Université Paris Cité

FACULTÉ DE SCIENCES

UFR D'INFORMATIQUE

DIRECTRICE : CAROLE DELPORTE

RAPPORT DE STAGE DE MASTER 2

Compilation et optimisation d'algorithmes de chiffrement symétrique avec Usuba

Tuteur pédagogique : Ralf TREINEN,

Tuteur en entreprise : Pierre-Évariste DAGAND

Nathan CARACCIOLO

ANNÉE ACADEMIQUE 2021/2022

Remerciement

Je tiens à remercier toutes les personnes qui ont contribué au bon déroulement de mon stage et à la rédaction de ce rapport.

Dans un premier temps, je tiens à remercier mon maitre de stage, Pierre-Evariste Dagand, pour m'avoir proposé un stage de recherche dans un domaine qui m'intéresse énormément : la compilation. Je le remercie aussi pour son accueil et ses conseils au quotidien qui m'ont été d'une grande utilité. J'ai pu bénéficier à la fois de sa confiance dans mon travail et de son expertise dans ce domaine technique.

Je tiens aussi à remercier Darius Mercadier et Mattias Roux pour m'avoir aidés à comprendre le fonctionnement d'Usuba et avoir répondu patiemment à toutes mes questions.

Durant ce stage, j'ai pu assister aux séminaires du pôle Preuves, Programme et Système. Je tiens à remercier l'ensemble des intervenants pour leurs présentations extrêmement enrichissantes.

Pour finir, je tiens également à remercier ma famille pour leur soutien indéfectible ainsi que leur aide lors de la rédaction de ce rapport. Mais aussi mes amis pour avoir été présent tout au long de ce stage.

Introduction

J'ai effectué mon stage de deuxième année de Master au sein du cursus Langage et Programmation de l'Université Paris Cité à l'Institut de Recherche en Informatique Fondamental (IRIF) et ce rapport retrace l'ensemble des activités que j'ai pu effectuer durant ce stage de recherche.

Le sujet de mon stage s'intitule « compilation et optimisation d'algorithmes de chiffrement symétrique avec Usuba ». Les travaux effectués concernent le domaine de la compilation et celui du chiffrement. Ce rapport de stage présente différents outils et techniques utilisés ainsi que la méthodologie employée pour compiler et optimiser des algorithmes de chiffrement symétrique.

Résumé des travaux

Pendant mon stage, j'ai dans un premier temps travaillé à créer un pont entre deux langages spécialisés dans l'écriture d'algorithme de chiffrement. Pour cela, j'ai mis en place un processus de compilation. Ce processus a pour langage source *Usuba* : un langage haut niveau et le langage cible est *Jasmin* : un langage bas niveau. Ce travail permet de profiter du compilateur *Jasminc* et ainsi produire du code exécutable sur différents supports. C'est un problème intéressant car il nécessite la réécriture de plusieurs algorithmes classiques de compilation comme un allocateur de registre, un algorithme de spilling ou des analyses statiques qui sont de bons exercices de programmation. De plus, cela nécessitait une adaptation pour exploiter les spécificités des algorithmes de chiffrement, comme l'absence de branchement dynamique.

Dans un second temps, j'ai implémenté et généralisé une optimisation mise en place sur l'algorithme de chiffrement *AES*. Le but principal est d'estimer les gains de cette optimisation sur d'autres algorithmes de chiffrement de manière automatique. Ce travail a permis d'évaluer l'utilité de l'optimisation sur différents algorithmes de chiffrement symétrique et pourra être utilisé sur de futurs algorithmes.

Table des matières

1 Présentation du labo

J'ai effectué mon stage à l'institut de recherche en informatique fondamental (IRIF) situé à Paris sur le campus de l'université Paris-Cité. L'institue est composé de neuf équipes thématiques regroupées en trois pôles de recherche.

Pendant mon stage, j'ai travaillé au sein du pôle *Preuves, programmes et systèmes* (abrégé PPS) dont la mission principale est de renforcer les fondements théoriques des différents formalismes de calcul comme les langages de programmation ou encore les assistants de preuves.

Les pôles PPS est composé de trois équipes :

1. *Preuves et programmes* qui développe des langages théoriques issus de formalismes logiques.
2. *Algèbre et calcul* qui étudie les structures mathématiques liées au calcul.
3. *Analyse et conception de systèmes* qui modélise et analyse des systèmes de calcul réels.

J'ai personnellement travaillé au sein de l'équipe *Analyse et conception de systèmes*.

2 Usuba

Pendant mon stage de recherche, j'ai travaillé sur Usubac [1] : le compilateur d'un langage de programmation appelé Usuba [2]. Le langage et le compilateur sont développés au sein de l'équipe ACS. Le langage est de haut niveau et possède des abstractions spécifiques au domaine des algorithmes de chiffrement symétrique. Le compilateur Usubac est développé en OCaml et a comme cible le langage C.

2.1 La syntaxe de Usuba

Usuba est un langage possédant des constructions de haut niveau comme des boucles, des fonctions et des tableaux. Le langage possède également des constructions spécifiques à l'écriture des algorithmes de chiffrement symétrique comme des tables de permutations et des tables de recherche.

2.1.1 Construction standard de Usuba

Le langage Usuba ne manipule que des suites de bits et possède seulement des types permettant de les décrire. Il est par exemple possible de manipuler des données de 8 bits avec le type $b8$.

2.1.2 Tables de permutations

Les tables de permutation permettent d'exprimer facilement un réordonnement d'un ensemble de bit.

Algorithme 1 – Exemple de permutation écrite avec Usuba

```
perm perm1(i : b8) returns (o : b8) {
    6,4,1,5,7,8,2,3
}
```

La permutation `perm1` s'effectue sur 8 bits de données réorganise les bits comme décrit sur la Figure 1 : le premier bit de i devient le sixième bit de o, le deuxième devient le quatrième et ainsi de suite.

