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 1.1 – 17 août 2025

Déclaration de paternité du document

Je certifie sur l'honneur que ce document que je soumets pour évaluation afin d'obtenir le diplôme de Master en *Sciences et Technologies*, Mention *Mathématiques* ou *Informatique*, Parcours *Cryptologie et Sécurité Informatique*, est entièrement issu de mon propre travail, que j'ai porté une attention raisonnable afin de m'assurer que son contenu est original, et qu'il n'enfreint pas, à ma connaissance, les lois relatives à la propriété intellectuelle, ni ne contient de matériel emprunté à d'autres, du moins pas sans qu'il ne soit clairement identifié et cité au sein de mon document.

Date et Signature

17 août 2025

Art des Spires et plongée Oniriques, *2025*

Résumé

Résumé

Table des matières

Conclusion

Annexes

Table des figures

Table des codes

Liste des tableaux

Introduction

Introduction

Le développement sécurisé est une tâche ardue. Si on porte notre regard vers le langage de programmation C, un guide [Can14] [1] porté par l'INRIA [2] est complet en 133 pages tandis qu'un guide pour du développement sécurisé[ANS20] produit par l'ANSSI [3] comprends 182 pages. Cette comparaison met en évidence la discipline requise par le développeur pour faire de la programmation sécurisée ; en sus des connaissances, pour améliorer son efficacité, en cryptologie, en architecture matérielle et en programmation bas niveau .

Malheureusement, malgré ces compétences, des erreurs peuvent être produites puis exploiter pour réaliser des attaques sur ces sytèmes sécurisés. Il existe de nombreuses classe d'attaques, certaines exploitant les défauts de conception (type A) tandis que d'autres utilisent les caractéristiques matériels (type B). Pour limiter ces effets de bords, la pratique de la programmation formelle permet de contraindre le développeur et empêcher l'apparitions de ces erreurs. La production de preuve formelle du code à l'issu de cet exercice permet d'avoir des garanties contres les attaques de type A.

En revanche, pour se défendre d'attaques de type B (ou attaques par canal auxiliaire) dépendantes du matériel support du programme, il est plus difficile d'avoir une méthode miracle. Actuellement, la solution la plus courante est d'identifier les attaques existantes pour ajouter les contre-mesures adéquates permettant d'avoir un système sécurisé. Une sous-classe d'attaque continue malgré tout de résister à cette méthode : les attaques temporelles.

Découverte par Paul Kocher en 1996 [Koc96], il les décrit comme «*une mesure précise du temps requis par des opérations sur les clées secrètes, permettrait à un attaquant de casser le cryptosystème*». Face à cette menace, l'enjeu d'avoir un code *achrognostique* [4] vient se rajouter aux pratiques de programmations sécurisées. Et pourtant, si contre les attaques de type A on arrive à concevoir des preuves mathématiques de sécurité associées à nos systèmes sécurisés, les garanties contre les attaques de type B sont plus faible ou inexistente.

En 2024, les travaux de SCHNEIDER et al. [Sch+24] prouvent qu'un usage inadapté de compilateur sur un système sécurisé introdui 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) que nous tenterons de répondre à travers ce document.

QR1 Est-il possible de détecter les failles qui permettent une attaque temporelles ?

QR2 Est-il possible d'automatiser la détection de ces failles ?

QR3 Est-il possible d'étendre ce mécanisme entre différentes architectures ?

Les réponses à ces questions permettraient de développer des systèmes sécurisés, communs entre différents supports et d'avoir des garanties de sécurité.

1. Développé par Anne Canteaut, chercheuse de l'équipe COSMIQ, récemment entrée à l'Académie des Sciences
2. Institut National de Recherche en Informatique et Automatisme
3. Agence nationale de la Sécurité des Systèmes d'Information
4. Néologisme de Thomas Pornin dans son article *Constant-Time Code : The Pessimist Case* [Por25] pour désigner un code sans connaissance de temps

Fin d'introduction - à finir

Dans la première section nous reviendrons sur les attaques temporelles, leurs impacts et comment s'en protéger. Puis, Dans la deuxième section nous présenterons les outils disponibles à l'analyse et pour la détection de failles. Nous continuerons, dans la troisième section, avec la présentation de nos contributions. *Enfin, dans la quatrième section nous présenterons les mécanismes présent au plus bas niveau de l'informatique pour se protéger des attaques temporelles.*

Ce travail a été réalisé au sein du centre INRIA de Paris dans le cadre du projet *Everest* concernant la mise au point de Hacl*.

Préambule

HACL*[5]

Acronyme pour "High assurance cryptography library", lire *"HACL star"*. Il s'agit d'une bibliothèque cryptographique développée au sein du **Projet Everest**[6]. Initié en 2016, ce projet porté par des chercheurs de l'INRIA (équipe PROSECCO[7]), du Centre de Recherche Microsoft et de l'Université Carnégie 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 prouvé 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, et bien d'autres *etc*.

Binsec[8]

***Bin**ary **Sec**urity* est un ensemble d'outils open source développé pour améliorer 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 par des chercheurs de Verimag[9] et de LORIA[10]. Il est utilisé pour la recherche de vulnérabilités, la désobfuscation de logiciels malveillants et la vérification formelle de code assembleur. Grâce à l'exécution symbolique et l'interprétation abstraite, Binsec peut explorer et modéliser le comportement d'un programmes pour détecter des erreurs ; détection réalisée avec des outils de fuzzing et des solveurs SMT.

5. `https://hacl-star.github.io/`
6. `https://project-everest.github.io/`
7. Équipe de recherche rattaché au centre INRIA de Paris, focalisé sur les méthodes formelles et la recherche en protocoles cryptologiques. Pour ces objectifs, l'équipe développe des langages de programmation, des outils de vérification…
8. `https://binsec.github.io/`
9. 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 rejoins par le CNRS, il a pour objectif la sécurité dans les domaines des transports et de la santé.
10. Laboratoire lorrain de recherche en informatique et ses applications ; crée en 1997, c'est un centre de recherche commun au CNRS, l'Université de Lorraine, CentraleSupélec et l'Inria.

Première partie

Constant time ou pourquoi poser un lapin n'est pas une option

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 les attaques par canal auxiliaire 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 l'on tend à distinguer dans l'enseignement : le matériel et le logiciel. Pourtant, si on gardait séparé ces deux domaines, on aurait des tas de piles de métal et de plastiques ou des bibliothèques de livres plein d'idées intéressantes. Au contraire, combiner les les deux parties permet de réaliser des prouesses technologoqies et scientifiques. Ainsi, lorsque l'on conçoit un système sécurisé, il faut prendre en compte ces deux composantes. Or pour implémenter un système sécurisé, il ne faut pas seulement un logiciel sécurisé, il est aussi important que le matériel le soit. Oublier comment fonctionne un support informatique, c'est oublier que programmer se résume à manipuler de l'électricité.

Les attaques par canal auxiliaires consistent à exploiter les caractéristiques matériels du support pour gagner en connaissances sur le programme ciblé. Puis exploiter ces connaissances pour acquérir d'avantages d'informations privées : identifiants, clés secrètes, messages personnels. On leurs attribue le terme "canal auxiliaire" car il ne s'agit pas d'essayer de pousser dans ses limites un logiciel ou vérifier que tous les cas particulier sont gérés à travers le canal conçus par le développeur (une interface graphique souvent) mais plutôt de se positionner hors du cadre. Voici quelques travaux présentant une attaque par canal auxiliaire et surtout 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 de ces attaques est la nécéssité 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 enfin 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 un interférence. Ces interférence sont causées par une irrégularité de l'alimentation électrique, des radiations életromagnétique, une perturbation environnementale *etc* ... En 2004, BAREL et al. 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[1], NVM[2], DES[3], EEPROM[4], JVM[5]. On y retrouve enfin une liste de contre-mesures et de méthodes de protections contre ces attaques.

Ainsi, donner un accès physique à un inconnu est une porte d'entrée pour un attaquant. Pourtant, penser que l'accés 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 et al. 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'illuson de compartimentation entre les sessions, en réalité, les adresses mémoires 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 rempli le cache, puis mesure les temps d'accés 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 et al. existent [YGH16 ; Mog+17 ; VPS18], mais on observe rapidement que ces techniques emploient aussi la méthode de chronométrage. En effet, si on cible un algorithme et que l'on mesure son temps d'exécution, si en fournissant différentes entrées (que l'on considère secrètes) des variations sont observées entre les mesures, alors cela signifie que l'algorithme présente une dépendance à ces entrées. Généralement une sous-fonction de cet algorithme est responsable de ces variations. Cette classe d'attaque est appelée «*attaque temporelle*»[6].

