc des fichiers binaires à fournir à Binsec. Or plus un binaire est imposant, plus son analyse sera difficile. Et comme Binsec emploie l'analyse symbolique, les formules se complexifient, le coût en mémoire augmente comme le nombre de branchement à explorer. Le temps d'analyse évoluent exponentiellement. 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, nous pouvons essayer de concevoir un binaire par fonction analysée. Nous distribuons ainsi l'analyse et nous parcourons ainsi toutes les fonctions présentes dans la bibliothèque.

Analyse correcte. En se rappelant comment fonctionnent les optimisations d'un compilateur (voir le tableau A.1), il nous faut être attentif avec certaines qui simplifient/modifient le code. Pour garantir la correction de nos analyses binaires, le fichier testé doit effectuer correctement un appel à la fonction analysée pour qu'il n'y ait aucune simplification/suppression de la part du compilateur.

```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 4.1 – Code d'analyse de la fonction Hacl_AEAD_Chacha20Poly1305_Simd128-_encrypt, testé lors de la prise en main de Binsec et HACL*

De même, comme nos fichiers analysés appartiennent à la bibliothèque extérieure HACL*, l'emploi de l'option `-static` est nécessaire pour prévenir la mise en place de liens vers la

bibliothèque partagée dans le fichier 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 nous désassemblons un fichier binaire. Retirer cette option lors de la compilation, c'est s'imposer des lourdeurs et rallonger le temps requis pour la vérification manuelle d'un fichier.

Couverture de compilateur. Les travaux de SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN [Sch+24] ont clairement mis en évidence que le choix du compilateur est à considérer. Nous allons pouvoir identifier quel compilateur nous permet d'avoir le plus de fichiers binaires sécurisés. Cette analyse nous permet d'identifier les limites de la pratique de la programmation en temps constant.

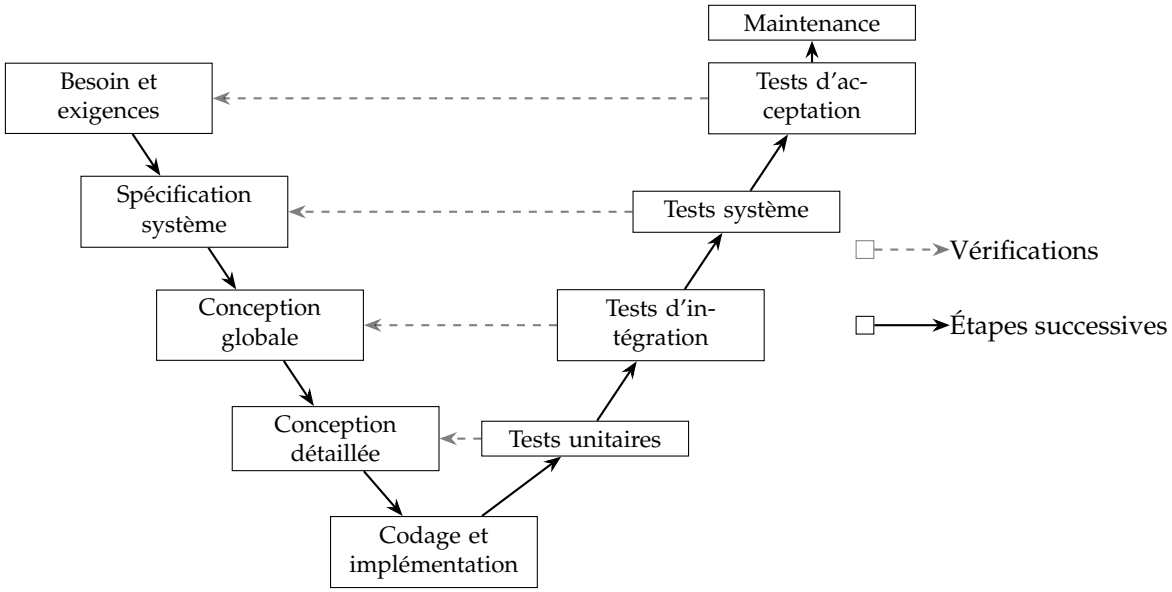
Couverture d'architectures. x86_64 et ARM sont les architectures matérielles les plus répandues dans le monde. Étendre l'analyse vers différentes plateformes et observer les différences qui émergent nous permettra d'avancer dans la direction de la conception d'une bibliothèque cryptographique universelle. Nous pouvons aussi étendre cette analyse vers d'autres architectures comme PowerPC ou RiscV.

Automatisation. Faire cette analyse sur un fichier binaire, comme le code 4.1, avec trois axes de complexité (complétude, de la couverture d'architectures et des compilateurs) n'est pas envisageable à la main. Il faut absolument que cette analyse soit automatisée.

4.2 Initialisation et tests variés

Dans le cadre de la programmation sécuritaire, où sont développés les systèmes avec pour objectif d'un accident par siècle (métros automatiques, trains, avions…), les projets sont conçus selon le principe du cycle en V. Cette méthode au contraire de la méthode Agile permet de prévoir tous les cas de figure et d'usages, nous permettant de nous épargner les problèmes de correction de bogues.

FIGURE 4.1 – Cycle en V



Appliquer cette méthode à l'entièreté de ce projet n'est pas envisageable à cause du coût temporel qui est très élevé. Nous nous concentrons sur la réalisation d'une preuve de concept avec un produit minimal mais opérationnel. Le développement sera concentré sur l'objectif d'automatisation. Le développement d'outils permettant la réalisation des objectifs des couvertures nécessiteront un travail futur.

Identification des besoins et exigences

Nous avons déjà conçu notre carnet d'exigences. En revanche nous ne connaissons pas le comportement des outils que nous souhaitons employer. La première opération est donc

de s'approprier le fonctionnement de ceux-ci. Le code 4.1 est un exemple de test réalisé dans cette phase du projet.

Binsec est un outil utilisable de manière privilégiée au travers d'un terminal. Il s'invoque avec son alias, le binaire à analyser et les options qui seront effectuées :

```
$ binsec -sse -sse-script $(BINSEC_SCRIPT) -checkct $(BINARY)
```

Code 4.2 – Commande Binsec basique

L'option `-sse` permet d'activer l'analyse par exécution symbolique, `-sse-script` associé à un fichier (ici `BINSEC_SCRIPT`) permet d'instruire notre analyse, préciser des stubs[1] et des initialisations. Enfin `-checkct` active la vérification de la politique 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'instructions qu'admet l'option `-sse-script`. Un tutoriel complet accessible depuis sa page officielle[2] nous a permis de comprendre le fonctionnement de l'outil Binsec.

```
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 4.3 – Instructions permettant de trouver le mot d'un passe d'un binaire d'exercice

Ce code présenté ici est un exemple d'usage de Binsec et permet de réaliser une attaque sur un binaire issu d'une plateforme d'apprentissage à la sécurité logicielle[3]. L'exercice consiste 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 4.4 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 pour objectif de vérifier les résultats apportés par [Sch+24] concernant une fuite présente sur la fonction «*FStar_UInt64_eq_mask*» et d'étendre cette analyse à d'autres architectures. Dans une première démarche d'automatisation, ce code a été généré automatiquement par un script shell. Nous pouvons voir que l'analyse ne parcourt pas l'entièreté du binaire, seulement 8 sections sont chargées (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 adresse 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 4.4 – Instructions permettant d'analyser le code 2.2 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

Les tableaux qui suivent présente une courte étude sur la fonction `cmovznz4` de HACL*. Nous pouvons voir les version des compilateurs utilisés, en vert les analyses qui se sont achevées et concluent que le binaire est sécurisé, en rouge que le binaire présente une (ou plus) faille et en orange que l'analyse n'a pu finir.

FIGURE 4.2 – 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	✓	✓	✓	✓	✓

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

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	✓	×	✓	×

FIGURE 4.3 – Tableau de résultats d'analyse Binsec pour architecture Risc-V

Nous comprenons, à la lecture du tableau 4.2, que la politique temps constant est considérée respectée par Binsec sur les versions testées ainsi que pour les différentes options de compilation. Ce résultat est encourageant pour la suite du projet. D'un autre côté, les résultats du tableau 4.3 sont indéniables : la version 19.1.7 de clang rend le code source perméable à des attaques temporelles.

Identification de défaut

Pour construire le tableau 4.3, plusieurs alertes se sont levées et ont 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 gérés. 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 par l'envergure de ce projet, il est possible que d'autres erreurs dues à Binsec soient découvertes. 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 conception de tests manuels.

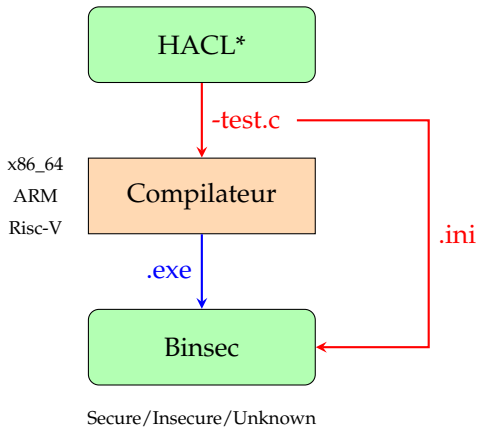
a. Acronyme anglais pour Architecture de Jeu d'Instruction, désigne l'ensemble des instructions assembleur associées à une architecture.