2.1.3 Tables de recherche

Les tables de recherche permettent d'exprimer aisément des substitutions de valeurs.

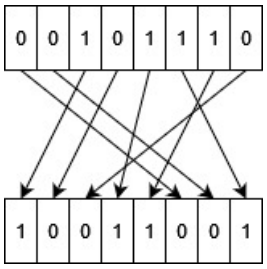

FIGURE 1 – Permutation de bits

Algorithme 2 – Table de recherche écrite avec Usuba

```
table table1(i : v4) returns (o : v4) {
    6,5,12,10,1,14,7,9,11,0,3,13,8,15,4,2
}
```

La `table1` prend 4 bits de données et associe les 2^4 valeurs possibles à une nouvelle valeur. Toutes les valeurs possibles sur 4 bits doivent se retrouver dans la table de recherche. Si les valeurs données à la table sont 3, 9 et 1, la table renverra respectivement 10, 0 et 5.

2.2 Bitslicing

Le bitslicing [3] est une technique d'optimisation permettant la parallélisation des calculs. Le bitslicing s'utilise sur des opérations bit à bit (*and*, *or*, *xor* et *not*). Les algorithmes de chiffrements utilisent beaucoup d'opérations bit à bit et bénéficient donc grandement des optimisations du bitslicing. Pendant mon stage, j'ai travaillé autour de cette technique de compilation qui est au coeur de la sémantique du langage Usuba.

L'idée principale peut se résumer à représenter des données de n-bits comme 1 bit dans n registres distincts. Cela permet de paralléliser des calculs bit à bit répétitifs sur des données ayant le même format. En possédant des registres de 64 bits, il est par exemple possible d'effectuer 64 calculs indépendants en une seule instruction processeur.

2.2.1 Exemple de bitslicing sur 3 bits

Imaginons que l'on ait un ensemble de données définies sur 3 bits et que l'on veuille effectuer l'opération *xor* sur le premier et le troisième bit de chaque donnée. La première manière qui nous viendrait à l'esprit serait d'effectuer chacun des calculs l'un après l'autre de la manière suivante :

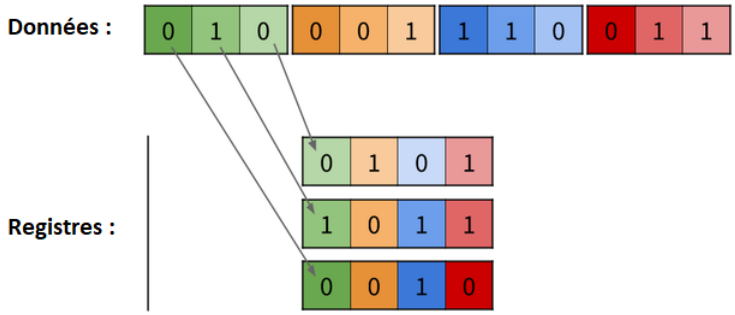

FIGURE 2 – Transformation des données pour pouvoir effectuer du bitslicing [3]

$r_0 = (d_{0_1}) \oplus (d_{0_3})$
$r_1 = (d_{1_1}) \oplus (d_{1_3})$
$r_2 = (d_{2_1}) \oplus (d_{2_3})$
...
$r_n = (d_{n_1}) \oplus (d_{n_3})$

Mais on peut aussi retransformer les données comme dans la Figure 2 afin d'effectuer du bitslicing et ainsi diminuer le nombre d'opérations à effectuer.

Il est alors possible de représenter un ensemble de 4 données de 3-bits dans 3 registres de 4 bits ce qui permet de remplacer les xor successifs par un unique xor entre le premier et le troisième registre comme décrit dans la Figure 3.

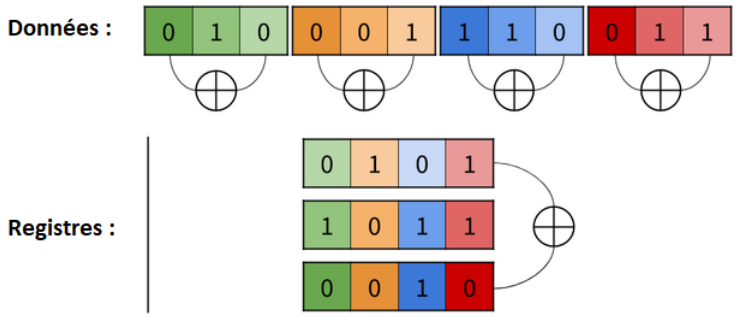

FIGURE 3 – Ou-exclusif effectué sur les données bitslicées [3]

3 Jasmin

Une des étapes importantes du travail que j'ai effectué pendant mon stage était de construire un lien entre le langage Usuba et le langage Jasmin [4]. Jasmin est un langage de bas niveau spécialisé pour l'écriture d'algorithme de chiffrement symétrique. Le langage possède une syntaxe à mi-chemin entre l'assembleur et le langage C.

Le projet possède son propre compilateur pour Jasmin : Jasminc [5] qui peut utiliser différents assembleurs en langage cible. Ce compilateur a été prouvé à l'aide de l'assistant de preuve Coq [6] qui offre des garanties sur la validité du code assembleur qu'il génère.

Le langage Jasmin n'étant pas muni d'un allocateur de registre, il est nécessaire d'en développer un afin de faire le lien entre les deux langages. En effet, le langage Usuba dispose d'une abstraction sur les variables (nombres de variable infinie) que Jasmin ne possède pas. Jasmin est un langage plus bas niveau séparant les variables stockées dans la pile de celles stockées dans les registres et possédant un nombre de registres disponibles fixé et restreint.