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 temps et exécution 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 à celle-ci.

```
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 }
```

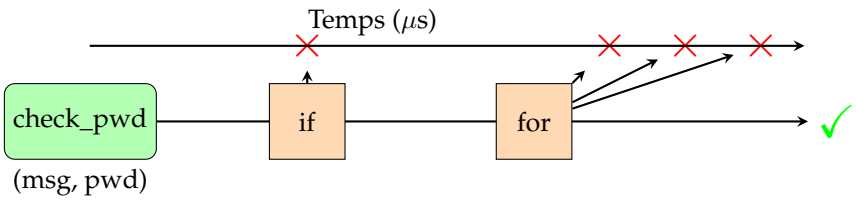
Code 1 – Exemple de code vulnérable à une attaque temporelle

1. Chiffrement asymétrique par clés secrète du nom de ces auteurs. Standardisé en 1983.
2. Non Volatile Memory ou mémoire non volatile est un composant informatique qui conservent son contenu en l'absence d'électricité.
3. Algorythme de chiffrement symétrique par bloc. Standardisé en 1977
4. Electrically-Erasable Programmable Read-Only Memory ou mémoire morte effaçable électriquement et programmable.
5. Machine virtuelle qui exécute des programmes compilés en bytecode Java.
6. Le terme générique dans la recherche scientifique est «*time attack*». Une traduction plus précise serait «*attaque par chronométrage*». on choisit ici d'utiliser le terme «*attaque temporelle*» car il est moins lourd et renvoie directement vers la faille exploitée plutôt que sur la méthode employée.

Si nous prenons le code présenté par le code 1, on peut observer que la fonction `check_pwd` compare deux chaînes de caractères. Si elles sont de même longueur, elle les compare caractère par caractère. Si elles sont de longueurs différentes, la fonction retourne immédiatement `False`. Ainsi, si l'on fournit un mot de passe de longueur différente, le temps d'exécution sera constant et court. En revanche, si l'on fournit un mot de passe de même longueur, le temps d'exécution dépendra du nombre de caractères identiques 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.

On peut synthétiser les éxécutions 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 cible 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 l'on suppose que le mot de passe est de 8 caractères de l'alphabet latin. On a alors 256 possibilités par caractère, pour un total de $256^8 = 2^{64}$ possibilités. En revanche, si l'on utilise la méthode de l'attaque temporelle, on peut réduire le nombre de possibilités à $8 + 8 \times 256 = 2056$ possibilités. En effet, on cherche dans un premier temps à identifier la longueur du mot de passe, puis on identifie charactère après charactère pour trouver le bon secret. Avec des temps d'exécution court on est dans les cas de figure d'échec, tandis qu'avec un allongement du temps d'exécution on sait que l'on est sur la bonne piste.

Les attaques temporelles présentent la particularité d'être générique. Tandis que les attaques décrites précédemment nécessite des conditions d'accés ou d'initialisation plus importante, cette classse d'attaque présente l'avantage d'être réalisable sur tous les types de systèmes, et notamment les systèmes accessible par internet. La connaissance de cette menace est donc primordiale pour l'implémentation et la mise en service de produit 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 on reprend le code 1, les branchement conditionnels ligne [4,6] sont des fuites d'informations. C'est grâce à ces instructions que l'attaque décrite précedemment 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. On y découvre les bonnes pratiques de programmation, les premiers outils automatique de vérification de code ainsi que les limitations auquelles est confronté le développeur qui souhaite être résistant à ces attaques.

2.1 Bonne pratique 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 on est 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 tel 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 à 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 compléxité 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 **sopérations 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. On a vs 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 observé après exécution.

Décalage et rotation

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

Saut conditionnel

Les saut conditionnels sont des instructions qui, comme pour les accès mémoires, 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 defausser la branche inutile ; ce qui entraîne un léger alentissement. 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 variable 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'instruction total.