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

Nous observons ici une manifestation indéniable des précédents résultats 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, donnée dans le code source, indique au compilateur de désactiver ses optimisations pour le code contenu entre les deux balises `off`,`on`. Cette solution entraîne des pertes de performance et des ralentissements quant au temps de compilation et à l'usage des ressources. Il est donc préférable de l'utiliser avec parcimonie.

Après avoir ciblé notre besoin, les exigences associées et effectué des tests pour comprendre le processus à automatiser, nous pouvons synthétiser la démarche avec la figure 4.4 : depuis HACL*, nous extrayons une fonction qui sera testée, nous fabriquons le fichier de test en C, nous identifions les paramètres secrets et nous concevons le script adéquat pour Binsec, nous compilons le fichier C à notre guise et nous terminons par l'analyse Binsec.

FIGURE 4.4 – Flot de travail de l'outil d'analyse à concevoir



Nous allons maintenant nous pencher sur le procédé de conception de notre outil de détection automatique de failles temporelles.

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

CHAPITRE 5

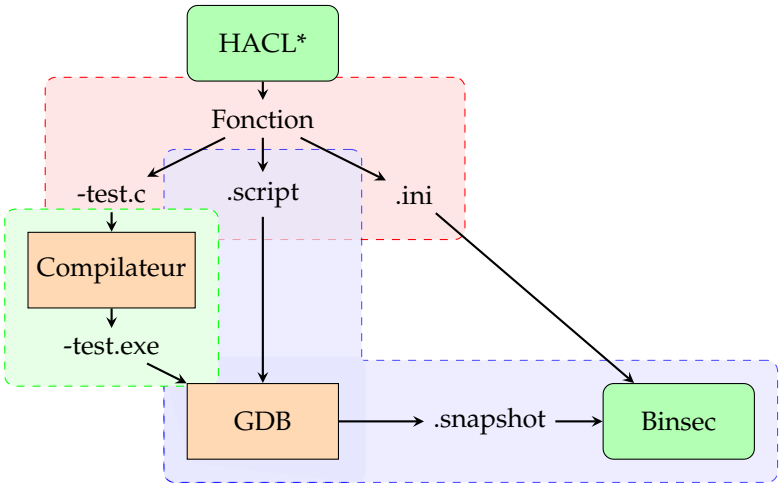
Implémentation d'Érysichthon

Nous avons nos spécificités techniques et nous savons quelle forme notre outil doit avoir (figure 4.4). Nous pouvons commencer par synthétiser les opérations nécessaires puis nous les implémenterons pour avoir un outil fonctionnel.

5.1 Planification et préparations

Nous allons normaliser les étapes nécessaires enfin que Binsec analyse entièrement un fichier et nous renvoie un parmi [`secure`, `unknown`, `insecure`]. Nous allons concevoir des protocoles présentant la méthodologie à suivre pour atteindre cet objectif. Nous commençons par le protocole `x86_64`, la première architecture que nous avons étudiée. Ce protocole est particulier. Depuis la version **0.5.0** de Binsec il est possible de fournir un «cliché mémoire»[1] pour accélérer l'analyse. Nous utiliserons donc cet avantage pour l'intégrer à notre graphe d'éxécution. La machine sur laquelle ce projet est développé est sur une architecture x86_64. Cela nous permet d'utiliser l'outil GDB pour la génération de clichés mémoire.

FIGURE 5.1 – Protocole pour analyser des fichiers compilés en x86_64



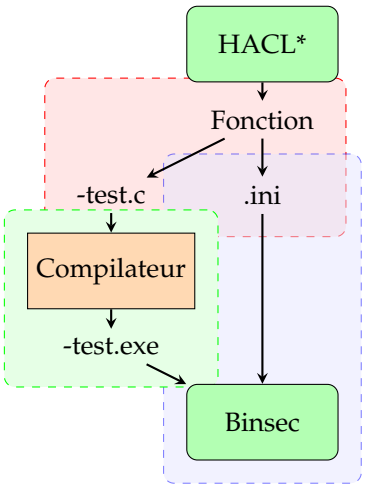
Ce graphe modélise la chaîne d'étapes nécessaires à l'obtention d'une analyse Binsec pour une fonction que nous ciblons. Plusieurs zones sont distinguées. La zone verte correspond à l'étape de compilation, la zone bleue à l'étape de préparation de l'analyse et la zone rouge à la synthèse de fichiers (de tests et d'instruction pour l'analyse de Binsec). Ce choix de couleur est adapté à la difficulté attendue de chaque étape. L'opération de compilation consiste en une commande. L'opération de préparation à l'analyse consiste simplement en

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 analysé, généralement par un débogueur.

deux commandes : un appel à GDB avec le binaire puis un appel à Binsec avec le cliché mémoire et les instructions d'analyse.

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

FIGURE 5.2 – Protocole générique d'analyse



Dans ce contexte, une question se pose : est-ce que la conception des scripts 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 5.1 ?

En effet, l'importance de cette question se révèle lorsque nous changeons d'architecture et que nous devons nous passer de clichés mémoire. Sur notre machine en x86_64, si nous analysons un fichier compilé en ARM, alors nous pouvons rencontrer des appels à des fonctions gérées par le mécanisme d'`IFUNC`. Or la résolution de ces fonctions est gérée dynamiquement lors de l'exécution du programme. Ce mécanisme permet d'utiliser des implémentations optimisées en fonction des configurations du système d'éxécution. Or comme Binsec réalise une analyse symbolique du programme, il faut lui spécifier quelles fonctions correspondent aux `IFUNC` qu'il peut croiser. Pour illustrer ce point, observons le script nécessaire pour une vérification de la fonction «Hacl _AEAD _Chacha20Poly1305_Simd128_encrypt» compilé vers ARMv8.

```
1  load sections .plt, .text, .rodata, .data, .got, .got.plt, .bss fr-
2  om file
3
4  secret global input1, aad1
5
6  @[0x00000048f008 ,8] := <__memcpy_generic>
7  @[0x00000048f018, 8] := <__memset_generic>
8  @[0x00000048f030 ,8] := <__memcpy_thunderx2>
9
10 starting from <main>
11 with concrete stack pointer
12
13
14 halt at <exit>
15 explore all
```

Code 5.1 – Script d'instruction pour analyser un binaire compilé vers ARM

Les lignes 5 à 7 sont présentes pour indiquer les branchements à effectuer par Binsec lorsqu'il rencontre ces adresses. Cette opération automatiquement exécutée lors de l'initialisation de l'exécution, doit ici être précisée avec les fonctions présentes dans le binaire. Automatiser ces affectations peut être difficile et nécessiter quelques outils d'analyse sup-

plémentaires pour repérer les adresses qui ont besoin d'être réaffectées et leur attribuer les fonctions les plus adaptées.

Nommer un outil

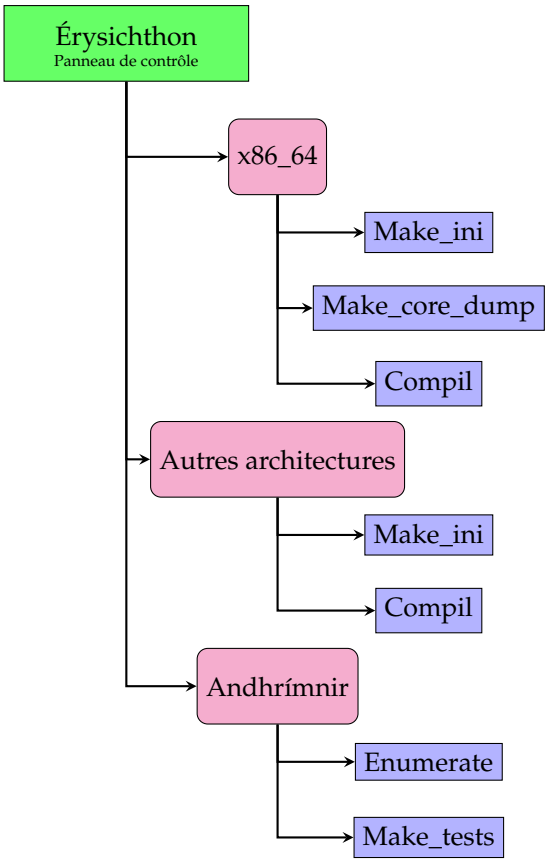
Rapidement il a fallu trouver un nom pour ce projet, l'appeler par "Notre outil… " devenait lourd et redondant entre les réunions hebdomadaires. En revanche trouver LE nom adéquat n'est pas une chose aisée, il peut être dû à une blague, une référence ou plus simplement être lié au sens du projet. Dans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

Érysichthon est un 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 il sera retenu pour la suite du projet.