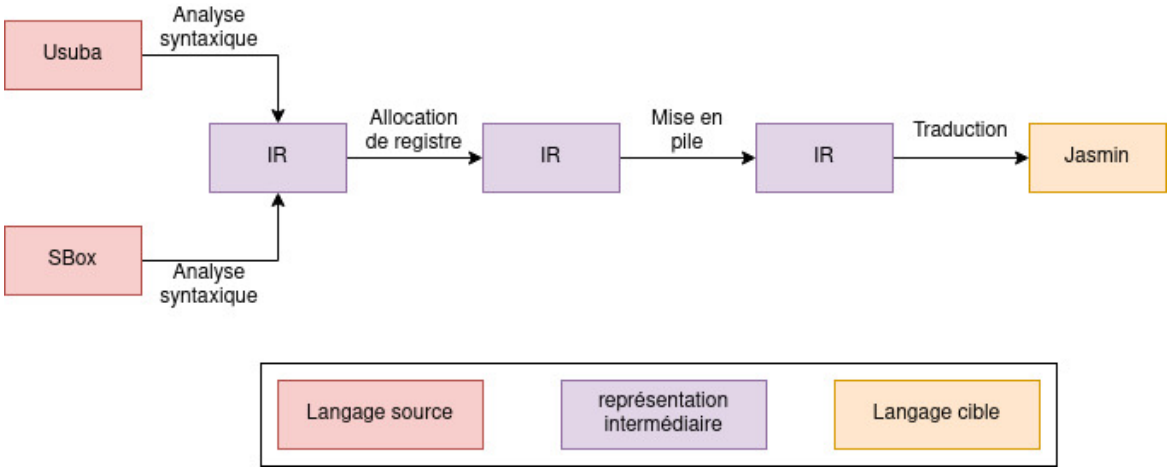

FIGURE 4 – Processus de compilation pour Jasmin

3.1 Langage d'entrée

Deux langages d'entrées ont été utilisés pour le processus de compilation : *S-Box* et *Usuba0*.

3.1.1 S-Box

J'ai travaillé au début de mon stage sur un premier langage [7, 8] simplifié en code à trois adresses. J'ai développé un parseur capable de lire ce langage et de le traduire vers la représentation intermédiaire. C'est un langage utile pour mettre facilement en place les différents algorithmes du processus de compilation.

Il m'a aussi servi d'environnement de test lorsque j'ai pu commencer à travailler sur des programmes plus imposants générés par Usuba.

3.1.2 Usuba0

Le second langage d'entrée Usuba0 est une forme réduite du langage Usuba. Le code est récupérable sous forme de *S-expression* à l'aide d'une option du compilateur Usubac.

3.2 Langage intermédiaire

Une méthode très répandue en compilation consiste à créer un langage intermédiaire afin de faciliter les transformations successives de programmes. C'est une méthode que j'ai pu appliquer en créant un langage intermédiaire ne contenant que les informations nécessaires à la suite de la compilation.

Du fait de l'ensemble réduit d'instructions permettant de décrire les algorithmes de chiffrement, le langage intermédiaire est très succinct et ne possède que deux instructions (Move et Oper) suffisantes pour exprimer l'ensemble des programmes produits par Usuba.

Dans la suite, l'ensemble des transformations de programme décrites opèrent sur ce langage intermédiaire.

3.2.1 Définition du langage intermédiaire

L'AST du langage intermédiaire est définie dans l'annexe 6.1. Le langage intermédiaire ne permet que la manipulation d'*entiers signés* sous la forme de valeur (*_val*). Il est aussi possible d'utiliser des variables (*_var*) représentées sous la forme d'une *chaîne de caractères*.

Le langage fait la distinction entre les sources des opérations (*src*) et les résultats des opérations (*dst*). Les sources des opérations peuvent être des variables ou des valeurs alors que les destinations ne peuvent être que des variables. Les deux instructions utilisent des données sources et destinations d'une manière spécifique.

Le *langage intermédiaire* est constitué de l'instruction Move qui est un simple déplacement de mémoire similaire en tout point à l'instruction *mov* de l'assembleur.

L'opération Oper sert à faire toutes les opérations binaires que demande les algorithmes de chiffrement. Il peut s'agir d'opération bit à bit (*and*, *or* et *xor*) ou d'opération arithmétique (*sum*, *sub*...). L'ensemble des opérations sont contenu dans le type **op**.

3.2.2 Contraintes du langage de sortie

Pour parfaire la compilation de Usuba vers Jasmin, les étapes du processus de compilation doivent nécessairement respecter les différentes contraintes liées à l'expressivité du langage Jasmin. En effet, le langage étant très bas niveau, son expressivité est restreinte et certaines utilisations sont tolérées par la syntaxe, mais pas par le compilateur Jasminc.

Jasmin étant un projet de recherche, les restrictions ne sont pas toutes explicitement citées dans la documentation du logiciel, il a donc fallu tâtonner et comprendre l'ensemble des contraintes que le langage comprend afin de les respecter.

On peut citer par exemple une contrainte qui fait que si deux registres sont utilisés lors d'un calcul, seul l'un d'entre eux pourra maintenir sa valeur et l'autre registre sera écrasé et sa valeur perdue. Mais aussi l'impossibilité d'utiliser deux variables provenant de la pile pour un calcul, une limitation que l'on trouve aussi en assembleur avec l'impossibilité d'utiliser deux variables provenant de la mémoire pendant une instruction.

J'ai donc développé plusieurs fonctions permettant de reformater et de réorganiser le code de la représentation intermédiaire pour qu'il respecte les contraintes de Jasmin au moment de sa traduction.

3.3 Allocation de registre

Le premier algorithme sur lequel j'ai travaillé était un algorithme d'allocation de registre. C'est un algorithme incontournable des compilateurs et j'ai donc commencé par établir l'état de l'art [9] et me renseigner sur les algorithmes d'allocation de registre existant décrit dans le Tiger Book [10]. L'allocateur de registre devait s'inspirer des algorithmes déjà existants, mais prendre en compte les spécificités de la représentation intermédiaire comme l'absence de branchement.