En reprennant ces règles, on peut modifier notre exemple de code 1 et appliquer des modifications sur lignes que l'on a déjà ciblées comme fuites d'informations. Les modifications sont libre 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          if (msg_hash[i] != pwd_hash[i]) {
10             equal = equal && false;
11         } else {
12             equal = equal || false;
13         }
14     }
15     return equal;
16 }
```

Code 2 – Exemple de correction pour rendre un code résistant aux attaques temporelles

On voit 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 l'on souhaite utiliser. Enfin, le second branchement conditionnel est purement supprimé, le parcours des tableaux se fait entièrement.

Avec cette modification, on a un code 2 qui ne présente plus de fuite de données. Pourtant, on peut 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, on a alors introduis une nouvelle surface de fuite d'informations. Il faut vérifier notre code pour supprimer ce doute.

Outils de garanties

Plusieurs outils existent est 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, on peut 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.

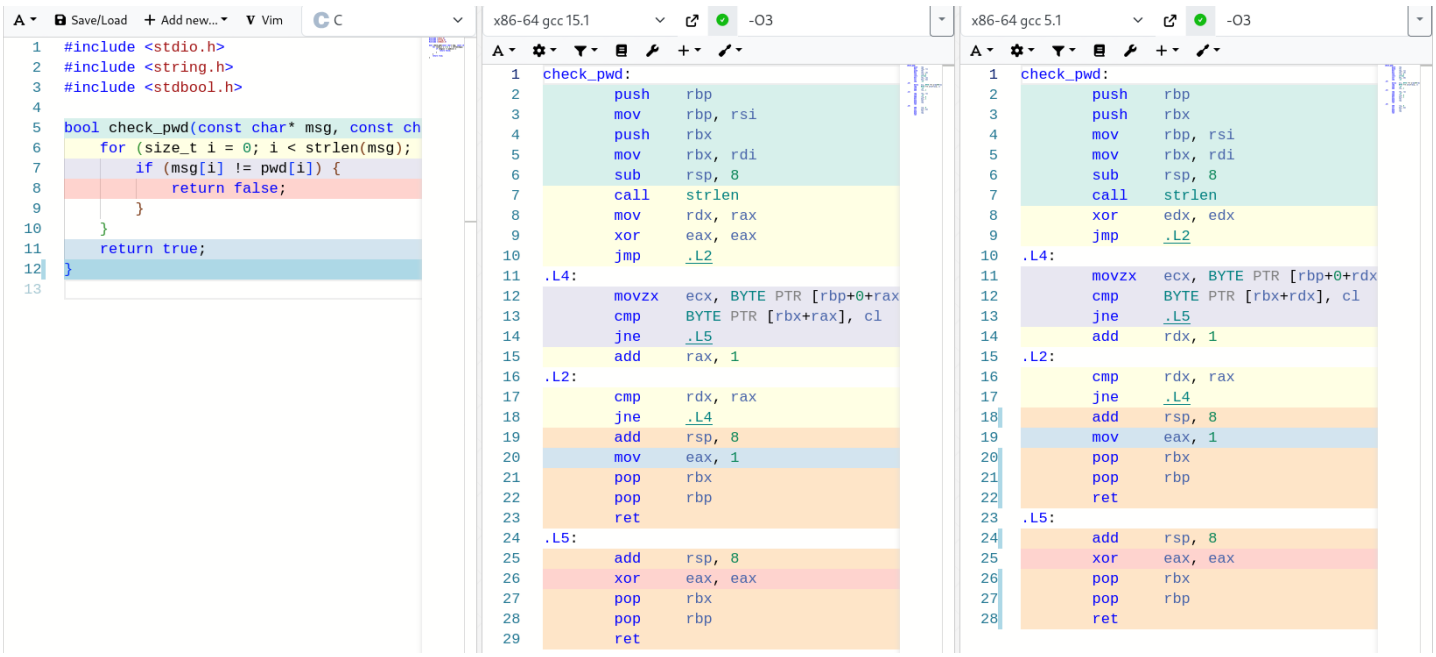


FIGURE 2.1 – Capture d'écran de comparaison de code assembleur x86_64 entre `GCC 15.1` et `GCC 5.1`

Si l'on souhaite 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 conçus pour vérifier la présence de fuite.

Plusieurs articles références l'ensemble des outils existant [Jan+21 ; Gei+23] pour réaliser ce travail. Le tableau 2.1 de JANCAR et al. liste 24 outils en libre accès conçus pour détecter des failles par canal auxiliaire.

Ils sont listés alphabétiquement et sont précisé le type de fichier analysé (*Cible*), la méthode d'analyse réalisée (*Techn.*) et les garanties attendues de ces analyses (*Garanties*). On reviendra plus en avant avec ces détails de méthodes et de fonctionnement 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 correct, ▲ = Correct 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* [Ler+05]. Cette solution rencontre en pratique de nombreux problèmes que l'on se garde 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 directment en assembleur. De cette manière, on a un contrôle total sur le flot d'exécution de notre programme, on peut ainsi insérer des optimisations qu'un compilateur pourrait ignorer. Écrire en assembleur requiert de connaître la plupart des opérandes disponibles pour l'architecture que l'on cible 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 synthaxe 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 librairie sécurisée est de pouvoir être employée par le plus de configurations possibles pour se protéger d'attaques.

Face à cette situation, on choisit donc d'utiliser un compilateur spécialisé ([Bor+21 ; RLT15]). Et à nouveau on se retrouve limité parce que ces compilateurs ne supportent pas l'ensemble du jeu d'instruction d'une architecture, ont besoin d'instructions supplémentaires (des annotations de code) pour réaliser la compilation, n'implémente pas les optimi-

sations qui apparaissent sur les processeurs les plus récents ou encore ne sont adaptés qu'à un seul langage de programmation.

À nouveau, on se retrouve donc à utiliser les compilateurs communs `GCC` et `LLVM` pour notre solution sécurisée. On se doit donc de programmer en respectant la politique temps constant. Et si cette pratique semble faire ses preuves, on peut lire dans la 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 a ensuite été prise en compte par SCHNEIDER et al. 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 et al. auprès des chercheurs de Hacl*. On peut voir deux fonctions dans le code 3, «*cmovznz4*» et «*FStar_UInt64_eq_mask*». La première appelle la seconde pour générer un masque qui sera ensuite appliqué au entrée de «*cmovznz4*». On a 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 3 – Fonction de masquage issu de *Hacl**

Avec le compilateur `RISC-V rv64gc clang 15.0.0`, si on entre les options de compilation `-O0` ou `-O1` on peut 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 réalisé et modifient le code pour accélérer son exécution. L'optimisation 2.2a suient les instructions précisées par le code source, de cette manière le compilateur réalise une compilation rapide. Au contraire dde 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 succesifs (les instructions `beqz`) permet une exécution

plus rapide. Les options de compilations sont rapportées 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 3 en fonction de différentes options de compilation données au compilateur, réalisée avec l'aide de *Compiler Explorer*.

Transition

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

Deuxième partie

Automatisme et vérification ou comment développer un détecteur de menace

CHAPITRE 3

Outils et méthodes

chapitre sur les outils + moyens pour détecter

intro

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 que l'on souhaite cibler. On parle parfois de *modèle de fuite*. Cette étape de synthèse et d'abstraction est importante pour identifer les risques encourus par le futur système, souvent en identifiant les point de fuites employés par les attaques déjà publiées. SCHNEIDER et al. [Sch+25] nous donne les trois modèles d'adversaires que l'on doit considérer lorsque l'on souhaite se défendre contre les attaques temporelles :

TABLE 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 des 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 mdodèles d'attaquant. Considérer le mode opératoire «récupération de traces» induit un modèle plus fort. Des travaux comme ceux de GAUDIN et al. [Gau+23] portent directement sur des améliorations matériel permettant une défense contre ce modèle. Considérer un attaquant plus puissant, avec des accès à des ressources supplémentaires, potentiellement hyptotétique, permet de concevoir un système plus sûr. Certains outils comme [HEC20 ; Wei+18] ou cette étude [Jan+21] exploitent notamment cette mécanique pour attester de la sécurité d'un programme.

Puis, avec ces modèles et les contre-mesures connus, on peut constituer un ensemble de règles qui valident ces risques. [Mei+21] résume celles-ci en une liste de trois règles :

1. Toute boucle révèle le nombre d'itérations effectuées.
2. Tout accès mémoire révèle l'adresse (ou l'indice) accédé.
3. Toute instruction conditionnelle révèle quelle branche a été prise.

Avec ces règles, il est alors possible de créer un outil qui analyse les programmes à sécuriser. C'est de cette façon que le premier outil existant a été produit : `ctgrind` (2010).

D'autres chercheurs comme DANIEL, BARDIN et REZK [DBR19] s'attellent à la création de modèles formels. Cette méthode demande un travail de formalisation du comportement de programmes binaire et une implémentation plus rigoureuse de leurs outils. Cela permet en retour une évaluation complète et correct de programmes complexes (*i.e.* primitives cryptographiques asymétriques).

Formalisation de modèle

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

On commence par définir un programme. Il s'agit d'une suite d'instruction binaire. Et une instruction est une action sur la mémoire. Cela nous permet de définir notre programme comme une suite de configuration (l, r, m) ; l la ligne d'instruction, r le dictionnaire de registre et m la mémoire. La configuration initiale est défini par $c_0 \triangleq (l_0, r_0, m_0)$ où l_0 est l'adresse de l'instruction d'entrée du programme, r_0 un dictionnaire de registre vide et m_0 une mémoire vide.

Ainsi, avec cette modélisation, une instruction est un changement appliqué à notre configuration. Ce changement peut être représenté par $c_0 \underset{f}{\rightarrow} c_1$, c_0 et c_1 deuc configurations successives, $\rightarrow$ la transition entre les deux et f une fuite émise par cette transition. Notons que certaines instructions ne produisent pas de fuites.

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

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

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

- formalisation des regles de securite

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

[DBR19]

3.2 Analyse d'un programme

4.1 Static analysis Static analysis approaches attempt to derive security properties from the program without actually executing it, extracting formally defined guarantees on all possible executions through binary or source code analysis. As a formal exploration of every reachable state is unfeasible, program behavior is often approximated, making them prone to false positives. Static approaches were the first to be considered, as side-channel security is closely related to information flow policies [53]. 4.1.1 Logical reduction Non-interference is a 2-safety property stat- ing that two executions with equivalent public inputs and poten- tially different secret inputs must result in equivalent public outputs. This definition covers side channels by considering resource usage (e.g., address trace) as a public output. Approaches based on logical reduction to 1-safety transform the program so that verifying its side-channel security amounts to proving the safety of the trans- formed program. Self-composition [23] interleaves two executions of a program P with different sets of secret variables in a single self-composed pro- gram P ;P' . Solvers can then be used to verify the non-interference property. This approach was used by Bacelar Almeida et al. [15] to manually verify limited examples, relying on a large amount of code annotations. ct-verif instead runs the two copies in lockstep, while checking their assertion-safety [13]. It is able to verify LLVM programs, leveraging the boogie verifier. Sidetrail [19] reuses this to verify that secret dependent branches are balanced (assuming a fixed instruction cost and excluding memory access patterns), providing a counter-example when this verification fails. However, such approaches suffer from an explosion in the size of the program state space. Blazer [17] verifies timing-channel security on Java programs by instead decomposing the execution space into a partition on secret-independent branches. Proving 2-safety is thus reduced to verifying 1-safety on each trace in the partition improving scalability at the cost of precision. Themis [48] uses static taint analysis to automatically annotate secret-dependent Java code with Hoare logic formulas as pre- and post-conditions. An SMT solver then verifies that the post-condition implies execution time differences remain bounded by given constant. Both tools provide a witness triggering the vulnerability otherwise. 4.1.2 Type systems Approaches based on verifying type safety of a program differ from language-level countermeasures [12, 30], as CCS '23, November 26–30, 2023, Copenhagen, Denmark the developer only needs to type the secret values with annotations instead of rewriting the program. The type system then propagates this throughout the program, similarly to static taint analysis. Type systems were considered relatively early to verify non-interference properties [7] and offer good scalability but their imprecision makes them difficult to use in practice. VirtualCert [22] analyzes a modified CompCert IR where each instruction makes its successors explicit. The authors define seman- tics for that representation, building the type system on top of it. An alias analysis giving a sound over-approximated prediction of targeted memory address is needed to handle pointer arithmetic. While this approach is more suited to a strict verification task, it can also provide a leakage estimate. FlowTracker [107] introduces a novel algorithm to efficiently compute implicit information flows in a program, and uses it to apply a type system verifying constant-time. 4.1.3 Abstract interpretation As a program semantics is generally too complex to formally verify non-trivial properties, abstract in- terpretation [50] over-approximates its set of reachable states, so that if the approximation is safe, then the program is safe. CacheAudit [55] performs a binary-level analysis, quantifying the amount of leakage depending on the cache policy by finding the size of the range of a side-channel function. This side-channel function is com-

puted through abstract interpretation, and the size of its range determined using counting techniques. It was later extended to support dynamic memory and threat models allowing byte-level observations [56] and more x86 instructions [91]. Blazy et al. [28] focus on the source code instead of the binary. Their tool is integrated into the formally-verified Verasco static an- alyzer, and uses the CompCert compiler. The analysis is structured around a tainting semantics that propagates secret information throughout the program. STAnalyzer [110] uses data-flow analysis to report secret-dependent branches and memory accesses. CacheS [126] uses an hybrid approach between abstract interpre- tation and symbolic execution. The abstract domain keeps track of program secrets—with a precise symbolic representation for values in order to confirm leakage—but keeps only a coarse-grain representation of non-secret values. To improve scalability, CacheS implements a lightweight but unsound memory model. 4.1.4 Symbolic execution Symbolic execution [81] (SE) denotes approaches that verify properties of a program by executing it with symbolic inputs instead of concrete ones. Explored execution paths are associated with a logical formula : the conjunction of condi- tionals leading to that path. A memory model maps encountered variables onto symbolic expressions derived from the symbolic in- puts and the concrete constants. A solver is then used to check whether a set of concrete values satisfies the generated formulas. Recent advances in SMT solvers have made symbolic execution a practical tool for program analysis [42]. CoCo-Channel [34] identifies secret-dependent conditions us- ing taint-analysis, constructs symbolic cost expressions for each path of the program uses SE and reports paths that exhibit secret- dependent timing behavior. Their cost model assigns a fixed cost per instruction, excluding secret-dependent memory accesses.CCS '23, November 26–30, 2023, Copenhagen, Denmark Several works use symbolic execution to derive a symbolic cache model and check that cache behavior does not depend on secrets. CANAL [117] models cache behaviors of programs directly in the LLVM intermediate representation by inserting auxiliary variable and instructions. It then uses KLEE [41] to analyze the program and check that the number of hits does not depend on secrets. Similarly, CacheFix [47] uses SE to derive a symbolic cache model supporting multiple cache policies. In case of a violation, CacheFix can synthesize a fix by injecting cache hits/misses in the program. CaSym [35] follows the same methodology and, to improve scala- bility, includes simplifications of the symbolic state and loop trans- formations, which are sound but might introduce false positives. SE suffers from scalability issues when applied to 2-safety prop- erties like constant-time verification. Daniel et al. [51] adapt its formalism to binary analysis, introducing optimizations to maxi- mize information shared between two executions following a same path. Their framework Binsec/Rel offers a binary-level CT analysis, performing a bounded exploration of reachable states and giving counterexamples for the identified vulnerabilities. Pitchfork [54] combines SE and dynamic taint tracking. It soundly propagates secret taints along all executions paths, reporting tainted branch conditions or memory addresses. Interestingly, Pitchfork can analyze protocol-level code by abstracting away primitives' im- plementations using function hooks, and analyzing them separately. ENCIDER [140] combines symbolic execution with taint analysis to reduce the number of solver calls. It also enables to specify information-flow function summaries to reduce path explosion. 4.2 Dynamic analysis Dynamic analysis groups approaches that derive security guar- antees from execution traces of a target program. Some form of dynamic binary instrumentation (DBI) is often used to execute the program and gather events of interest, such as memory accesses or jumps. Dynamic approaches differ in the events collected, and how traces are processed. They can be grouped depending on whether they reason on a single trace, or compare multiples traces together. 4.2.1 Trace comparison approaches Statistical tests. Statistical tests can be used to check if different secrets induce statistically significant differences in recorded traces. Cache Template [70] monitors cache activity to detect lines associ- ated with a target event, then finds lines correlated with the event using a similarity measure. A first pass using page-level observa- tions instead of lines can be used to improve scalability [112]. Shin et al. [114] use K-means clustering to produce two groups of traces for each line. The confidence in the partition indicates which line is likely to be secret-dependent. DATA [132] employs a Kuiper test then a Randomized Dependence Coefficient test to infer linear and non-linear relationships between traces and secrets. This was later extended to support cryptographic nonces as secrets [130]. Mutual information (MI) can be used to quantify the information shared between secret values and recorded traces, with a non-zero MI score giving a leakage estimation. MicroWalk [133] computes MI scores between input sets and hashed traces, with leakage lo- cation pinpointed using finer-grained instruction-level MI scores. MicroWalk-CI [134] optimizes this process by transforming the

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

Transition

CHAPITRE 4

Automatisme et couverture

chapitre sur les architectures à couvrir
les problèmes et les enjeux
les benchmarks en place
introduction Binsec
- intro

4.1 Outils et mode d'emploi

4.2 Emploi d'un usage industriel

Le premier outil à être créé est *ctgrind* [Lan10], en 2010. Il s'agit d'une extension à *Valgrind* observe le binaire associé au code cible et signale si une attaque temporelle peut être exécuter. En réalité, *ctgrind* utilise l'outil de détection d'erreur mémoire de *Valgrind* : Memcheck. Celui-ci détecte les branchement conditionnels et les accès mémoire calculés vers des régions non initialisée, alors les vulnérabilités peuvent être trouvées en marquant les variables secrètes comme non définies, au travers d'une annotation de code spécifique. Puis, durant son exécution, Memcheck associe chaque bit de données manipulées par le programme avec un bit de définition V qu'il propage tout au long de l'analyse et vérifie lors d'un calcul d'une adresse ou d'un saut. Appliquée à *Valgrind* l'analyse est pertinente, cependant, dans le cadre de la recherche de faille temporelle cette approche produit un nombre considérable de faux positifs, car des erreurs non liées aux valeurs secrètes sont également rapportées.

https ://blog.cr.yp.to/20240803-clang.html

Transition

Troisième partie

Érysichton ou avoir tellement faim que tu finis par manger ton corps

CHAPITRE 5

Implémentations pour un usage industriel

Ce chapitre permet de présenter le raisonnement qui a motivé la conception d'un outil de détection automatique de failles par canal auxiliaire de type temporelle.

5.1 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)
— métériel, contre des attaques très bas niveau (les attaques temporelles)

Avec l'objectif de concevoir un sytème sûr, il nous faut donc identifier toutes les tâches à réaliser pour arriver à bout de ce projet. En plus de ce travail de planification, l'identification et l'intégration d'outils déjà implémentées nous permettra de 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 concoivent et maintiennent des librairies cryptographiques, des boîtes à outils qui proposent les briques de sécurité nécessaires aux systèmes sécurisés.

Plusieurs librairies existent [AHa98 ; Por16 ; Pol+20] et remplissent différents objectifs : rétro-compatiblité, politique temps constant, *etc*. Notre choix est à réaliser en fonction des spécifités du produits que l'on cherche à déployer.

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

Utiliser la bibliothèque **Hacl*** [Pol+20 ; Zin+17] permet d'avancer la première étape et la première partie de la seconde étape. Cette bibliothèque a été conçue formellement et vient avec les preuves mathématique de la sécurité de son implémentation. Comme présenté en Préambule, cette librairie est programmée en F*. Le projet permet une exploitation en C et en assembleur [Zin+17].

En revanche, la seconde étape de la seconde partie nous demande une vérification au niveau de l'assembleur. Si certaine partie de cette librairie sont codées en assembleur, la majorité du projet reste du F* traduit vers C. Il faut réaliser une analyse. Dans le cadre de cette étude, l'outil d'analyse binaire retenu pour réaliser cette tâche est **Binsec**. Cet outil est implémenté en Ocaml et il est maintenu par une équipe de chercheurs ingénieurs géographiquement proche de l'équipe PROSECCO Inria. Cet avantage permet des échanges plus directs et donc une facilité quand à la mise en place du projet.

L'objectif est donc d'analyser Hacl* dans son entièreté. Avec cette analyse complète, si elle est correcte, alors les deux étapes de réalisation d'un système sûr seront réalisées. Cela signifie qu'elle sera la première librairie 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 comme chacun le sait, plus un binaire est imposant, plus son analyse est difficile. Et comme Binsec emploi l'analyse symbolique, explorer un binaire imposant a un coût de mémoire quadratique sur le parcours des instructions du binaire. L'idéal est donc d'analyser plein de petits fichiers binaires.

Analyse complète. Chaque fonction de Hacl* doit être analysée. En poursuivant la condition précédente, nous pouvons essayer de concevoir un binaire par fonction analysée. Nous distribuons ainsi l'analyse et on parcourt ainsi toute les fonctions présentent dans la librairie.

Analyse correcte. En se rappelant comment fonctionne les optimisations (voir le tableau A.1) il nous faut être attentif avec certaines qui simplifient/modifient le code. Pour éviter des suppressions d'instructions, le fichier nécéssite une légère contextualisation avant l'appel de la fonction analysée.