Conception d'Érysichthon

Nous avons vu les protocoles nécessaires pour construire une analyse complète. Nous avons fait des tests pour comprendre le fonctionnement de Binsec et comment doivent être déclarées les fonctions de HACL*. Nous passons donc en phase de conception et construisons notre outil *Érysichthon*. Il sera une combinaison de script Python, script shell et de Makefile. Nous appelons module un ensemble de scripts qui réalise une tâche au sein d'*Érysichthon*. Nous présentons sur la figure 5.3 comment s'organisent les modules et les tâches qu'ils effectuent.

FIGURE 5.3 – Structure des modules d'Érysichthon



Nous retrouvons les différentes étapes de nos protocoles représentées par des rectangles symbolisant les modules associés depuis leurs noeuds respectifs : «`Make_ini`» pour la génération des 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 d'Érysichthon est particulier car il est lui aussi baptisé. Ce module fabrique les fichiers qui seront compilés puis analysés par Binsec. Son nom est celui du cuisinier des dieux de la mythologie nordique, un travail répétitif et quotidien qu'il réalise ici pour notre outil.

Ce module est nommé car il constitue un projet dans le projet, sa conception seule a pris plus de la moitié du temps de développement total. C'est un outil qui, à partir d'un projet en C, est capable de générer automatiquement des tests qui compilent et peuvent ensuite être proposés à des outils d'analyse binaire. Ce module est aggrégé à Érysichthon mais peut être porté vers d'autres 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), il y a ici une garantie quant à la complétude des tests produits. Toutes les fonctions présentes dans le projet C analysées auront un test associé.

Ce module, comme son grand frère, est fonctionnel et abouti. En revanche, il nécessite quelques opérations manuelles supplémentaires et quelques améliorations pour pouvoir supporter n'importe quel projet C. Additionnellement il possède quelques optimisations propres à HACL* permettant d'accélérer la mise en service d'Érysichthon (fichiers à ignorer spécifié en dur dans le code et certain chemins d'accès aux informations *e.g.* : si Andhrímnir a besoin des informations de `Spec_Hash_Definitions_hash_alg` en réalité elles sont définis dans `Hacl_Streaming_Types`).

5.2 Conception et usages

Nous commençons par le petit frère, Andhrímnir. Il fonctionne avec une phase d'initialisation «`Enumerate`» et une phase de production de tests «`Make_tests`», elle-même découpée en plusieurs étapes. La génération de **548** fichiers de tests est réalisée en moins de deux secondes.

Enumerate

Cette étape, de réalisation très simple, consiste à identifier toutes les fonctions pour lesquelles un fichier test sera généré. Comme HACL* génère automatiquement son code C, nous exploitons cette particularité pour lister efficacement les fonctions. L'opération actuellement réalisée est de lister l'ensemble des fichiers "**.h**" contenu dans le répertoire cible. Ensuite un parcours et une lecture de ceux-ci nous donnent toutes les fonctions de l'API[2] d'HACL* et d'avoir une couverture complète du projet.

C'est lors de cette étape que nous spécifions les fonctions à tester (ou nous pouvons aussi retirer des fonctions de la chaîne de production). Ce garde-fou permet d'accélérer l'obtention des résultats finaux et d'aider grandement lorsque nous souhaitons déboguer.

Make_test

Le modèle de fichier que nous construisons est similaire aux tests minimaux préalablement réalisés. Des paramètres, une déclaration de fonction et un appel à la fonction `exit`, c'est notre recette pour une analyse simple. Binsec réalise une analyse symbolique, il ignore donc la valeur réelle des entrées. Notre objectif avec notre recette est de concevoir un test qui compile et qui contient toutes les instructions binaires qui pourront être analysées. L'exemple 5.4 illustre comment sont générés nos fichiers de tests.

Une première partie initie la fabrication du fichier. Elle rédige dans le fichier de test les appels d'inclusions de fonctions de la bibliothèque standard C, l'invocation de la bibliothèque HACL* au travers du fichier d'en-tête de référence (ici `Hacl_EC_K256`) et une signature de fabrication en commentaire. L'utilisation de la bibliothèque standard permet d'utiliser la fonction `exit`. Avec cet appel, nous construisons nos scripts Binsec avec une interruption sur cette fonction. Cet arrêt précoce permet d'accélérer l'analyse du binaire de la cible (ici `Hacl_EC_K256_felem_sqr`) et nous garantit que cette analyse soit complète.

2. Interface de Programmation d'Application : détermine les définitions et comportement entre les différentes fonctions présentent dans une bibliothèque.