L'allocation de registre permet d'attribuer à chaque variable d'un programme un emplacement mémoire, qu'il s'agisse des registres ou bien de la pile. Cela permet de supprimer l'abstraction des variables infinies.

L'allocation de registre que j'ai développé effectue une analyse de vivacité en lisant le programme en partant de la fin. Pour chaque instruction i l'analyse détermine les ensembles def_i et use_i.

L'ensemble def_i contient l'ensemble des variables définies par l'instruction i tel que $def(\mathsf{Move}(dst, src)) = \{dst\}$ et que $def(\mathsf{Oper}(o, dst, s1, s2)) = \{dst\}$. L'ensemble use_i contient l'ensemble des variables utilisés par l'instruction i tel que $use(\mathsf{Move}(dst, src)) = \{src\}$ et que $use(\mathsf{Oper}(o, dst, s1, s2)) = \{s1; s2\}$. L'ensemble des variables déclarées à une instruction i donnée se note mem_i. Les valeurs de l'ensemble sont données par l'équation $mem_i = (use_i \cup mem_{(i-1)}) \backslash def_i$

i	instruction	$(use(i) \cup mem_{(i-1)}) \backslash def(i)$	mem_i
3	x = 1	$(\{\} \cup \{x\}) \backslash \{x\}$	{}
2	y = 0	$(\{\} \cup \{x; y\}) \backslash \{y\}$	{x}
1	i = x $\vee$ y	$(\{x; y\} \cup \{x; y\}) \backslash \{i\}$	{x ; y}
0	j = x $\wedge$ y	$(\{x; y\} \cup \{\}) \backslash \{j\}$	{x ; y}

Chaque variable est renommée pendant cette étape et le nom d'un registre libre lui est attribué. Une table *usage* associe à chaque registre son nombre d'utilisations. On note r_n le $n^{\text{ième}}$ registre.

i	instruction	renommage	*usage*
3	x = 1	$r_1 = 1$	$\{r_0 : 0 ; r_3 : 0\}$
2	y = 0	$r_2 = 0$	$\{r_0 : 0 ; r_1 : 2 ; r_3 : 0\}$
1	i = x $\vee$ y	$r_3 = r_1 \vee r_2$	$\{r_0 : 0 ; r_1 : 2 ; r_2 : 2 ; r_3 : 0\}$
0	j = x $\wedge$ y	$r_0 = r_1 \wedge r_2$	$\{r_0 : 0 ; r_1 : 1 ; r_2 : 1\}$

L'algorithme réassigne en priorité les variables dans les registres ayant le plus d'utilisations répertoriées dans *usage*.

L'algorithme s'inspire des analyses statiques de programme et utilise des fonctions du Tiger Book comme *use* et *def*. Ces deux fonctions permettent respectivement de récupérer sur une instruction : les variables utilisées et les variables définies.

3.4 Mise en pile

La mise en pile a pour but de limiter à la fois l'utilisation de la pile et le nombre de transferts entre la pile et les registres. Cela a pour de maximiser l'utilisation des registres. Pour ce faire, des instructions sont ajoutées au programme pour mettre en piles les variables (« spill ») quand elles ne sont pas utilisées afin de libérer des registres pour les autres variables du programme.

L'étape de mise en pile utilise les métriques d'utilisation calculées par l'allocation de registre et stockées dans la table *usage*. Il sépare l'ensemble des variables en deux groupes, les variables *de registre* et les variables *de la pile*. L'algorithme parcourt chaque instruction et cherche à déterminer si un spilling est nécessaire. Pour une instruction i, l'algorithme ajoute une mise en pile si les deux conditions suivantes sont respectées :

1. Une variable de la pile est utilisée pendant l'instruction.
2. Une variable stockée dans un registre n'est pas utilisé durant les sept instructions suivantes.

Le nombre sept est la configuration permettant d'obtenir les meilleurs gains de performance pendant la mise en pile. Je l'ai obtenu en comparant les performances et en faisant varier le nombre d'instructions durant lesquelles le registre ne devait pas être utilisé.

Si les deux conditions sont validées, l'algorithme ajoute une instruction pour *push* la variable qui n'est plus utilisé dans un registre. Il applique aussi un renommage sur l'ensemble des instructions restantes afin que la suite du programme reste cohérente avec cette transformation.

Dans le cas où les deux conditions ne sont pas validées, l'instruction i n'est pas modifiée et l'algorithme passe à l'instruction $i+1$.

instructions	instructions avec mise en pile
$r_1 = 1$	$r_1 = 1$
$r_2 = 0$	$r_2 = 0$
$r_2 = r_1 \vee r_2$	$r_2 = r_1 \vee r_2$
$r_1 = 0$	$r_1 = 0$
$p_1 = 1$	$p_1 = r_2$
...	$r_2 = 1$
...	...
$p_{11} = r_2 \wedge p_9$	...
	$r_2 = p_1$
	$p_{11} = v_2 \wedge p_9$

Sur cet exemple les variables r_1 et r_2 sont des variables de registre, toute les autres sont dans la pile. On peut voir sur cet exemple simplifié que deux instructions ont été ajoutées afin d'inverser les variables p_3 et r_2. De plus, un renommage a aussi été effectué, remplaçant les occurrences de la variable p_3 par r_2.

3.5 Évaluation

Dans le but d'évaluer la performance et la qualité du code généré par l'allocateur de registre et l'étape de mise en pile, j'ai effectué une évaluation de performance (benchmark). Le benchmark que j'ai mis en place permettait de comparer les performances du code généré par mon processus de compilation avec Jasminc avec celles générées par Usubac avec un compilateur C. J'ai utilisé les compilateurs C : Gcc et Clang avec l'option d'optimisation maximum *O3*.

3.5.1 Micro-Benchmark

J'ai utilisé la librairie de micro-benchmark développée par Google [11] pour étudier avec précision les performances des algorithmes produits en sortie du processus de compilation Usuba. Il était important d'utiliser un système de mesure précis, car le temps d'exécution des algorithmes sont de l'ordre de la nanoseconde.