```
1  #include <stdlib.h>
2
3  #include "Hacl_AEAD_Chacha20Poly1305_Simd128.h"
4
5  #define BUF_SIZE 16384
6  #define KEY_SIZE 32
7  #define NONCE_SIZE 12
8  #define AAD_SIZE 12
9  #define TAG_SIZE 16
10
11 uint8_t plain[BUF_SIZE];
12 uint8_t cipher[BUF_SIZE];
13 uint8_t aead_key[KEY_SIZE];
14 uint8_t aead_nonce[NONCE_SIZE];
15 uint8_t aead_aad[AAD_SIZE];
16 uint8_t tag[16];
17
18 int main (int argc, char *argv[])
19 {
20 Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt
21     (cipher, tag, plain, BUF_SIZE, aead_aad, AAD_SIZE, aead_key, aead_nonce);
22 exit(0);
23 }
```

Code 4 – Code d'anlayse de la fonction Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt, testé lors de la prise main de Binsec et Hacl*

De même, comme nos fichiers analysés appartiennent à la librairie extérieur Hacl*, l'emploi de l'option `-static` est nécessaire pour prévenir la mise place de lien vers la librairie

partagée dans le ficher binaire. Cette option ne nuit pas à la qualité de l'analyse, elle permet en revanche d'avoir tous les éléments sous la main lorsque l'on déassemble un fichier binaire. Retirer cette option lors de la compilation, c'est se rajouter des lourdeurs et rallonger la temps requis pour la vérification manuelle d'un fichier.

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

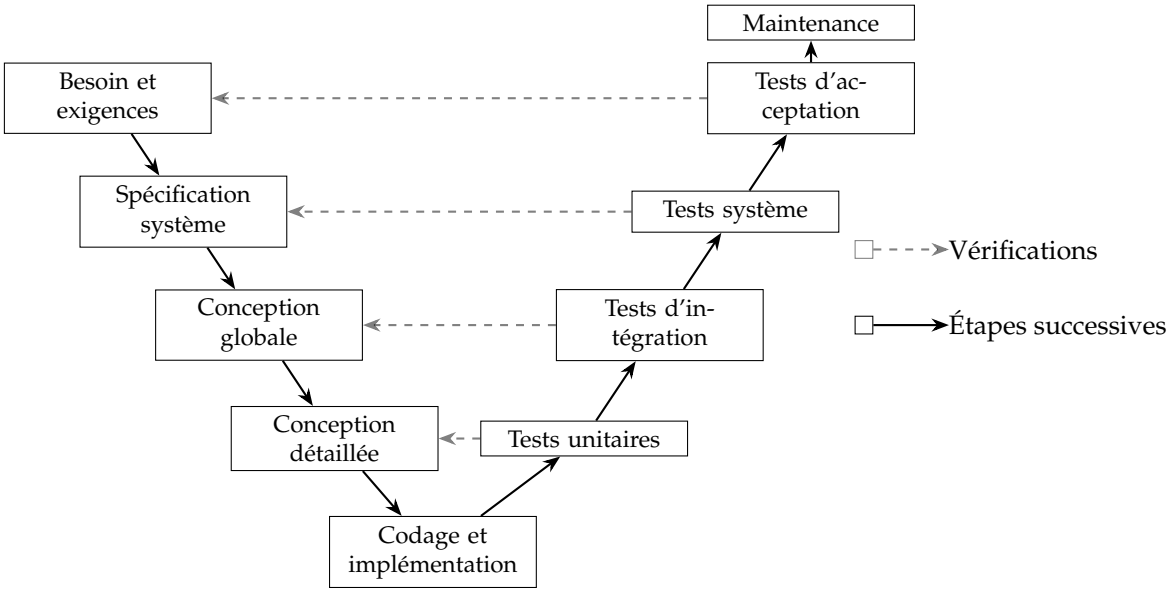
Couverture d'architectures. x86_64 et ARM sont les architectures matérielles les plus répandues dans le monde. Étendre l'analyse vers différentes plateformes et observer les différences qui émergent nous permettraient d'avancer dans la direction de la conception d'une librairie cryptographique universelle. 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, avec trois axes de compléxité (complétude, de la couverture d'architectures et des compilateurs) n'est pas envisageable à la main. Il faut absolument que cette analyse soit automatisée.

5.2 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 figures et d'usages, nous permettant de nous épargner les problèmes de corrections de bogues.

FIGURE 5.1 – Cycle en V



Appliquer cette méthode à l'entièreté de ce projet n'est pas envisageable à cause du coût temporel qui est très élévé. 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 nécéssiteront un futur travail.

Identification des besoins et exigences

Nous avons déjà conçu notre carnet d'exigence. 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 est un exemple de test réalisé dans cette phase du projet.

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

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

Code 5 – Commande Binsec basique

L'option `-sse` permet d'activer l'analyse par exécution symbolique, `-sse-script` associer à un fichier (ici `BINSEC_SCRIPT`) permet d'instruire notre analyse, préciser des stubs[1] et des initialisations. Enfin `-checkct` active la vérification 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 est accessible pour comprendre le fonctionnement l'outil Binsec depuis sa page officielle[2].

