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 – 12 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

12 août 2025

Résumé

Résumé

Table des matières

Annexes

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 }
```

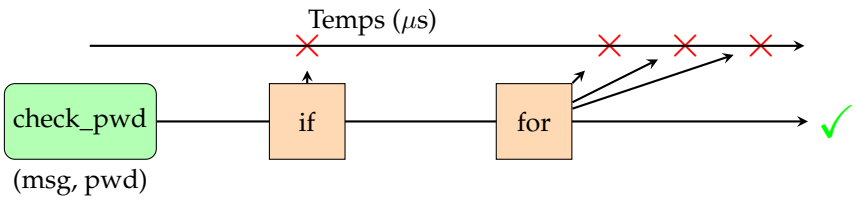
Code 1 – Exemple de code vulnérable à une attaque temporelle

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 ce perçoit directement grâce à la petite taille du code observé.

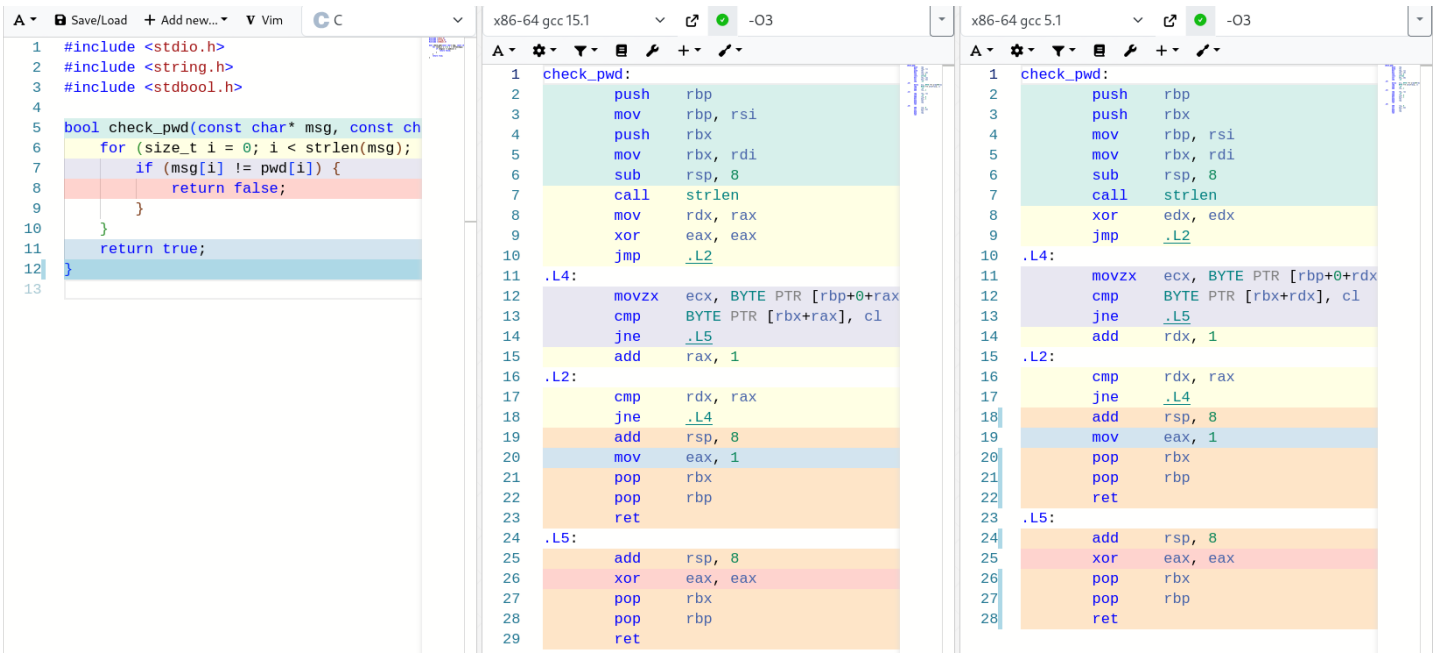


FIGURE 2.1 – Capture d'écran de comparaison de code assembleur x86_64 entre `GCC 15.1` et `GCC 5.1`

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 disponible pour l'architecture que l'on cible et les modèles des composants présent 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 configuration 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 adapté qu'un un seul langage de programmation.

À nouveau, on se retrouve donc à utiliser les compilateurs communs `GCC` et `LLVM` pour notre solution sécurisé. 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 performant 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` permettent d'identifier le tour de passe passe qui lui été proposé. Le code 2.2a suient les instructions précisées par le code source, le compilateur avec cette optimisation compile vite. Au contraire du code 2.2b produit par une analyse plus longue du compilateur où les sauts succesifs entre les `beqz` permettent une exécution plus rapide. Les options de compilations sont rapportées en annexe 1[2].

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