3.5.2 Analyse des résultats

Les résultats démontrent une différence de performance entre le code produit par mon processus de compilation. Les performances indiquées par le benchmark sont 6% en deçà

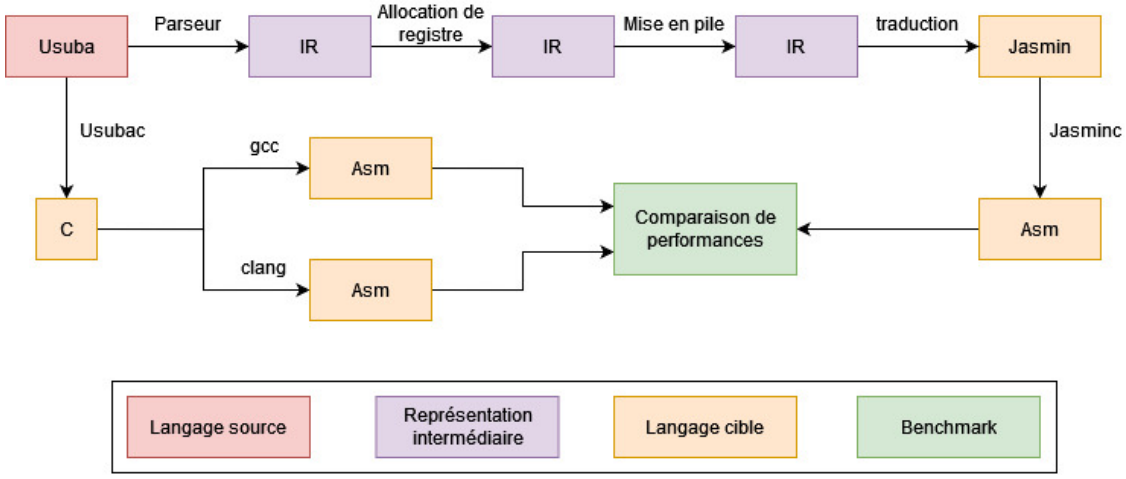

FIGURE 5 – Environnement mis en place pour comparer les performances des programmes générés

des performances du code obtenu avec gcc. Les performances sont aussi 7% en deçà des performances obtenues avec clang. On peut expliquer les différences de performance par les meilleures heuristiques utilisés par gcc et clang pour choisir le moment opportun pour mettre en pile. Les compilateurs gcc et clang possèdent aussi des étapes supplémentaires pendant l'allocation de registre comme le *scheduling* que je n'effectue pas pendant mon processus de compilation.

3.6 Conclusion

Dans cette partie, j'ai détaillé le travail effectué afin d'établir un lien entre Jasmin et Usuba. J'ai pu développer un processus de compilation complet entre ces deux outils. Grâce à Jasmin il est possible de compléter le processus de compilation d'Usuba vers les différents langages machines que propose le compilateur Jasminc.

Travailler sur l'algorithme d'allocation de registre était un bon choix. Cela permettait de commencer à développer rapidement tout en étudiant la syntaxe spécifique du langage Usuba. C'était un très bon moyen d'appréhender Usuba.

4 Optimisation du mode CTR

La seconde partie de mon stage a consisté à étudier, implémenter et généraliser une optimisation sur des algorithmes de chiffrement. Cette optimisation a été mise en place sur l'algorithme *AES* (Advanced Encryption Standard) [12] dans le papier Fast AES CTR mode Encryption (FACE) [13]. J'ai cherché à reproduire l'optimisation et à la généraliser à l'ensemble des algorithmes de chiffrement [14] afin de voir si des gains de performances significatives pouvaient être obtenus.

C'était un travail très intéressant car il fallait comprendre en détail certaines spécificités des algorithmes de chiffrement pour pouvoir y appliquer des techniques d'optimisations comme la mémoïsation définie dans la section 4.3. J'ai pu obtenir des résultats pour l'algorithme AES que j'ai pu comparer aux résultats de *FACE* mais aussi obtenir des nouveaux résultats pour d'autres algorithmes montrant les gains de performances envisageables.

4.1 Le mode CTR

L'optimisation sur laquelle j'ai travaillé utilise le mode de chiffrement CTR [15]. Le mode CTR (abréviation de « counter ») est un mode d'opération de chiffrement qui utilise un compteur dont la valeur d'initialisation n'est pas connue à l'avance. Les différentes valeurs du compteur seront chiffrées avant d'être utilisées pour chiffrer les données comme on peut le voir sur la Figure 6.

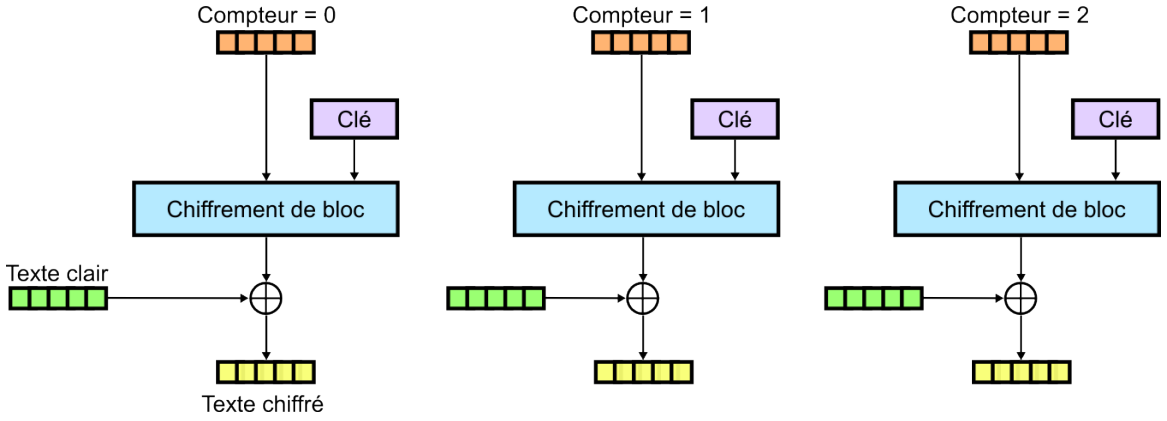

FIGURE 6 – Mode CTR [16]

Ce mode d'opération est particulièrement propice aux optimisations, car il est parallélisable avec du bitslicing. Il est aussi possible de pré-calculer le chiffrement des blocs avant de les appliquer à des données.