```
1  starting from core with
2    argv<64> := rsi
3    arg1<64> := @[argv + 8, 8]
4    size<64> := nondet              # 0 < strlen(argv[1]) < 128
5    assume 0 < size < 128
6    all_printables<1> := true
7    @[arg1, 128] := 0
8    for i<64> in 0 to size - 1 do
9      @[arg1 + i] := nondet as password
10     all_printables := all_printables && " " <= password <= "~"
11   end
12   assume all_printables
13 end
14
15 replace <puts>, <printf> by
16 return
17 end
18
19 reach <puts> such that @[rdi, 14] = "Good password!"
20 then print ascii stream password
21
22 cut at <puts> if @[rdi, 17] = "Invalid password!"
23
24 halt at <printf>
```

Code 6 – Instructions permettant de trouver le mot d'un passe d'un binaire exercice

Ce code présenté ici est un exempe d'usage de Binsec et permet de réaliser une attaque sur un binaire issu d'une plateforme d'apprentissage à la sécurité logiciel[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 7 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 vers d'autres architectures. Dans une première démarche d'automatisation, ce code a été généré automatiquement par un script shell. On voit ici 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 addresse 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 7 – Instructions permettant d'analyser le code 3 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

FIGURE 5.2 – 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	✓	✓	✓	✓	✓

✓ : *binaire secure*

Nous comprenons, à la lecture du tableau 5.2, 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.

FIGURE 5.3 – Tableau de résultats d'analyse Binsec pour architecture Risc-V

opt\fonction analysée	**cmovznz4** - 64 bits		**cmovznz4** - 32 bits	
Compilateur et architecture	gcc 15.1.0	clang 19.1.7	gcc 15.1.0	clang 19.1.7
-O0	~	×	~	×
-O1	✓	×	✓	×
-O2	✓	×	✓	×
-O3	✓	×	✓	×
-Os	✓	×	✓	×
-Oz	✓	×	✓	×

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

Les résultats dans le tableau 5.3 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 5.3, 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 part 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.

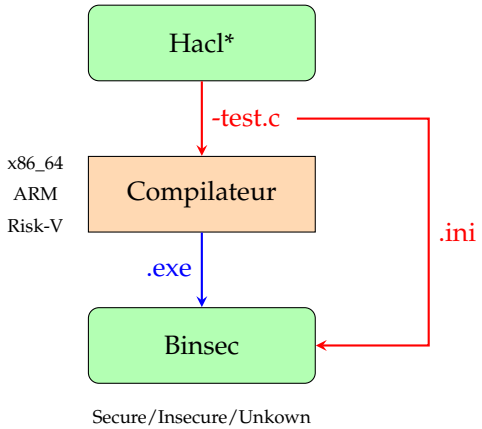
a. Ancronyme anglais pour Architecture de Jeu d'Instruction, désigne l'ensemble des instructions assembleur associées à une architecture.

En explorant plus en avant le code binaire, on découvre que ces erreurs sont dues à l'opérande `beqz`[4]. L'ISA de Risc-V n'a pas à sa dispostion 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 résulats 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 optmisations pour le code contenu entre les deux balises `off`,`on`. Cette solution entraîne des pertes de performance et des ralentissements quand 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 5.4 : depuis Hacl*, nous extrayons une fonction qui sera tester en un fichier 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 avec l'analyse Binsec.

FIGURE 5.4 – Flot de travail de l'outil d'analyse à concevoir



Nous allons maintenant nous pencher vers 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 6

Érysichthon à jamais affamé

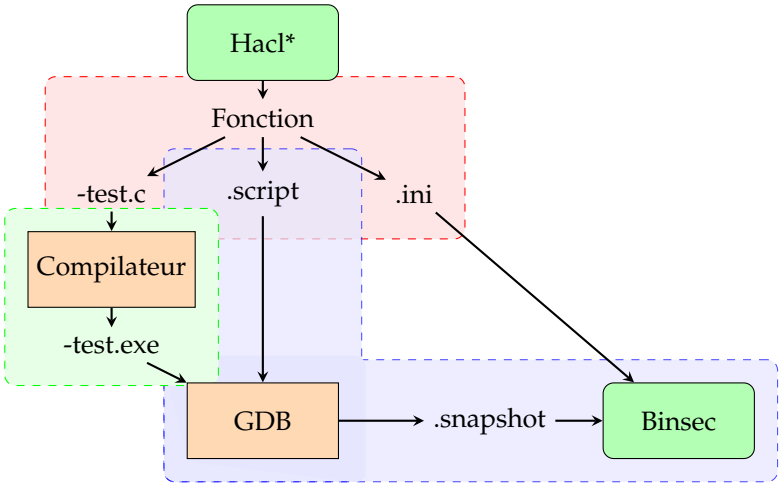
intro

6.1 Planification et préparations

Nous avons nos spécificités techniques et nous savons qu'elle forme notre outil doit avoir (fig **??**). Nous pouvons commencer par synthétiser les opérations nécessaires.

Nous allons donc concevoir des protocoles pour identifier les étapes nécessaires pour que Binsec analyse entièrement un fichier et nous renvoie un parmi [`secure, unknown, insecure`]. Le protocole `x86_64` est particulier. Depuis la version **0.5.0** de Binsec il est possible de fournir un «cliché mémoire» [1] pour accélérer l'analyse. Nous utilisons cet avantage pour l'intégrer à notre graphe d'éxécution.La machine sur laquelle le projet sera développé est sur une architecture x86_64, cela nous permet d'utiliser l'outil GDB pour la génération de cliché mémoire.

FIGURE 6.1 – Protocole pour analyser des fichiers compiler en x86_64

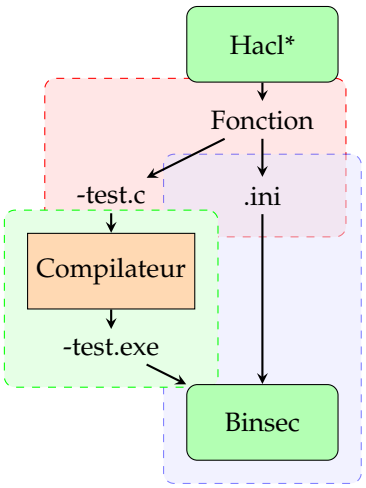


Ce graphe modélise la chaîne d'étapes nécessaire à l'obtention d'une analyse Binsec pour une fonction que nous ciblons. Plusieurs zones sont distinguées. La zone verte correspond à l'étape de compilation, la zone bleue à l'étape de préparation de l'analyse et la zone rouge à la synthèse de fichiers (de tests et d'instruction pour l'analyse de Binsec). Ce choix de couleur est adapté à la difficulté attendu de chaque étape. L'opération de compilation consiste en une commande. L'opération de préparation d'analyse consiste aussi en deux commandes : un appel à GDB avec le binaire puis un appel à Binsec avec le cliché mémoire et les instruction d'analyse.

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 analyser, généralement par un débogueur.

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 6.2 – 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 6.1 ?

En effet, l'importance de cette question se révèle lorsqu'on change d'architecture et que l'on doivent se passer de clichés mémoire. Sur notre machine en x86_64, si nous analysons un fichier compilé en ARM, alors on peut croiser des appels à des fonctions systèmes, les `IFUNC`. Or la résolution de ces fonctions est gérée dynamiquement lors de l'exécution du programme. Cette mécanique 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 qu'elles fonctions correspondent aux `IFUNC` qu'il peut croiser. pour illustrer ce point, observons le scipt nécessaire pour une vérification de la fonction «Hacl _AEAD _Chacha20Poly1305_Simd128_encrypt» compilé vers ARMv8.

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

secret global input1, aad1

@[0x00000048f008 ,8] := <__memcpy_generic>
@[0x00000048f018, 8] := <__memset_generic>
@[0x00000048f030 ,8] := <__memcpy_thunderx2>

starting from <main>
with concrete stack pointer


halt at <exit>
explore all
```