```
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*.

Deuxième partie

Automatisme et vérification ou comment développer un détecteur de menace

CHAPITRE 3

Outils et méthodes

chapitre sur les outils + moyens pour détecter

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 trace	Suivi des appels de fonctions, des accès réussis ou manqués à la mémoire.

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

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

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

D'autres comme [DBR19] emploient des modèles de fuites en représentation formelle. En s'appuyant sur les travaux de BARTHE, GRÉGOIRE et LAPORTE [BGL18]
- formalisation des regles de securite

Nous adoptons une position pessimiste, en supposant que chaque violation individuelle de ces règles fuit parfaitement vers l'adversaire.

La règle 1 se justifie par une observation triviale : une boucle plus longue utilise plus d'opérations. En pratique, il est difficile d'observer la durée de chaque boucle dans un programme plus vaste, ce qui rend cette règle pessimiste.

La règle 2 est justifiée par divers canaux auxiliaires et attaques basées sur le cache [**Ber05** ; **YGH17** ; **CAPGATB19**]. Puisque les caches ne chargent l'information qu'une ligne entière

à la fois, cette règle peut sembler trop pessimiste. Peut-être que seule la ligne de cache accédée devrait rester secrète [**Bri11**]. Malheureusement, il est possible de mener des attaques basées sur des accès à l'intérieur d'une ligne de cache [**BS13** ; **OST06** ; **YGH17**]. C'est pourquoi nous adoptons une position pessimiste, et supposons que les accès révèlent leur adresse exacte.

La justification de la règle 3 est double. Premièrement, si différentes branches d'une instruction conditionnelle exécutent un nombre différent d'opérations, on peut observer quelle branche a été prise. Deuxièmement, même si les deux branches exécutent des opérations identiques, le prédicteur de branche du processeur peut être exploité pour révéler des informations sur la branche sélectionnée [**AKS06** ; **AKS07** ; **EPAG16**].

En plus de ces règles, nous avons besoin d'un ensemble de base d'opérations de confiance pour construire nos programmes. Nous supposons que l'addition, la multiplication, les opérations logiques et les décalages, tels qu'implémentés matériellement, sont en temps constant par rapport à leurs entrées. C'est le cas sur la plupart des processeurs, une exception notable étant certains microprocesseurs [**Por**]. Cette hypothèse est raisonnable pour les plateformes ciblées par notre bibliothèque.

[DBR19]

3.2 Analyse d'un programme

- analyse statique - analyse dynamique - analyse symbolique - analyse de trace

CHAPITRE 4

Automatisme et couverture

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

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

Troisième partie

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

CHAPITRE 5

Implémentations pour un usage industriel

5.1 Identification des besoins et spécificités

On a pus voir grâce aux chapitres précédents que la conception et l'implémentation d'une système sécurisé est un problème difficile. Une première étape est de concevoir des primitives et protocoles mathématiquement sécurisés. Une seconde étape est de s'assurer que leurs implémentations sont effectivement sécurisées, d'abord d'un point de vue mathématique contre des attaques logiques (aspect fonctionnel : le code implémente correctement les bons concepts cryptographiques), mais aussi 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 implémenté nous permettra de d'avancer plus rapidement vers cet objectif.