4.2 Structure des algorithmes de chiffrement

Un algorithme de chiffrement est constitué d'une fonction de chiffrement f qui est appliquée plusieurs fois. Chaque répétition de la fonction est appelée une *ronde* et elle prend en entrée les données produites par la ronde précédente. On note $f(x)$ la x^{ieme} application de la fonction de chiffrement f. Il est possible d'appliquer l'optimisation CTR sur chaque ronde de l'algorithme de chiffrement.

Application à l'algorithme *AES* L'algorithme *AES* est constitué de 12 rondes de la fonction de chiffrement *aes*.

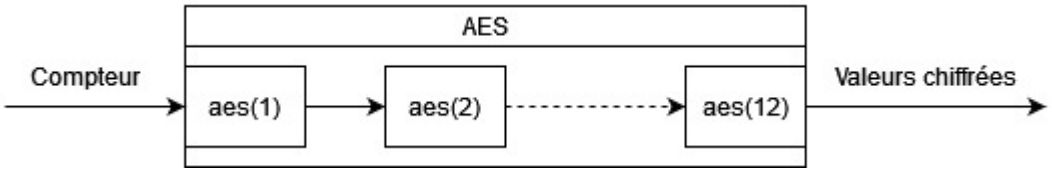

FIGURE 7 – Algorithme *AES* et la répétition de la fonction de chiffrement *aes*

4.3 Mémoïsation des données stables

Mémoïsation : La mémoïsation est une technique d'optimisation qui consiste à mettre en mémoire les résultats d'un calcul dans le but de ne pas refaire le calcul. En pratique, quand une fonction f est utilisé pour la première fois avec un paramètre x, le résultat de $f(x)$ est mis en cache dans une table. La fois suivantes, le résultat est récupéré directement dans la table, sans aucun calcul à refaire.

La fonction de chiffrement f prend n-bits de données en entrée et produit n-bits de données chiffrées en sortie. Du fait de l'utilisation du mode CTR, les bits de poids faibles qui entrent dans la fonction f varient beaucoup plus fréquemment que les bits de poids forts.

Pour l'algorithme *AES*, le compteur fait 128 bits.

Il est possible de séparer les bits en deux ensembles distincts. On note k le bit séparant les deux ensembles. Le premier ensemble est composé des bits *stables* qui s'actualisent au minimum tous les 2^k calculs. En effet, le y^{ieme} bit stable se met à jour tout les 2^{k+y} calculs. Les bits stables sont les bits de poids forts lors de la première application de la fonction f. Le second ensemble contient les bits *instables* qui s'actualisent à chaque calcul et qui sont les bits de poids faible à la première application de la fonction. Le k^{ieme} bit est un bit stable. La portion *stable* et la portion *instable* utilisées en entrée de la ronde $n+1$ sont celles données en sortie de la ronde n.

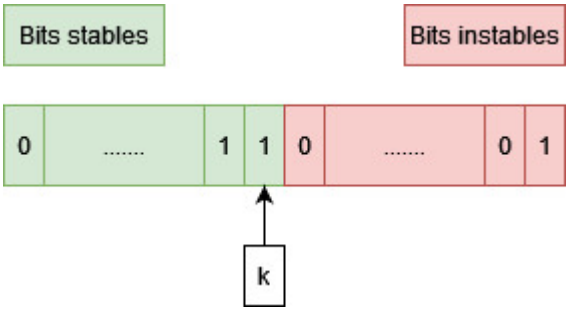

FIGURE 8 – Séparation des bits stables et des bits instables

On cherche à construire deux fonctions f_{Stable} et $f_{Instable}$ avec $f_{Instable}$ la fonction prenant en entrée *la totalité des bits* et f_{Stable} la fonction prenant en entrée les bits *stables*. On souhaite que la combinaison des résultats des fonctions f_{Stable} et $f_{Instable}$ soit équivalant aux résultats de la fonction f. Les fonctions sont typées : $f : n\ bits \rightarrow n\ bits$, $f_{Stable} : (n-k)\ bits \rightarrow a\ bits$ et $f_{Instable} : n\ bits \rightarrow b\ bits$ tel que $n = a+b$.

Si l'on a n-bits *instables*, il est possible de mémoïser les résultats de la fonction f_{Stable} pendant 2^n calculs.

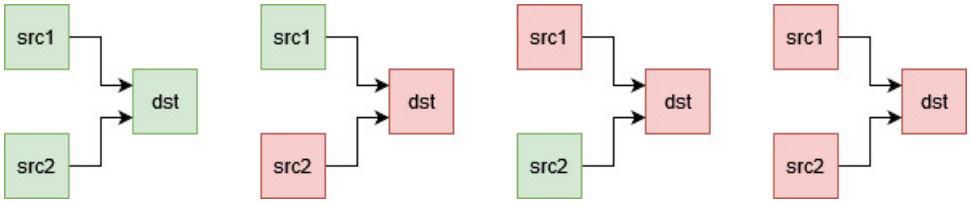

FIGURE 9 – Séparation des bits stables et des bits instables

Le nombre de bits instables est croissant d'une ronde à l'autre car les résultats de l'instruction Oper sont très souvent instables comme on peut le voir sur la Figure 9, de plus l'instruction Move propage la stabilité comme on peut le voir sur la Figure 10 donc il n'existe aucune instruction diminuant le nombre de variables instables.