FIGURE 5.4 – 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];
```

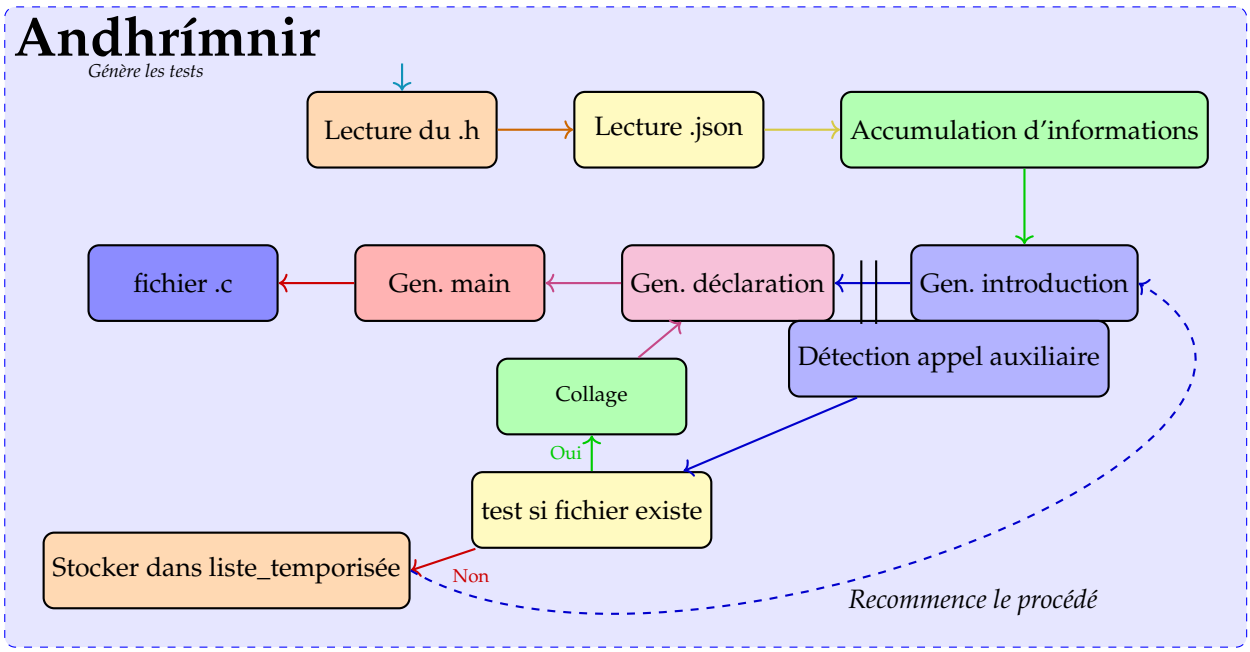
Phase déclarative

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

Phase principale

La deuxième partie contient tous les éléments déclaratifs nécessaires à l'invocation de la fonction. Puis se 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 standardisée entre les fichiers et permet de mettre en place quelques optimisations dans la rédactions des scripts d'analyse Binsec. Cela nous permet aussi de nous repérer plus facilement lorsque nous effectuons du désassemblage de binaire à la main.

FIGURE 5.5 – Représentation schématique du fonctionnement d'Andhrímnir



Comme illustré par la figure 5.5, 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" est ensuite réalisée pour avoir toutes les informations nécessaires à la constitution du fichier test. Une fois ces étapes réalisées, Andhrímnir commence sa préparation. Or il se trouve que parfois, certaines fonctions de HACL* font appel à des structures propres à la bibliothèque 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 5.4 : génération des déclarations puis génération du `main`.

À l'inverse, lorsqu'un appel est détecté, il est possible que la fonction soit déclarée 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 est de temporiser le problème. Andhrímnir prépare des tests pour toutes les fonctions. Donc s'il a besoin d'une fonction qu'il a déjà préparé, nous pouvons accéder aux informations contenues dans le fichier de test associé. Au contraire, 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 fois qu'il a fini son premier passage sur toutes les fonctions d'HACL*. Ce procédé est récursif pour pallier le problème d'appels en cascade.

En réalité Andhrímnir ne recharge pas les informations d'une fonction dont il a besoin, il effectue 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 de données *json* existe afin d'accélérer le développement et la mise en service d'Andhrímnir. Cela permet d'ajouter manuellement des instructions de haut niveau pour la conception des tests. Le code en annexe B.1 illustre ce point : certaines fonctions ont besoin que les paramètres déclarés respectent certaines conditions. Cet exemple est accompagné de son fichier *json* associé B.2 et de son fichier de test B.3.

Emploi et besoin de fichiers de données

L'emploi de fichier *json* est la solution qui nous est apparu lors de l'implémentation d'Érysichthon et d'Andhrímnir. Les deux ont besoin d'avoir des connaissances supplémentaires quand à leurs sources. Lors de la génération d'un test, il fallait déterminer la valeur des paramètres des fonctions appelées pour avoir une compilation optiamle. Lors de la génération des scripts Binsec, il fallait connaître le symbole des paramètres à suivre durant l'analyse symbolique. Dans le premier cas, l'information est caché dans le code *F** de HACL*, dans le second cas, l'information est dans le binaire. Cette solution permet de répondre aux deux besoins.

Make_core_dump et Compilation

Les opérations de production de clichés mémoire et de compilation sont un assemblage de commandes shell et de script pour GDB qui sont concevables sans problèmes. L'élément difficile à cette étape est la compilation de la bibliothèque HACL*. Cette étape est nécessaire pour correctement compiler nos fichiers tests qui appellent HACL*. Or cette gestion de la compilation est réalisée par le projet HACL* lui-même et a besoin d'être améliorée pour permettre une compilation croisée vers d'autres architectures.

Une modification du script de compilation «`configure`» a été proposé et modifié sur le dépôt officiel du projet HACL*.

Make_ini

Ce module consiste à concevoir les fichiers d'instructions pour Binsec. Il doit spécifier les variables secrètes associées à la fonction analysée. À la suite des exemples cités précédemment, le code B.4 illustre comment ces instructions s'organisent. Il est adapté pour l'architecture ciblé et pour le cas x86_64 exploite la mécanique des clichés mémoires.

Un premier temps initie le chargement des données, ensuite l'étiquette «`secret`» est accrochée aux variables à suivre durant l'analyse. Cette phase emploie le fichier *json* de référence du test pour détecter les variables qui ont besoins d'être étiqueter. Des commandes de gestion d'instructions particulières : des appels systèmes, des vérifications de registres

inconnus de Binsec ; permettent que l'analyse ne s'interrompe pas et nous donne un résultat pertinent (`secure`, `insecure`). Enfin nous indiquons notre arrêt d'exploration sur la fonction «`exit`» et nous donnons notre feu vert avec la commande d'exploration totale.

Dans le cadre d'autres architectures, comme ARM, le code 5.1 montre que la différence à considérer est cette affectation manuelle des «`IFUNC`». Pour le moment, la solution en place qui gère une affectation correcte est conçue en fonction du support matériel sur lequel l'outil est activé.

Notre outil est construit, nous allons effectuer nos premiers cas d'usage et analyser les résulats obtenus.

CHAPITRE 6

Résultats et réponses à nos questions de recherche

Nous avons ici un artefact de recherche qui initie la conception d'un outil entièrement automatisé, capable de couvrir plusieurs architectures et plusieurs compilateurs. Cet outil, présenté tout au long de ces chapitres précédents, permet d'obtenir des rapports précis sur la sécurité d'une bibliothèque cryptographique.

6.1 Motivation de la méthode expérimentale

Notre objectif est d'automatiser le processus de vérification d'un binaire d'une fonction d'une bibliothèque cryptographique. Nous savons effectuer ce travail localement, avec un fichier à tester, une compilation effectuée avec plusieurs compilateurs et une analyse Binsec adaptée à chaque version (*e.g.* figures 4.2 et 4.3). Ce mécanisme, permettant une comparaison nette et forte entre les compilateurs est analogue à la méthode du parcours en largeur. Nous testons une variété de compilateurs et les résultats obtenus au fur et à mesure nous permettent d'évaluer une fonction à chaque fois.

La méthode développée intrésèquement avec Érysichthon est analogue au parcours en profondeur. Nous étudions toute la bibliothèque (ou du moins les fonctions présentes dans la liste des cibles autorisés) selon les paramètres ciblés avant de considérer réaliser une analyse vers une autre configuration (architecture / compilateur / optimisation). Cette approche demande une réflexion concernant le choix de la configuration puis sera plus tard ajouté à une liste de configuration à analyser. Les rapports obtenus permettent d'identifier plus rapidement quelles fonctions ont besoin d'être corrigées mais surtout de repérer les éventuels bogues qui peuvent survenir.

Cette méthode nous a permis d'obtenir nos premiers rapports. Nous allons maintenant discuter des résultats récoltés.

6.2 Étude simple sur x86_64

L'exécution s'est réalisée sur une machine équipée d'un processeur *Intel Xeon E5-2620v4* avec 32 Gio de mémoire. Le temps nécessaire pour une analyse complète en x86_64 avec `GCC 12.02` avec les options de compilations courantes est de 4h07 en moyenne. Une analyse complète comprend la compilation de HACL*, la génération des fichiers de tests, la compilation desdits fichiers et l'analyse individuelle de chacun par Binsec. La génération des rapports se fait en fin d'analyse de Binsec avant qu'une nouvelle compilation de HACL* ne soit réalisée. Le rapport consiste à agréger les rapports d'analyse Binsec, compter les événements que l'on cible et dresser des tableaux.

La configuration de notre première étude est donc : [x86_64, `GCC 12.02`, `-O0/1/2/3/s/z`].

Les données détaillées peuvent être consultées en annexe B.1. En l'état, l'analyse rapporte entre 139 et 168 fichiers (en fonction des options de compilation, voir la figure 6.1)

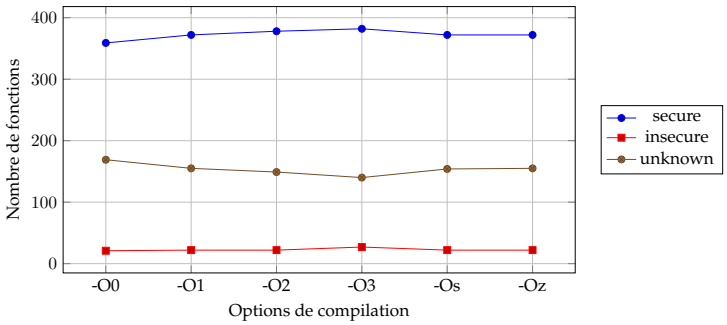


FIGURE 6.1 – Graphes des résultats d'Érysichthon en x86_64

dont l'analyse n'a pas pu se terminer. Il faudrait observer plus en détails ces fichiers pour connaître les causes précises de ces arrêts. Nous pouvons faire dans un premier temps une analyse haut niveau des raisons probables de ces interruptions.

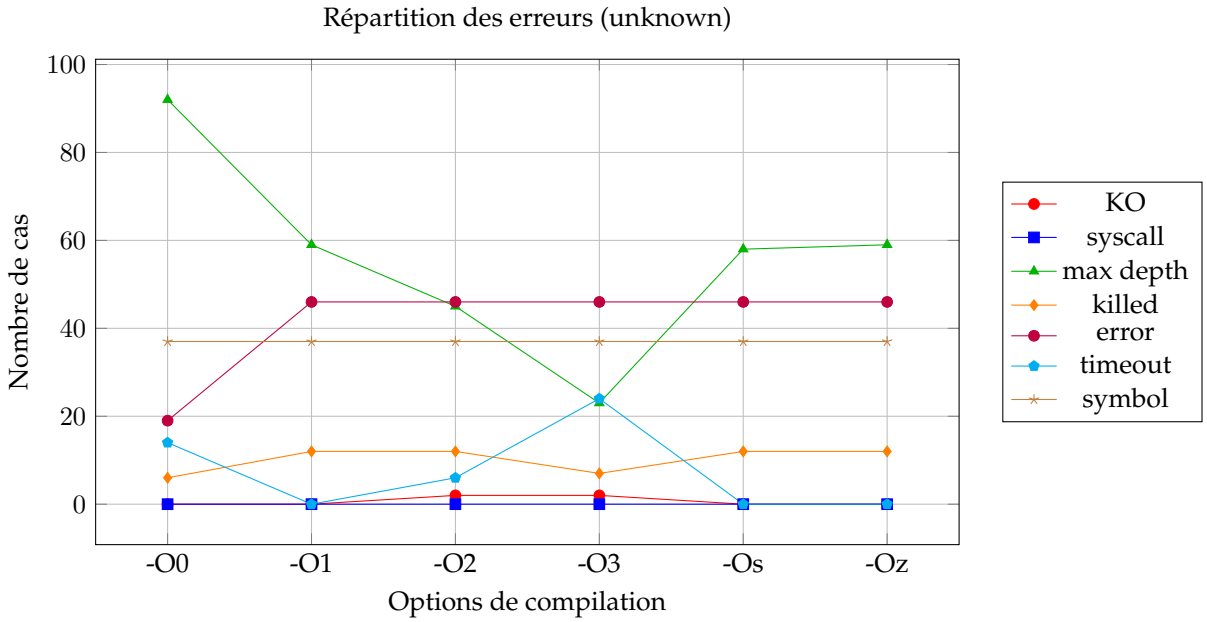


FIGURE 6.2 – Graphes détaillant les erreurs interrompant l'analyse Binsec

Le détail des valeurs est rapporté en annexe B.2.

On peut voir avec cette figure 6.2 que les erreurs sont dues à :

`max depth` Arrêt par limitations du nombre d'instruction à analyser, cela permet de réduire la profondeur des branchements conditionnels à explorer et de limiter le risque de parcours infini.

`timeout` Comme le précédent, limitation par le temps.

`killed` Consommation excessive des ressources, processus interrompu.

`error` Instruction inconnue de Binsec, il a besoin que le script d'instruction soit corrigé.