Point de départ

En reprenant ces deux étapes, on va identifier quels sont nos leviers et nos possibilités pour un développeur pour avancer dans la conception de notre graal.

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 propose 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**. Cette outil

est implémenté en Ocaml et 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 que la première librairie cryptographique formellement sûre et résistante aux attaques temporelles sera conçus.

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'un carnet d'instructions à préciser. Avec ce point de départ, on peut commencer 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, on peut essayer de concevoir un binaire par fonction analysé. On distribue ainsi l'analyse et on parcourt ainsi toute les fonctions présentent dans la librairie.

Analyse correcte. Si on se rappel comment fonctionne les optimisations (voir le tableau 1) il faut faire attention avec certaines optimisations qui simplifient le code par soustraction d'oprations. Le fichier ne doit pas seulement contenir un appel de fonction, il faut une légère mise contexte.

```c
#include <stdlib.h>

#include "Hacl_AEAD_Chacha20Poly1305_Simd128.h"

#define BUF_SIZE 16384
#define KEY_SIZE 32
#define NONCE_SIZE 12
#define AAD_SIZE 12
#define TAG_SIZE 16

uint8_t plain[BUF_SIZE];
uint8_t cipher[BUF_SIZE];
uint8_t aead_key[KEY_SIZE];
uint8_t aead_nonce[NONCE_SIZE];
uint8_t aead_aad[AAD_SIZE];
uint8_t tag[16];

int main (int argc, char *argv[])
{
Hacl_AEAD_Chacha20Poly1305_Simd128_encrypt
    (cipher, tag, plain, BUF_SIZE, aead_aad, AAD_SIZE, aead_key, aead_nonce);
exit(0);
}
```

Code 4 – 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 font appel 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. Il faut donc identifier quel compilateur nous permet d'avoir des fichiers binaires les plus sécurisés. On peut aussi identifier quels optimisations produisent la rupture de sécurité dans le binaire en étudiant plus en avant le comportement de ceux-ci.

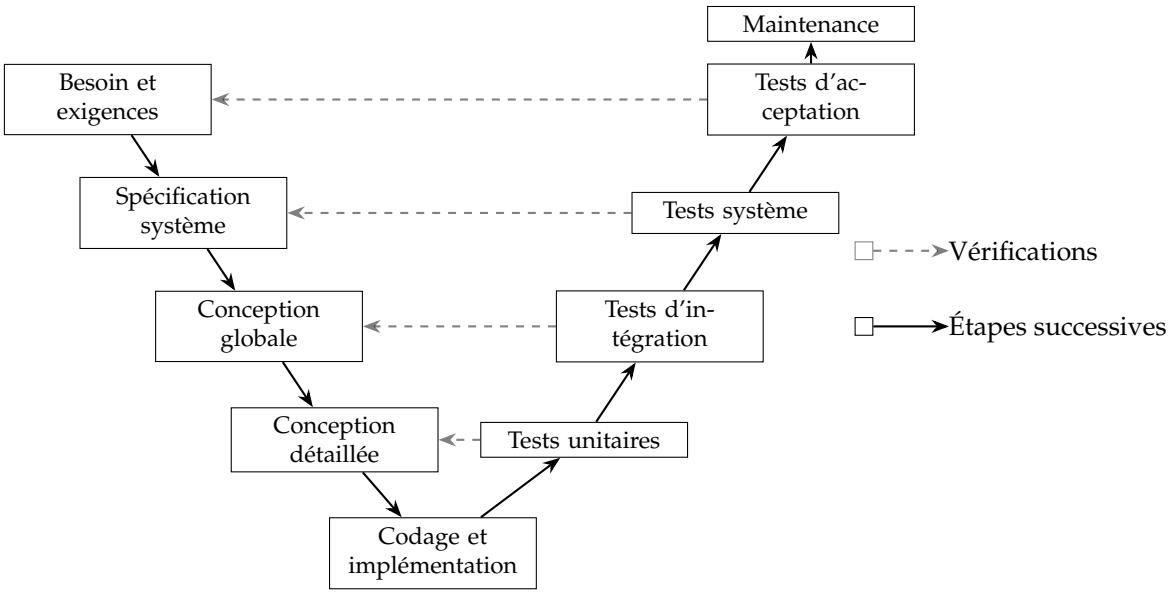
Couverture d'architectures. x86_64 et ARM sont les architectures matérielles les plus répandues dans le monde. Étendre l'analyse vers différentes plateformes et observer les différences qui émergent nous permettraient d'avancer dans la direction de la conception d'une librairie cryptographique universelle. On aussi étendre l'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 de un accident par siècle (métros automatiques, trains, avions…), les projets sont conçus selon le principe du cycle en V. Au contraire de la méthode Agile où on avance vers les problèmes en les résolvants au fur et à mesure, avec ce principe le développement est beaucoup plus long mais permet d'esquiver les problèmes qui, dans son contexte d'usage, peuvent entraîner des décés.