Cela implique que les premières rondes sont plus à même d'être optimisées car l'optimisation du mode CTR est dépendante du nombre de bits stables.

FIGURE 10 – Séparation des bits stables et des bits instables

4.3.1 Analyse de propagation de l'instabilité

Le premier algorithme à développer est une analyse statique descendante du code écrit dans la représentation intermédiaire. L'analyse reçoit la fonction de chiffrement f ainsi que l'ensemble des bits considérés instable en entrée du programme et donne les ensembles de bits stables et instables qui seront obtenus à la fin de l'exécution de la fonction f.

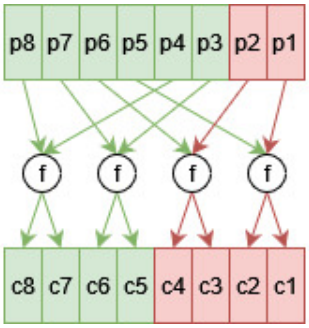

FIGURE 11 – Propagation à travers un programme simpliste

On peut voir sur la Figure 11 que l'instabilité des bits $p1$ et $p2$ s'est propagée sur l'ensemble de bit $\{c1, c2, c3, c4\}$. Il ne reste plus que 4 bits stables $\{c5, c6, c7, c8\}$ dans les bits de sorties.

4.3.2 Analyse de propagation de l'instabilité sur *AES*

On applique l'algorithme d'analyses de propagation de l'instabilité sur *AES*. Cela permet de déterminer les portions stables des différentes rondes de l'algorithme *AES*. Le compteur d'entrée pour *AES* fait 128 bits.

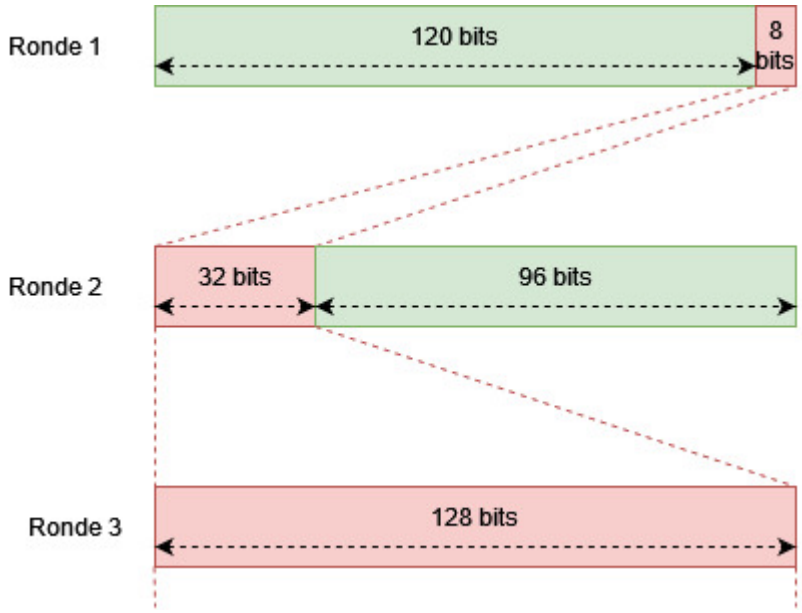

FIGURE 12 – Propagation de l'instabilité dans *AES*

On peut voir sur la Figure 12 qu'avec les 8 bits instables en entrée de la première ronde (les 8 bits de poids faibles du compteur) on obtient avec l'analyse statique que lors de la ronde 2 ce seront les 32 bits de poids fort qui seront instables. On voit aussi que pour l'algorithme *AES* l'ensemble des bits sont instables à partir de la ronde 3. La première ronde est davantage détaillée dans l'annexe 6.2.

4.4 Mémoïsation des données instables

Il est aussi possible de mémoïser les résultats de la fonction $f_{Instable}$. Pour cela, il faut trouver la position du *premier bit stable*, noté i, étant impliquée dans les calculs de la fonction $f_{Instable}$.

Tant que le bit i n'est pas modifié, la fonction $f_{Instable}$ travaille uniquement sur les n bits instables et est de type $f_{Stable} : n\ bits \rightarrow n\ bits$. L'ensemble des 2^n résultats de calculs fournis par la fonction $f_{Instable}$ peut donc être mémoïsé pendant 2^i calculs.

4.4.1 Analyse d'interférence

L'algorithme permettant de calculer la position du bit i est une analyse statique remontante. L'analyse prend la fonction de chiffrement en représentation intermédiaire ainsi que les valeurs de sortie de l'analyse de propagation de l'instabilité. L'analyse détermine l'ensemble des bits ayant un impact sur le calcul de la partie instable de f et renvoie la position du plus petit d'entre eux, le bit i.

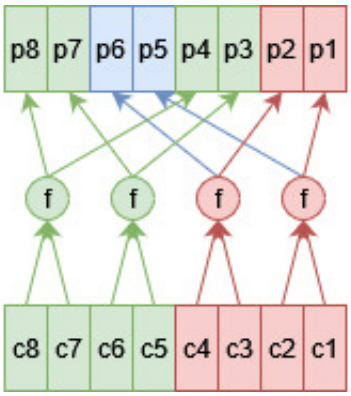

FIGURE 13 – Interférences à travers un programme simpliste

On peut voir sur la Figure 13 que les bits colorés en bleu représentent les bits stables ayant un impact sur le calcul des données instables. Le plus petit d'entre eux est le bit 5.

4.4.2 Analyse d'interférence sur *AES*

On peut appliquer l'analyse d'interférence afin de trouver la position du i^{ieme} bit pour la première ronde d'*AES*. Il est inutile de faire l'analyse pour la ronde 2 car l'ensemble des bits sont instables en ronde 3.

