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 plus des connaissances en cryptologie, en architecture matérielle et en programmation bas niveau pour améliorer l'éfficacité de son travail.

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 programmation sécurisée. Et, si dans notre premier contexte on a des preuves mathématiques associées à nos systèmes sécurisés, avec la pratique de la programmation en temps constant ce n'est pas le cas.

En 2024, les travaux de SCHNEIDER et al. [Sch+24] prouvent qu'un usage inadapté des compilateurs introduis des failles exploitables. Ces résultats, observables partiellement avec des travaux antérieurs (par exemple [DBR19]), montrent qu'un usage inadapté d'options fournies au compilateur introduisent des failles dans un code admis sécurisé. 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 méchanisme 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.

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 code c'est deux composantes. Or pour implémenter un système sécurisé, il ne faut pas seulement un logiciel sécurisé, il faut aussi 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 exécuté. Puis, avec suffisamment de connaissances : usurper une identité, récupérer des clés secrètes ; acquérir des informations qui ne nous étaient pas destinées. On leurs attribuent 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 quelqus 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 ressources physiques 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'acès 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 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 qu'une dépendance à mes entrées existe. Et généralement une fonction de cet algorithme est responsable de ces variations. Cette classe d'attaque est appelée *attaque temporelle*.

Le lien entre temps et exécution de code est connus 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

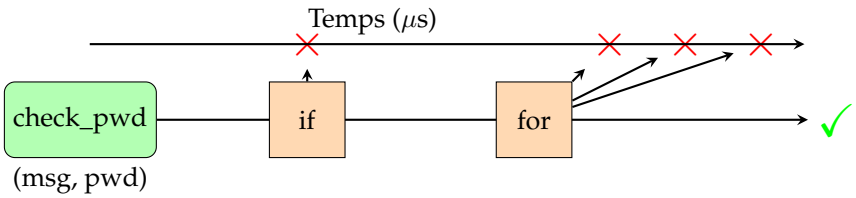
Si nous prenons le code présenté par le code 1, on peut observer que la fonction `check_pwd` compare deux chaînes de caractères. Si elles sont de même longueur, elle les compare caractère par caractère. Si elles sont de longueurs différentes, la fonction retourne immédiate-

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.

ment `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'acès ou d'initialisation plus importante, cette classse d'attaque à 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 bonne pratiques de programmation, les premiers outils automatique de vérification de code ainsi que les limitations auquel est confronté le développeur qui souhaite être résistant à ces attaques.

2.1 Bonne pratique et usages

Face à la menace des attaques temporelles, qu'elles 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, *Constant-Time Programming*, est une pratique de programmation qui vise a résoudre exactement ce problème. Directement lié à la compléxité algorithmique, cette pratique modifie et adapte les algorithmes pour que toutes les opérations effectuées est 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 vus 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 mise en mémoire ou pas encore ; de plus, les changement de valeurs 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. Certain 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 décalages. 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'algorithmes 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'information. 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.

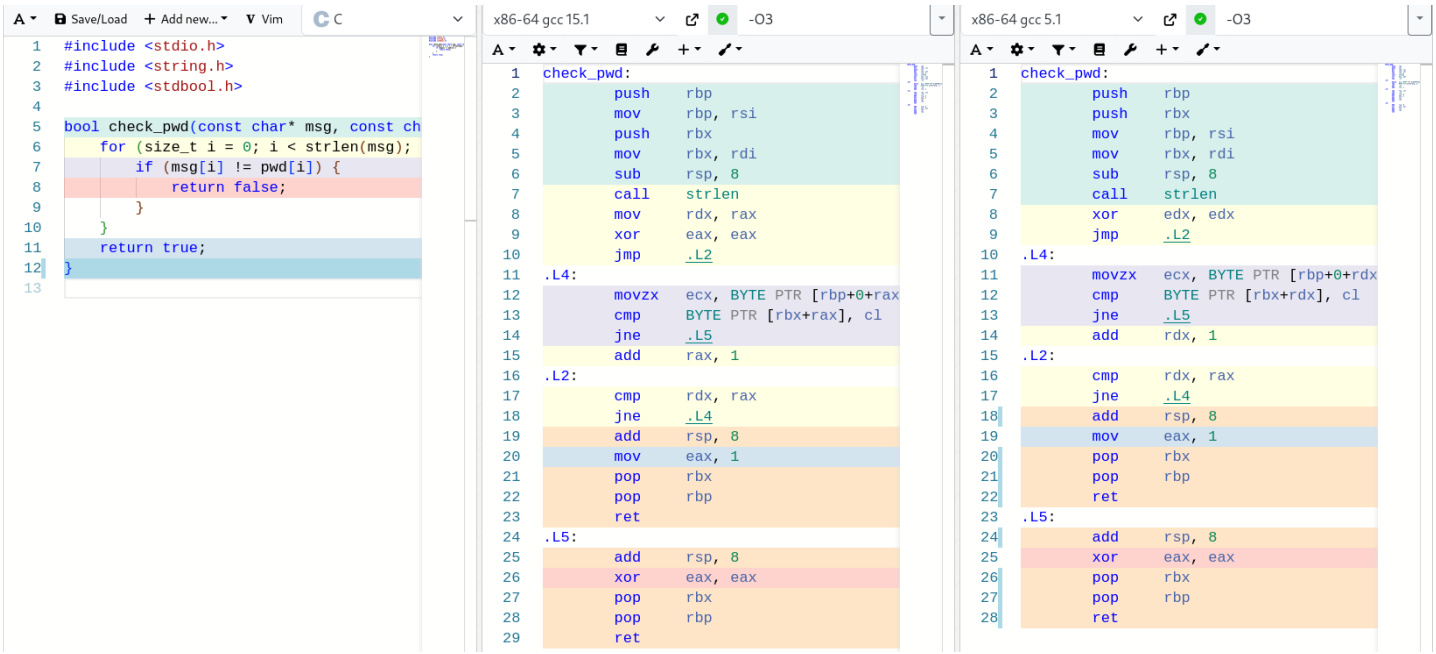


FIGURE 2.1 – Capture d'écran de comparaison de code assembleur x86_64 entre `GCC 15.1` et `GCC 5.1`

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 **??**.

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. 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, à l'inverse du code 2.2b où les sauts succesifs entre les `beqz` permet une exécution plus rapide. Les options de compialtions sont rapportées en annexe **??**[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*.