`symbol` Comme le précédent, mais peut-être que le fichier de test a besoin d'être modifié.

`KO` Instruction inconnue de Binsec, il a besoin d'être amélioré.

Chaque erreur préconise un correctif à apporter à Érysichthon. Nous allons détailler les solutions qui nous sont apparus.

Correctifs à implémenter

Pour résoudre la limitation `max depth`, nous utiliserons l'outil `perf`. Cela nous permet de déterminer le nombre d'instructions que contient le binaire. Identifier cette variable nous permet d'exécuter la commande 4.2 précisément.

Concernant l'erreur de type `timeout`, par défaut une exécution Binsec ne s'interrompt pas, nous avions ajouté ce garde-fou pour forcer l'arrêt de l'analyse de certaines fonctions qui s'étendaient dans le temps. Nos scripts Binsec peuvent être affiné en ciblant précisément les fonctions qui ont besoin de cette limitation. Actuellement, il est sûrement préférable de retirer cette option.

Cela nous amène à l'erreur `killed`. Ce message n'est pas une erreur provoqué par Binsec mais par un moniteur externe qui vérifie que l'exécution d'Érysichthon ne consomme pas toute la mémoire vive de la machine hôte. Comme présenté au chapitre 4, l'analyse d'un fichier binaire évolue exponentiellement. Il se trouve que ces fonctions sont comme un sandwich de plus petites fonctions et font exploser la complexité des formules SMT. L'erreur `max depth` n'est pas déclenché donc il faut sûrement affiner le script Binsec pour ces fonctions.

L'erreur `symbol` indique une erreur présente dans le scrit Binsec. Cela signifie que le fichier binaire n'a pas le symbole observé par Binsec. Cette erreur est due à un fichier de test produit par Andhrímnir où deux fonctions utilisent des paramètres nommés identiquement. Cette erreur se corrige donc en améliorant le module Andhrímnir.

Pour finir, `KO` et `error` sont des erreurs provoqués par Binsec. Dans le premier cas l'instruction binaire lui ait inconnu et dans le second cas il effectue une analyse vers un segment du binaire incompréhensible (fin d'une instruction et début d'une autre ou même une zone sans instruction), il faut donc modifier le script Binsec pour prévenir ce cas de figure.

Sécurité de HACL*

Revenons sur les résultats présentés sur la figure 6.1 et ignorons les fonctions marquées `unknown`. Les fichiers non sécurisés sont les plus nombreux avec l'option `-O3` (27) et le moins avec l'option `-O0` (21). Nous retrouvons le détail des résultats en annexe B.3. Cette expérimentation confirme les travaux de SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN, dans le sens où augmenter le niveau d'optimisation de compilation augmente le nombre de fonctions qui sont non sécurisées. Nos résultats pourront s'aligner avec l'ajout du compilateur `LLVM`.

En revanche, si la question de la sécurité des fonctions identifiées non sécurisées se pose, actuellement la réponse est non. Comme nous analysons toute la bibliothèque HACL*, il est normal que certaines fonctions ne soient pas sécurisées. Elles n'ont pas pour objectif de l'être. Ce sont des fonctions comme `Hacl_P256_validate_public_key` ou `Hacl_P256_ecdsa_verif_p256_sha384` qui effectuent des vérifications, des conversions, sur des données publiques.

Actuellement, dans la liste, aucune fonction indiquée non sécurisée ne demande une réimplémentation et peut être conservée dans la bibliothèque. En première conclusion préliminaire, nous pouvons affirmer qu'il est préférable de ne pas utiliser l'option de compilation `-O3`.

Ces premiers résultats sont encourageants quant à la méthode employée et laissent de la place à de nombreuses améliorations futures. Une application concrète et exploitable pour l'amélioration de HACL a été développée et nous permettra d'avoir des garanties formelles sur les binaires utilisables par des utilisateurs de la bibliothèque. Nous allons discuter des actions que nous pouvons réaliser pour perfectionner le travail déjà accompli.*

Discussion et Ouverture

Discussion

Les résulats obtenues grâce à l'analyse d'Érysichthon ne permettent pas encore de conclure sur la sécurité globale de la bibliothèque Hacl*, il reste encore trop de fonctions non analysées ($\sim 39\%$). En revanche les premiers résultats sont encourageants et montre que l'utilisation des contre-mesures développé au chapitre 2 est effectivement une bonne pratique. Compléter Érysichthon pour avoir une analyse complète est en tête de liste de la liste des tâche du projet.

Actuellement, un seul compilateur a permis de produire ces résultats, il faut absolument étendre l'utilisation à d'autres compilateurs pour pouvoir croiser les résultats et avoir une étude plus complète quant à la sécurité de cette bibliothèque.

Retour sur les résultats

Nous savons que les optimisations de compilateur modifient le binaire et peuvent insérer des fuites parce que les modifications changent la structure du programme. Avec cette outil, au lieu d'appeler frontalement `-O2`, nous pouvons plutôt appeler nominativement les options qui se cachent derrière : `-falign-functions`, `-falign-jumps`, *etc*. Cette solution permettrait d'identifier les passes de compilateurs qui sont déterminantes pour l'apparition de failles. Il faut en revanche être attentif avec cette solution car `GCC`,utilisé ici, fonctionne différemment de `LLVM+Clang`. Ce dernier emploie une représentation interne pour effectuer la transformation entre un programme source et un programme binaire.Cette représentation interne évolue entre les différentes étapes de compilations. Cela induit que certaines étapes, même si elle ne nous plaisent pas car elles ajoutent des failles, sont nécessaires pour les suivantes. Nous avons notamment pu croiser ce cas de figures durant notre période de test et à moins d'avoir un code adapté à chaque compilateur, la solution retenu fu de désactiver les optimisations de compilateurs pour le bloc de code concerné.

En poursuivant dans cette voie, il nous est aussi possible d'identifier précisément les fonctions qui ont besoins d'être sécurisée et d'adapter leur compilation. Nous pourrions avoir une liste `sécurisées` qui identifie les fonctions dont la compilation induit trop de dégâts et les autres seraient compilées avec niveau d'optimisation plus élevé. Par exemple, si *A.c* est compilé avec `-O0`, *A.o* sera généré sans optimisation. De même, si *B.c* est compilé `-O3`, *B.o* contiendra des fonctions optimisées. Ainsi, dans le binaire final *C.o*, les appels vers une fonction de *A.o* sont des instructions de saut vers le code compilé avec `-O0`, et les appels vers *B.o* des sauts vers du code compilé avec `-O3`. Nous obtenons un mélange de fonctions optimisées et non optimisées dans le même exécutable. Cette solution réduira les performances globales et mais préservera un niveau de sécurité plus élevé. Cette solution pave la voie vers une nouvelle étude.

Implémentation dans le cadre d'une CI

Utiliser Érysichthon c'est retrouver des vulnérabilités documentés, il nous a aussi permis d'identifier une vingtaine de fonctions présentant des fuites (même si la sécurité n'est pas engagée). Donc ajouter Érysichthon à une chaîne de tests, *CI* ou Intégration Continue en langue de molière, est l'objectif principal que nous nous fixons. SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN, notre référence, énumèrais une liste de solution permettant de garantir la sécurité temps constant d'une librairie. Les solutions notables étaient de mettre en place

un "*distributeur*" de binaire, permettant au développeur de piocher selon ses besoins avec bien entendu tous les codes sources publiques, ou d'effectuer massivement des tests. La voie prise pour assurer la sécurité d'HACL* s'aligne avec cette dernière solution. Et ajouter cet outil à un mécanisme d'intégration continue permettra d'avoir des rapports réguliers dès la publication d'une nouvelle version de HACL* (lors d'ajout de nouvelles primitives cryptographiques ou de modification du code source).

L'avantage indéniable de cette méthode est que les failles seront découverte au fur et à mesure des tests effectués sur des compilateurs de plus en plus récents, limitant le risque d'avoir une faille se révélant dans la nature.

Ouverture

La sécurité des binaires face aux attaques temporelles n'est pas chose aisé. Nous nous sommes concentrés avec ce projet sur la vérification de binaire et la conception d'une certification de sécurité vers un compilateur. Nous considérons le compilateur comme une boîte noire. Cette approche se justifie par la non main mise sur le développement de tels projets. Historiquement les ordinateurs étaient lents et la conception du logiciel devait être réalisé avec précision pour optimiser ou guider la conception des binaires. Les compilateurs et leurs optimisations ont permis d'améliorer globalement la production de binaire et réduire les coûts de compilation. Aujourd'hui les ordinateurs ont des composants très rapide. La lecture des instructions est plus rapide que l'exécution desdites instructions.