FIGURE 5.1 – Cycle en V



Appliqué cette méthode à l'entièreté de ce projet n'est pas envisageable à cause du coût temporel qui est très élévé. On va se concentrer sur la réalisation d'une preuve de concept et se concentrer sur la partie automatisation. La conception du produit sera minimale et le développement des couvertures sera soumis à un futur travail.

Identification des besoins et exigences

On a déjà conçus notre carnet d'exigence, en revanche on ne connais pas le comportement des outils que l'on souhaite employer. La première opération est de s'approprier le fonctionnement des outils que l'on s'apprête à employer. 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 de l'analyse qui sera effectué :

```
$ 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 des propriétés 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'instruction 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'exerice consistant 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.

Ce script a été conçu avec pour objectif de vérifier les résultats apportés par [Sch+24] concernant une fuite présente sur la fonction «*FStar_UInt64_eq_mask*» et d'étendre l'analyse vers d'autres architectures. Dans une première démarche d'automatisation, ce code à

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/

été généré automatiquement par un script shell. On voit ici que l'analyse ne parcourt pas l'entièreté du binaire, seulement 8 sections sont chargés (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	✓	✓	✓	✓	✓

Légende :

✓ : *binaire secure*

On comprend, à la lecture du tableau 5.2, que la politique temps constant est considérée respectée par Binsec sur les versions testé 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	✓	×	✓	×

Légende :

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

Les résultats dans le tableau 5.3 est indéniable : 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 on 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 géré. 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'autre erreurs dû à Binsec soient découverte. 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 construction 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 dus à l'opérande `beqz` : effectue un branchement si la valeur du registre consulté est zéro. L'ISA de Risc-V n'a pas accès à un opérande comme `cmov` en X86_64 ou ARM. Donc l'application d'optimisation de compilation encourage l'usage de cette opérande qui n'est pas en temps constant.

On a pus observer 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 donné dans le code source indique au compilateur de désactiver ses optmisation pour le code contenu entre les deux balise `off,on`. Cette solution entraîne des pertte de performance et des ralentissements quand au temps de compilation et à l'usage des ressources, il vaut mieux l'utiliser avec parcimonie que désactiver toutes les optimisations de compialtions.

CHAPITRE 6

Érysichton à jamais affamé

- point histoire
- structure / schemas
- usage
- Andrihminir

usage de l'outil, comment ça rend

Quatrième partie

Microarchitecture sécurisé ou comment concevoir un support sûr même s'il nous quitte des mains

CHAPITRE 7

Tour d'horizon des pratiques de conceptions

comment on fait en ce moment, pourquoi RISC-V, aide des constructeurs

CHAPITRE 8

Innovation scientifique

sujet de thèse rennes, idées des articles précédents

Conclusion

conclusion

Ressources

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

Table des figures

Listes des codes

Liste des tableaux

Annexes

TABLE 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