Code 8 – 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 lorsque qu'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 supplémentaires pour attraper les adresses qui ont besoin d'être réaffecté et leur attribuer les fonctions les plus adaptées.

Nommer un outil

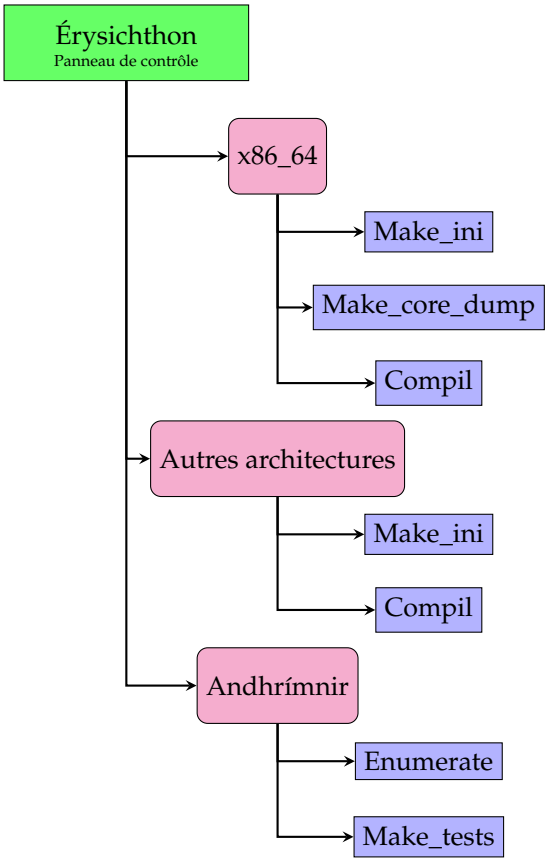
Rapidement il a fallu trouver un nom pour ce projet, l'appeler par "Notre outil… " devenait lourd et redondant entre les réunion hebdomadaire. En revanche trouver LE nom adéquat n'est pas une chose aisée, il peut être dû à une blague, une référence ou plus simplement liés au sens du projet. Dans notre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

Érysichthon est une 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 sera retenu pour la suite du projet.

Conception d'Érysichthon

Nous avons vus les protocoles nécessaires pour construire une analyse complète. Nous avons fait des tests pour comprendre le fonctionnement de Binsec et comment doivent être déclarer 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 script qui réalise une tâche au sein d'*Érysichthon*. Nous vous présentons en figure **??** comment s'organise les modules et les tâches qu'ils effectuent.

FIGURE 6.3 – 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 de Érysichthon est particulier car il est lui aussi nommé. Ce module consiste à produire 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 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 proposé à des outils d'analyse binaire. Ce module est aggrégé à Érysichthon mais peut être porter vers d'autre 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 quand à la complétude des tests produit. Toutes les fonctions présente dans le projet C analysé auront un test associé.