Prompt état de l'art des processeurs

Le rapport de PORNIN [Por25] décrit très bien les mécanismes employé par les processeurs pour accélérer l'exécution de programmes. Une modélisation d'un processus peut être présenté ainsi :

1. Charge l'instruction pointé par le compteur de programme, incrémente ce dernier.
2. Décode l'instruction.
3. Exécute l'instruction.

La dernière étape peut induire une modification de la mémoire, des chargements ou modifications de valeurs de registre ou résoudre des calculs. Tandis que chaque étape peut se résoudre selon un même rythme, cette troisième opération, à cause de sa variance, peut entraîner des ralentissements. Les constructeurs de matériels ont donc développé des techniques accélérer le temps total d'exécution et permettre ainsi des gains de performances.

Voici les principales techniques employées :

Pipeline Permet l'exécution parallèle de différentes instructions. Les instructions se résolvent toujours dans l'ordre abstrait, mais l'exécution de la suivante commence alors que la précédente (et éventuellement d'autres encore en cours) n'est pas terminée.

Prédiction de branche Pré-chargement de branchement. Grâce à la méthode précédente, le processeur peut avancer dans son préchargement d'instructions en explorant les deux côtés d'un branchement mémoire puis défausser la branche inutile.

Renommage de registres Modification des registres employés pour éviter les dépendances. Grâce au Pipeline, il peut y avoir des ralentissements dans la prédiction à cause de valeurs dépendantes entre les instructions. Cette méthode emploie un autre registre temporairement pour simuler le comportement attendu.

Micro-opérations Utilisation d'un encodge interne pour découper les instructions binaires en unité plus petites. Cette opération invisible pour le développeur permet d'optimiser le binaire (*e.g.* combiner une comparaison et un branchement en une seule opération).

Exécution désordonné[1] Certaines opération demande malgré tout plus de temps d'exécution, grâce à sa capacité à lire et traduire en vance les opérations à réaliser, le processeur peut réordonner la file d'exécution.

Ces résultats impliquent une rupture de la politique temps constant et sont invisible aux yeux du développeur. Une connaissance approfondi sur le fonctionnement des processeurs est nécessaires pour avoir connaissance de ces éléments et des possiblitié de les contre-carrés. Intel et ARM ont mis à disposition des variables à spécifier pour permettre

la désactivation de tels optimisations pour certaines opérations (multiplication et division d'entier). Ces solutions partielles ne permettent pas d'avoir des garanties sur l'ensemble du binaire que l'on exécute.

Travaux de recherche au niveau du processeur

Les architectures portées par Intel et ARM sont propriétaires et donc nous offre peu de possiblités d'évolution. RISC-V de l'autre côté, grâce à son enregistrement dans l'espace commun peut acceuillir des travaux d'universitaires. Cela a permis à des équipes Inria de développer des mécanismes au niveau du processeur qui permet de garantir une exécution sans fuite par canal auxiliaire. Les travaux de GAUDIN, HATCHIKIAN-HOUDOT, BESSON, COTRET, GOGNIAT, HIET, LAPOTRE et WILKE [Gau+23] proposent l'ajout d'instructions *lock* et *unlock* dans l'ISA de l'architectures pour encapsuler des opérations qui doivent être réalisées avec des accès au cache mémoire. Cette protection vient avec l'ajout d'une nouvelle structure microarchitecturale à ajouter sur une carte mère.

Plus récemment, LAMMERS, MÜLLER, DHOOGHE et MORADI [Lam+25] propose un circuit logique qui réalise une opération de masquage et permet de conserver les propriétés temps constant.

Conclusion

À travers ce mémoire nous avons vus la mise au point d'Érysichthon, un outil d'analyse automatique de bibliothèque cryptographique. Cette conception a demandé l'implémentation d'un ensemble de module et notamment d'Andhrímnir, un outil de génération de tests. Ce module permet d'avoir au minimum un test par fonctions présentes dans la bibliothèque analysé. Érysichthon est un outil utilisant Binsec, notamment les extensions permettant la vérification formelles des fuites temporelles par l'analyse symbolique non-interférante.

Cette conception c'est réalisé grâce l'étalonnage des réponses à nos questions de recherche. Les travaux de SCHNEIDER, LAIN, PUDDU, DUTLY et CAPKUN ont mis en lumière (**QR1**) que la propagation des garanties de sécurité n'est pas nécessairement conservé lors de la compilation. Vouloir conserver cette propagation demande d'utiliser un compilateur spécialisé et de rajouter des spécifications dans le code source pour préserver le niveau de sécurité attendu. Nous avons aussi pu voir les contraintes inhérente à cette solution, ce qui nous a décidé à explorer d'autres solutions. Celle retenue nous a guidé vers l'étude et l'analyse de binaire après compilation. Cette solution consiste à de vérifier si les éléments de sécurité ont été préservés ou si des fuites sont apparues. Le choix judicieux (**QR2**) d'utiliser Binsec nous a permis de mettre en place une méthodologie et l'automatisation de l'analyse de binaire à travers un éventail d'architectures et de compilateur. Avec ces briques et de nouveaux protocoles, nous avons pu dresser un cahier des charges (**QR3**) pour effectuer un changement d'échelle et réussir à concevoir un outil effectuant la vérification de bibliothèque cryptographique.

La première bibliothèque cryptographique vérifié par Érysichthon est HACL*. Cette bibliothèque a la particularité d'être formellement vérifiée et présente des garanties de sécurité à propos du code source de ses fonctions. L'étude [Sch+24] avait mis en lumière des fuites en fonction des compilateurs employés. Nous avons avons reproduis les analyses réalisée et cela nous a permis d'identifier des bogues présent dans Binsec.

Cette étape nous a permis de concevoir des méthodes pour automatiser les analyses manuelles que nous venions de réaliser. La production de protocoles précis et détaillés nous a permis d'accélérer le processus d'implémentation d'Érysichthon et d'effectuer notre première analyse complète de HACL* sur x86_64 avec `GCC 12.02`.

Discuter des résulats obtenus nous a permis d'établir une liste de correctifs nécessaire pour avoir une analyse complète d'une bibliothèque cryptographique. Nous avons pu, avec cette analyse de HACL*, concevoir une pré-certification de la sécurité de cette bibliothèque et attester d'une sécurité réellement traduite au niveau machine.

Remerciements

Ces remerciements sont avant tout dirigés à Aymeric FROMHERZ et Yanis SELLAMI, mes tuteurs de stage. Ils ont permis la réalisation de ce projet en m'offrant l'opportunité d'évoluer entre les bureaux d'Inria Paris et ceux du CEA. J'ai pus rencontré leurs collègues et évoluer avec eux. Je présente donc mes remerciements à mes collègues de Paris pour ces chouettes moments. Je souhaite décerner une mention spéciale à Hadrien, qui a passé son été quasiment en tête à tête avec moi pendant que les autres se reposaient loin de la capitale. Une autre mention spéciale est attribué à Fernando, partenaire de bureau, avec qui la musique fut célébré et joué chaque jour. Concernant les chercheurs du CEA, mes remerciements vont à Sébastien BARDIN qui supervisait ce stage sous l'amas de travail qu'il devait abattre et m'encourageait sans cesse, ainsi qu'à Frédérique RECOULES, dont l'expertise et la pédagogie mon permis de poursuivre plus en avant dans la conception d'Érysichthon.

Merci à l'équipe pédagogique du master CSI, pour m'avoir permis d'écrire ces lignes et d'aposer un point final à ce mémoire.

Merci aux copains pour les rires et les moments de détente, précieux pour nous faire relacher la pression.

Mes derniers remerciements vont à mes proches et à ma chère partenaire.

Bibliographie

[Avi71] Faulty-Tolerant Computing : An Overview, *A.* AVIZIENIS, *1971*.

[Koc96] Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems, *Paul C.* KOCHER, *1996*.

[AHa98] OpenSSL, *Eric* ANDREW YOUNG, *Tim* HUDSON *et OpenSSL* AUTHORS, *1998*, URL : `https://www.openssl.org/`.

[KJJ99] Differential Power Analysis, *Paul* KOCHER, *Joshua* JAFFE *et Benjamin* JUN, *1999*.

[Bar+04] The Sorcerer's Apprentice Guide to Fault Attacks, *Hagai* BAR-EL, *Hamid* CHOUKRI, *David* NACCACHE, *Michael* TUNSTALL *et Claire* WHELAN, *2004*, URL : `https://eprint.iacr.org/2004/100`.

[AKS06] Predicting Secret Keys Via Branch Prediction, *Onur* ACIIÇMEZ, *Çetin Kaya* KOÇ *et Jean-Pierre* SEIFERT, *2006*.

[Lan10] Adam LANGLEY. *ctgrind : Checking that functions are constant time with Valgrind*. Rapp. tech. 2010. URL : `https://github.com/agl/ctgrind`.

[KMO12] Automatic quantification of cache side-channels, *Boris* KÖPF, *Laurent* MAUBORGNE *et Martín* OCHOA, *2012*.

[Alm+13] Formal Verification of Side-Channel Countermeasures Using Self-Composition, *José Bacelar* ALMEIDA, *Manuel* BARBOSA, *João Sousa* PINTO *et Benjamin* VIEIRA, *2013*.

[Doy+13] CacheAudit : A Tool for the Static Analysis of Cache Side Channels, *Boris* DOYCHEV, *Daniel* FELD, *Benjamin* KÖPF, *Laurent* MAUBORGNE *et Jan* REINEKE, *2013*.

[Bar+14] System-level non-interference for constant-time cryptography, *Gilles* BARTHE, *Gustavo* BETARTE, *Juan Diego* CAMPO, *Carlos* LUNA *et David* PICHARDIE, *2014*.

[Can14] Programmation en langage C, *Anne* CANTEAUT, *2014*.

[Liu+15] Last-Level Cache Side-Channel Attacks are Practical , *Fangfei* LIU, *Yuval* YAROM, *Qian* GE, *Gernot* HEISER *et Ruby B.* LEE, *2015*, URL : `https://yuval.yarom.org/pdfs/LiuYGHL15.pdf`.

[Mas+15] Thermal Covert Channels on Multi-core Platforms, *Ramya Jayaram* MASTI, *Devendra* RAI, *Aanjhan* RANGANATHAN, *Christian* MÜLLER, *Lothar* THIELE *et Srdjan* CAPKUN, *2015*, URL : `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/masti`.

[RLT15] Raccoon : Closing Digital Side-Channels through Obfuscated Execution, *Ashay* RANE, *Calvin* LIN *et Mohit* TIWARI, *2015*, URL : `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane`.

[Bar+16] Computer-Aided Verification for Mechanism Design, *Gilles* BARTHE, *Marco* GABOARDI, *Eduardo J.* ARIAS, *Justin* HSU, *Aaron* ROTH *et Pierre-Yves* STRUB, *2016*.

[Pes+16] DRAMA : Exploiting DRAM Addressing for Cross-CPU Attacks, *Peter* PESSL, *Daniel* GRUSS, *Clémentine* MAURICE, *Michael* SCHWARZ *et Stefan* MANGARD, *2016*, URL : `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl`.

[Por16] BearSSL :A constant time cryptographic library, *Thomas* PORNIN, *2016*, URL : `https://www.bearssl.org/`.

[RPA16] Sparse representation of implicit flows with applications to side-channel detection, *Bernardo* RODRIGUES, *Francisco M. Q.* PEREIRA *et Diego F.* ARANHA, *2016*.

[YGH16] CacheBleed : A Timing Attack on OpenSSL Constant Time RSA, *Yuval* YAROM, *Daniel* GENKIN *et Nadia* HENINGER, *2016*, URL : *https://eprint.iacr.org/2016/224*.

[Ant+17] Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels, *Thomas* ANTONOPOULOS, *Pietro* GAZZILLO, *Michael* HICKS, *Eric* KOSKINEN, *Tachio* TERAUCHI *et Shiying* WEI, *2017*.

[BPT17] Verifying Constant-Time Implementations by Abstract Interpretation, *Sandrine* BLAZY, *David* PICHARDIE *et André* TRIEU, *2017*.

[CFD17] Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic, *Jie* CHEN, *Yu* FENG *et Isil* DILLIG, *2017*.

[Mog+17] MemJam : A False Dependency Attack Against Constant-Time Crypto Implementations, *Ahmad* MOGHIMI, *Jan* WICHELMANN, *Thomas* EISENBARTH *et Berk* SUNAR, *2017*, URL : *http://dx.doi.org/10.1007/s10766-018-0611-9*.

[RBV17] Dude, is my code constant time ?, *Oscar* REPARAZ, *Josep* BALASCH *et Ingrid* VERBAUWHEDE, *2017*.

[Tea17] MemorySanitizer, *LLVM* TEAM, *2017*.

[Wan+17] Cached : Identifying Cache-Based Timing Channels in Production Software, *Shuai* WANG, *Pei* WANG, *XiaoFeng* LIU, *Dinghao* ZHANG *et Dongyan* WU, *2017*.

[Zin+17] HACL* : A verified modern cryptographic library, *Jean-Karim* ZINZINDOHOUÉ, *Karthikeyan* BHARGAVAN, *Jonathan* PROTZENKO *et Benjamin* BEURDOUCHE, *2017*, URL : *https://hacl-star.github.io/*.

[Ath+18] Sidetrail : Verifying time-balancing of cryptosystems, *Konstantinos* ATHANASIOU, *Brian* COOK, *Michael* EMMI, *Colm* MACCÁRTHAIGH, *Daniel* SCHWARTZ-NARBONNE *et Serdar* TASIRAN, *2018*.

[BGL18] Secure Compilation of Side-Channel Countermeasures : The Case of Cryptographic "Constant-Tim *Gilles* BARTHE, *Benjamin* GRÉGOIRE *et Vincent* LAPORTE, *2018*.

[Bre+18] Symbolic Path Cost Analysis for Side-Channel Detection, *Thomas* BRENNAN, *Subarna* SAHA, *Tevfik* BULTAN *et Corina S.* PASAREANU, *2018*.

[Nei18] Timecop, *Moritz* NEIKES, *2018*.

[VPS18] Nemesis : Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic, *Jo* VAN BULCK, *Frank* PIESSENS *et Raoul* STRACKX, *2018*.

[Wei+18] DATA - Differential Address Trace Analysis : Finding Address-Based Side-Channels in Binaries, *Samuel* WEISER, *Andreas* ZANKL, *Raphael* SPREITZER, *Kasper* MILLER, *Stefan* MANGARD *et Georg* SIGL, *2018*.

[Wic+18] Microwalk : A framework for finding side channels in binaries, *Jan* WICHELMANN, *Ahmad* MOGHIMI, *Thomas* EISENBARTH *et Berk* SUNAR, *2018*.

[Wu+18] Eliminating timing side-channel leaks using program repair, *Mingjie* WU, *Shouling* GUO, *Patrick* SCHAUMONT *et Chao* WANG, *2018*.

[Bar+19] Formal Verification of a Constant-Time Preserving C Compiler, *Gilles* BARTHE, *Sandrine* BLAZY, *Benjamin* GRÉGOIRE, *Rémi* HUTIN, *Vincent* LAPORTE, *David* PICHARDIE *et Alix* TRIEU, *2019*, URL : *https://eprint.iacr.org/2019/926*.

[Cau+19] FaCT : A DSL for timing-sensitive computation, *Srinath* CAULIGI, *Craig* SOELLER, *Brian* JOHANNESMEYER, *Fraser* BROWN, *Riad S.* WAHBY, *Jan* RENNER, *Benjamin* GRÉGOIRE, *Gilles* BARTHE, *Ranjit* JHALA *et Deian* STEFAN, *2019*.

[DBR19] Binsec/Rel : Efficient Relational Symbolic Execution for Constant-Time at Binary-Level, *Lesly-Ann* DANIEL, *Sébastien* BARDIN *et Tamara* REZK, *2019*, URL : *http://arxiv.org/abs/1912.08788*.

[Wat+19] Ct-wasm : Type-driven secure cryptography for the web ecosystem, *Connor* WATT, *Jan* RENNER, *Nicholas* POPESCU, *Srinath* CAULIGI *et Deian* STEFAN, *2019*.

[ANS20] Règles de programmation pour le développement sécurisé de logiciels en langage C, ANSSI, *2020*.

[Dis20] haybale-pitchfork, *Craig* DISSELKOEN, *2020*.

[HEC20] ct-fuzz : Fuzzing for Timing Leaks, *Sizhuo* HE, *Michael* EMMI *et Gabriel F.* CIOCARLIE, *2020*.

[Pol+20] HACLxN : Verified generic SIMD crypto (for all your favourite platforms), *Marina* POLUBELOVA, *Karthikeyan* BHARGAVAN, *Jonathan* PROTZENKO, *Benjamin* BEURDOUCHE, *Aymeric* FROMHERZ, *Natalia* KULATOVA *et Santiago* ZANELLA-BÉGUELIN, *2020*.

[Wei+20] Big Numbers - Big Troubles : Systematically Analyzing Nonce Leakage in (EC)DSA Implementations, *Samuel* WEISER, *David* SCHRAMMEL, *Lukas* BODNER *et Raphael* SPREITZER, *2020*.

[Bor+21] Constantine : Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization, *Pietro* BORRELLO, *Daniele Cono* D'ELIA, *Leonardo* QUERZONI *et Cristiano* GIUFFRIDA, *2021*, URL : `http://dx.doi.org/10.1145/3460120.3484583`.

[Jan+21] "They're not that hard to mitigate" : What Cryptographic Library Developers Think About Timing Attac *Jan* JANCAR, *Marcel* FOURNÉ, *Daniel De Almeida* BRAGA, *Mohamed* SABT, *Peter* SCHWABE, *Gilles* BARTHE, *Pierre-Alain* FOUQUE *et Yasemin* ACAR, *2021*, URL : `https://eprint.iacr.org/2021/1650`.

[Mei+21] Constant-Time Arithmetic for Safer Cryptography, *Lúcás Críostóir* MEIER, *Simone* COLOMBO, *Marin* THIERCELIN *et Bryan* FORD, *2021*, URL : `https://eprint.iacr.org/2021/1121`.

[Gau+23] Work in Progress : Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protecti *Nicolas* GAUDIN, *Jean-Loup* HATCHIKIAN-HOUDOT, *Frédéric* BESSON, *Pascal* COTRET, *Guy* GOGNIAT, *Guillaume* HIET, *Vianney* LAPOTRE *et Pierre* WILKE, *2023*.

[Gei+23] A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptograph *Antoine* GEIMER, *Mathéo* VERGNOLLE, *Frédéric* RECOULES, *Lesly-Ann* DANIEL, *Sébastien* BARDIN *et Clémentine* MAURICE, *2023*, URL : `https://doi.org/10.1145/3576915.3623112`.

[Sch+24] Breaking Bad : How Compilers Break Constant-Time Implementations, *Moritz* SCHNEIDER, *Daniele* LAIN, *Ivan* PUDDU, *Nicolas* DUTLY *et Srdjan* CAPKUN, *2024*, URL : `https://arxiv.org/abs/2410.13489`.

[Lam+25] Constant-Cycle Hardware Private Circuits, *Daniel* LAMMERS, *Nicolai* MÜLLER, *Siemen* DHOOGHE *et Amir* MORADI, *2025*, URL : `https://eprint.iacr.org/2025/1331`.

[Por25] Constant-Time Code : The Pessimist Case, *Thomas* PORNIN, *2025*, URL : `https://eprint.iacr.org/2025/435`.

[Sch+25] Developers : Beware of Timing Side-Channels, *Dominik* SCHNEIDER, *Jannik* ZEITSCHNER, *Michael* KLOOS, *Kerstin* LEMKE-RUST *et Luigi Lo* IACONO, *2025*.

ANNEXE A

Références

TABLE A.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

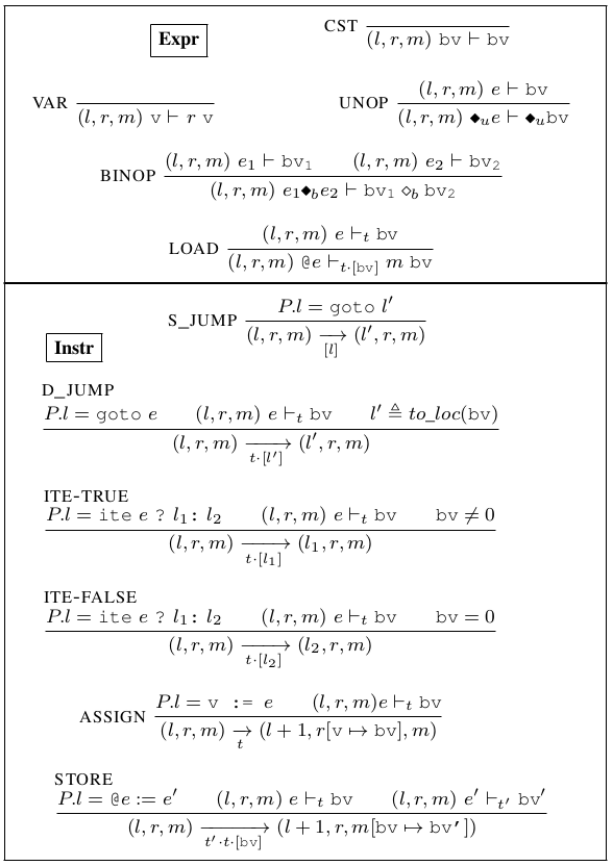


FIGURE A.1 – Ensemble d'instructions défini formellement par [DBR19]

L'ensemble de ces règles a permis de décrire formellement le comportement de l'extension `checkct`, qui permet de vérifier les propriétés temps constants d'un programme.

ANNEXE B

Érysichthon, structure, exemples et résultats

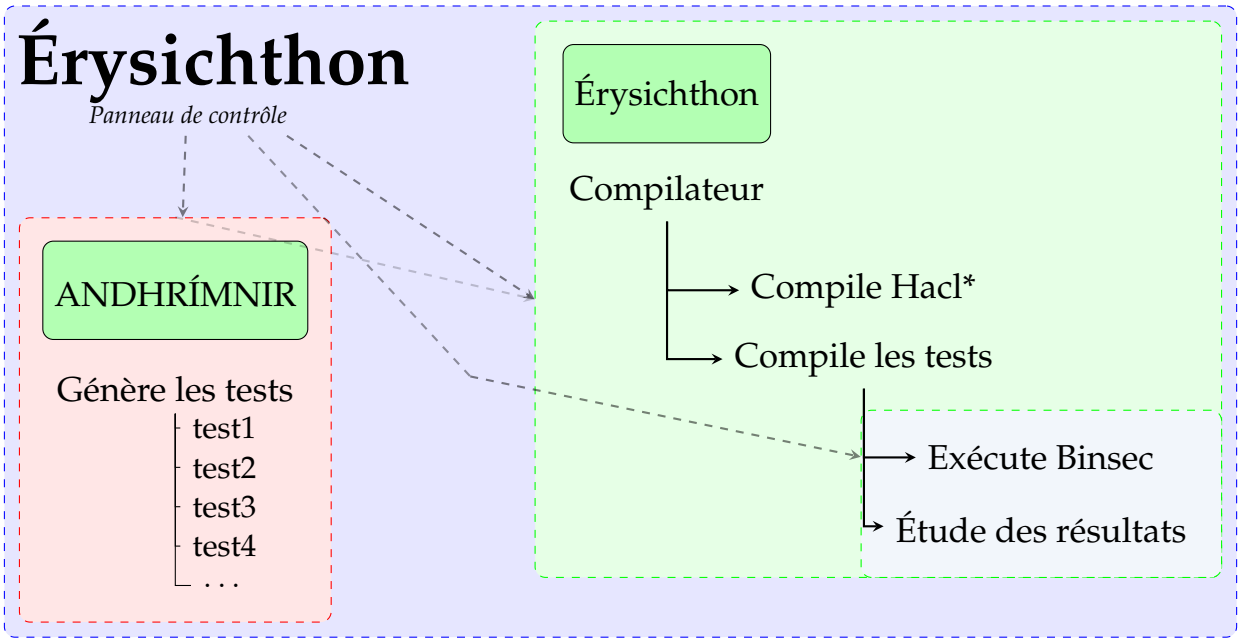


FIGURE B.1 – Structure d'Érysichthon, schéma du point de vue de l'usager
Les flèches grises indiquent tous les éléments actionnables individuellement.