Ce module, comme son grand frère, est fonctionnel et abouti. En revanche, il nécessite quelques opérations manuelles supplémentaires et quelques amélioration pour pouvoir supporter n'importe quel projet C. Additionnellement il possède quelques optimisations propre à Hacl* permettant d'accélérer la mise en service d'Érysichthon.

6.2 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 découpé en plusieurs étapes. La génération de **548** fichiers de tests est réalisé moins de 2 secondes.

Enumerate

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

C'est lors de cette étape que nous spécifions ou retirons des fichiers ou plus précisément 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 débuguer.

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 assembleurs qui pourront être analyser. L'exemple 6.4 illustre comment sont générer nos fichiers de tests.

Une première partie initie le fichier. Cette partie contient les appels inclusifs de la librairie standard C, l'invocation de la librairie Hacl* au travers du fichier d'en-tête de référence (ici `Hacl_EC_K256`) et la signature de fabrication en commentaires. L'utilisation de la librairie standard permet d'utiliser la fonction `exit`. Avec cet appel, nous construisons nos scripts Binsec avec une intérruption 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.

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 places quelques optimisations.

Comme illsutré par la figure 6.5, 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 nécessaire à la constitution du fichier test. Une fois ces étapes réalisées, Andhrímnir commence sa préparation. Or il se trouve que quelques fois, certaines fonctions de Hacl* font appel à des structures propres à la librairie qui ont une instanciation particulière. Le module «`Détection appel auxiliaire`» permet de vérifier ce cas de figure.

FIGURE 6.4 – 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

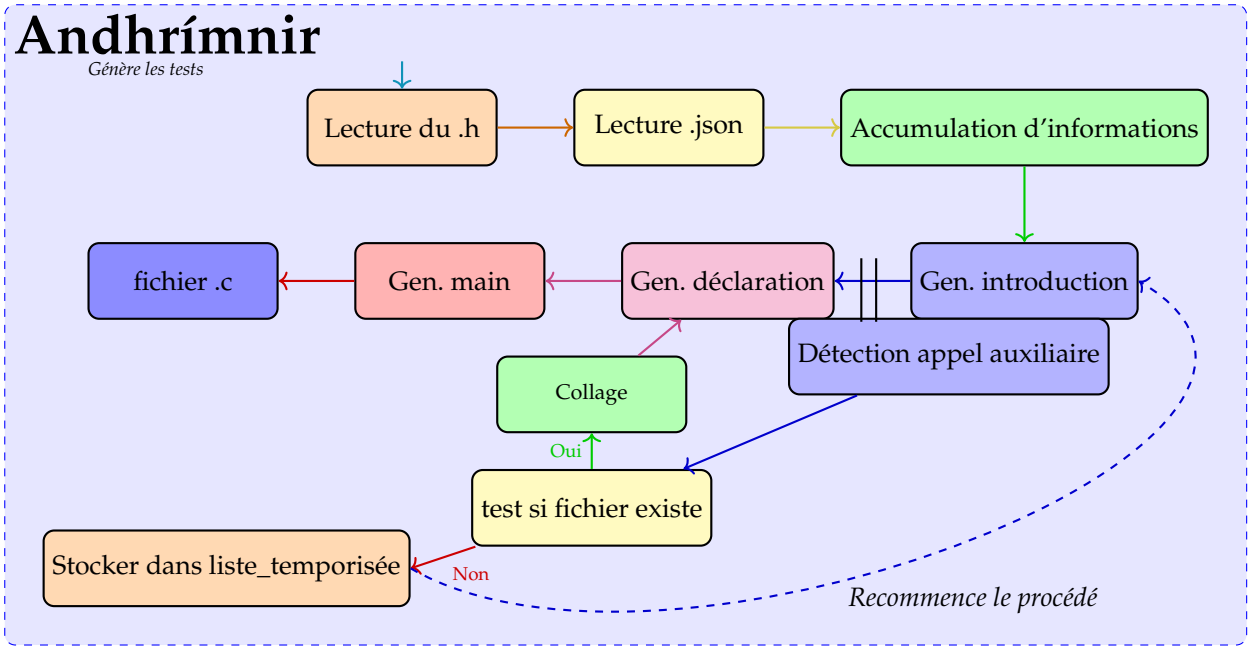
```
#define BUFFER_SIZE 5
uint64_t a[BUFFER_SIZE];
uint64_t out[BUFFER_SIZE];
```

Phase déclarative

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

Phase principale

FIGURE 6.5 – Schéma de conception d'Andhrímnir



Dans le cas où aucun appel n'est détecté, Andhrímnir continue sa préparation avec les étapes successives illustrées par la figure 6.4 : génération des déclarations puis génération du `main`.

À l'inverse où 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 finis son premier passage sur toute les fonctions d'Hacl*. Ce procédé est récursif pour pallier le problème d'appels en

cascade.

En réalité Andhrímnir ne recharge les informations d'une fonction qu'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 donées json existe pour accélerer le développement de Andhrímnir. Cela permet de venir ajouter manuellement des instructions haut niveau pour la conception des tests. Le code en annexe 9 illustre ce point : certaines fonctions ont besoins que les paramètres déclarés respecte certaines conditions. Cet exemple est accompgnée du fichier json associé et du fichier de tests final **??**.

Make_core_dump et Compilation

Les opérations de production de clichés mémoire et de compilation sont un assemblage de commandes shell et script pour GDB qui sont concevable sans problèmes. L'élément difficile à cet 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 à besoin d'être améliorer 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 associé à la fonction analysé. À la suite des exemples cités précédemment, le code 12 illustre comment ces instructions s'organise. Il est adapté pour l'architecture x86_64 et exploite la mécanique des clichés mémoires.

Un premier temps initie chargement des données, ensuite l'étiquette «`secret`» est accrochée aux variables à suivre durant l'analyse. Des commandes de gestion d'instruction particulières : des appels systèmes, des vérifications de registres inconnu de Binsec; permettent que l'analyse ne s'intérrompe pas et nous donne un résultat pertinent (`secure`, `insecure`). Enfin on indique notre arrêt d'exploration sur fonction «`exit`» et on donne notre feu vert avec la commande d'exploration totale.

Dans le cadre d'autre architectures, comme ARM, le code 8 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 conçu en fonction du support matériel sur lequel l'outil est activé.