```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 B.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 B.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 B.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 B.4 – Instruction Binsec générée automatiquement, Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.ini

Optimisation	Secure	Unknown	Insecure
-O0	359	168	21
-O1	372	154	22
-O2	378	148	22
-O3	382	139	27
-Os	372	154	22
-Oz	373	153	22

TABLE B.1 – Résultats d'Érysichthon en x86_64

Erreur / Option	-O0	-O1	-O2	-O3	-Os	-Oz
KO	0	0	2	2	0	0
syscall	0	0	0	0	0	0
max depth	92	59	45	23	58	59
killed	6	12	12	7	12	12
error	19	46	46	46	46	46
timeout	14	0	6	24	0	0
symbole	37	37	37	37	37	37

TABLE B.2 – Tableau détaillant les erreurs interrompant l'analyse Binsec

Fonctions	Options concernées					
Hacl_EC_Ed25519_point_eq	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_ecdh	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_ecdsa_verify_hashed_msg	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_ecdsa_verify_sha256	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_is_public_key_valid	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_public_key_compressed_from_raw	O0					
Hacl_K256_ECDSA_public_key_uncompressed_to_raw	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_is_signature_normalized	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_signature_normalize	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_verify_hashed_msg	O0	O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_verify_sha256	O0	O1	O2	O3	Os	Oz
Hacl_NaCl_crypto_box_open_detached_afternm	O0	O1	O2	O3	Os	Oz
Hacl_NaCl_crypto_secretbox_open_detached	O0	O1	O2	O3	Os	Oz
Hacl_P256_compressed_to_raw	O0					
Hacl_P256_dh_responder	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_p256_sha2	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_p256_sha384	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_p256_sha512	O0	O1	O2	O3	Os	Oz
Hacl_P256_ecdsa_verif_without_hash	O0	O1	O2	O3	Os	Oz
Hacl_P256_uncompressed_to_raw	O0	O1	O2	O3	Os	Oz
Hacl_P256_validate_public_key	O0	O1	O2	O3	Os	Oz
Hacl_FFDHE_ffdhe_shared_secret_precomp		O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_sign_hashed_msg		O1	O2	O3	Os	Oz
Hacl_K256_ECDSA_secp256k1_ecdsa_sign_sha256		O1	O2	O3	Os	Oz
Hacl_NaCl_crypto_box_beforenm				O3		
Hacl_NaCl_crypto_box_detached				O3		
Hacl_NaCl_crypto_box_easy				O3		
Hacl_NaCl_crypto_box_open_detached				O3		
Hacl_NaCl_crypto_box_open_easy				O3		

TABLE B.3 – Détails des fonctions non sécurisées en fonction des optimisations entrées, éxecution d'Érysichthon en x86_64