6.3 Résultats

Conclusion

Discussion

point à revoir / améliorer / poursuivre

Ouverture

travaux directement sur le compilateur / support matériel

Conclusion

conclusion

Annexes

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

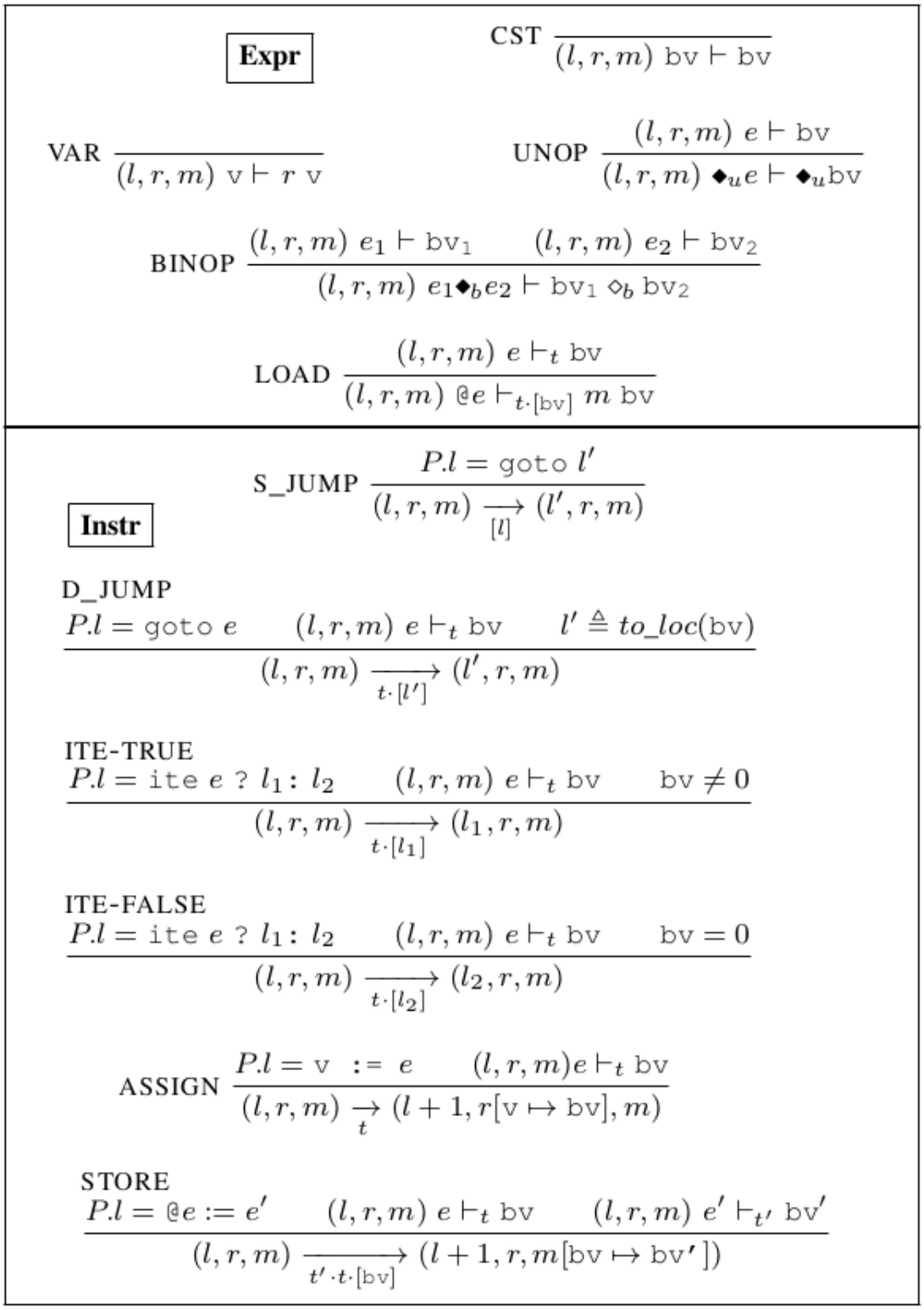


FIGURE A.1 – Ensemble d'instructions définis formellement par [DBR19]

ANNEXE B

Érysichthon, structure et exemples

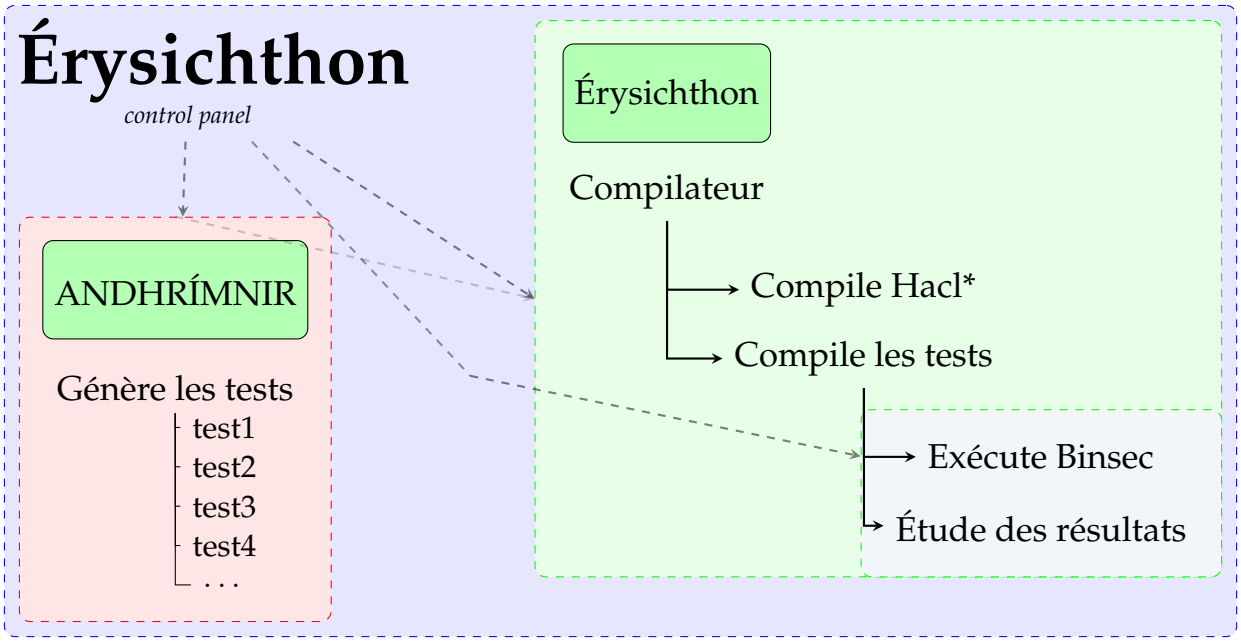


FIGURE B.1 – Structure d'Érysichthon, schéma du point de vue de l'usager
Les flèches grises indiquent tous les éléments actionnables individuellement.

```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 9 – 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 10 – 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 11 – 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 12 – Instruction Binsec générée automatiquement,
Hacl_AEAD_Chacha20Poly1305_Simd256_encrypt.ini

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 *et al., 2004*, URL : *https://eprint.iacr.org/2004/100*.

[Ler+05] CompCert, *Xavier* LEROY *et al., 2005*.

[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 et al., 2013".

[Doy+13] "'CacheAudit : A Tool for the Static Analysis of Cache Side Channels', Boris DOYCHEV et al., 2013".

[Bar+14] "'System-level non-interference for constant-time cryptography', Gilles BARTHE et al., 2014".

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

[Liu+15] "' Last-Level Cache Side-Channel Attacks are Practical ', Fangfei LIU et al., 2015, URL : `https://doi.ieeecomputersociety.org/10.1109/SP.2015.43`".

[Mas+15] "'Thermal Covert Channels on Multi-core Platforms', Ramya Jayaram MASTI et al., 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`".

[Alm+16] "'Verifying Constant-Time Implementations', Jose Bacelar ALMEIDA et al., 2016, URL : `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida`".

[Bar+16] "'Computer-Aided Verification for Mechanism Design', Gilles BARTHE et al., 2016".

[Pes+16] "'DRAMA : Exploiting DRAM Addressing for Cross-CPU Attacks', Peter PESSL et al., 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 et al., 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 et al., 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 et al., 2017".

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

[Ath+18] "'Sidetrail : Verifying time-balancing of cryptosystems', Konstantinos ATHANASIOU et al., 2018".

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

[Bre+18] "'Symbolic Path Cost Analysis for Side-Channel Detection', Thomas BRENNAN et al., 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 et al., 2018".

[Wic+18] "'Microwalk : A framework for finding side channels in binaries', Jan WICHELMANN et al., 2018".

[Wu+18] "'Eliminating timing side-channel leaks using program repair', Mingjie WU et al., 2018".

[Cau+19] "'FaCT : A DSL for timing-sensitive computation', Srinath CAULIGI et al., 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 et al., 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 et al., 2020".

[Wei+20] "'Big Numbers - Big Troubles : Systematically Analyzing Nonce Leakage in (EC)DSA Implementations', Samuel WEISER et al., 2020".

[Bor+21] "'Constantine : Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization', Pietro BORRELLO et al., 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 Attacks, *Jan* JANCAR *et al.*, *2021*, URL : *https://eprint.iacr.org/2021/1650*.

[Mei+21] Constant-Time Arithmetic for Safer Cryptography, *Lúcás Críostóir* MEIER *et al.*, *2021*, URL : *https://eprint.iacr.org/2021/1121*.

[Gau+23] "'Work in Progress : Thwarting Timing Attacks in Microcontrollers using Fine-grained Hardware Protections', Nicolas GAUDIN et al., 2023".

[Gei+23] "'A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries', Antoine GEIMER et al., 2023, URL : https://doi.org/10.1145/3576915.3623112".

[Sch+24] Breaking Bad : How Compilers Break Constant-Time Implementations, *Moritz* SCHNEIDER *et al.*, *2024*, URL : *https://arxiv.org/abs/2410.13489*.

[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 et al., 2025".

[SGP25] OwlC : Compiling Security Protocols to Verified, Secure, High-Performance Libraries, *Pratap* SINGH, *Joshua* GANCHER *et Bryan* PARNO, *2025*, URL : *https://eprint.iacr.org/2025/1092*.

[Tea25] Project Everest TEAM. *Project Everest : Perspectives from Developing Industrial-grade High-Assurance Software*. Rapp. tech. Project Everest, 2025.

Index