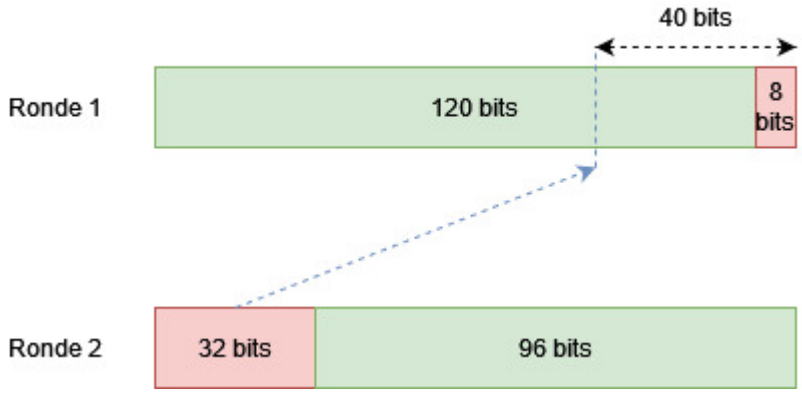

FIGURE 14 – Interférences dans *AES*

On peut voir sur la Figure 14 et plus en détail dans l'annexe 6.3, que le premier bit qui interfère avec les parties *instables* et qui ne fait pas partie des bits *instables* est le 40^{e} bit. On peut donc mémoïser les 256 calculs possibles de la fonction f_{Stable} pendant 2^{40} calculs. Quand le bit 40 sera modifié par l'incrémentation du compteur d'entrée, il faudra refaire la mémoisation des 256 calculs de la fonction $f_{Instable}$. La fonction f_{Stable} prendra en entrée l'ensemble des valeurs possibles pour les bits instables ainsi que les bits stables mis à jour.

4.5 Séparation de la fonction de chiffrement

Il est important de pouvoir créer les fonctions f_{Stable} et $f_{Instable}$ pour un nombre de bit instable n quelconque afin de comparer les différentes configurations.

4.5.1 Génération de f_{Stable} et $f_{Instable}$

Pour cela, il faut développer un algorithme capable, en prenant la fonction f en paramètre, de générer les deux nouvelles fonctions. Il s'agit d'une transformation de programme qui s'applique sur une fonction écrite en représentation intermédiaire et qui prend en paramètre l'ensemble des bits *instables*. L'ensemble des instructions ayant utilisé les données instables sont regroupées dans la fonction f_{Stable}. À l'inverse, tous ceux n'ayant utilisé que des données stables sont regroupées dans la fonction $f_{Instable}$.

On peut voir sur la Figure 15 comment les instructions sont réparties pour former les fonctions f_{Stable} et f_{Stable} avec $n = 2$.

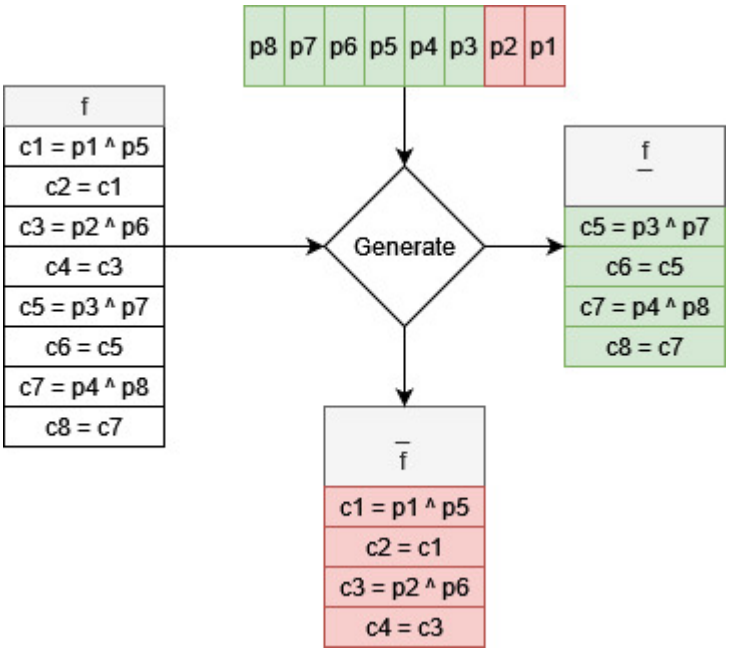

FIGURE 15 – Utilisation de generate

4.5.2 Comptage des portes logiques et des buffers pour *AES*

Afin de pouvoir obtenir une approximation des gains de l'optimisation sur *AES* il est important de connaître le nombre de calculs effectué par chacune des fonctions. On note $|f|$ le nombre de portes logiques de la fonction f. L'algorithme de calcul du nombre de portes donne les résultats suivant :

$\|aes\|$	2560
$\|aes_{1Instable}\|$	192
$\|aes_{1Stable}\|$	2368
$\|aes_{2Instable}\|$	928
$\|aes_{2Stable}\|$	1632

On note *aes* la fonction de chiffrement de *AES* et aes_x la fonction de chiffrement aes à la ronde x.

4.6 Modèle analytique

Afin de trouver le nombre de bits stables n donnant les meilleures optimisations, il faut pour chaque algorithme de chiffrement comparer toutes les valeurs de n possibles. Pour cela, j'ai calculé une approximation du temps de calcul et j'ai choisi de compter le nombre d'opérations (Oper) nécessaires au calcul de l'algorithme de chiffrement. Il faut aussi récupérer la taille maximale des buffers servant à la mémoïsation. Les résultats pour chaque algorithme sont donnés en section 4.8.1.

4.7 Validation CTR

Il est important de pouvoir garantir la validité des différentes transformations appliquées à l'algorithme de chiffrement, afin de valider que $f_{Instable}$ et f_{Stable} est bien équivalent à la fonction f. Pour cela, il est nécessaire de pouvoir comparer deux programmes écrits dans la représentation intermédiaire.

J'ai donc mis en place une traduction permettant de générer automatiquement des vérifications pour les différentes étapes de la compilation que j'ai programmé. L'algorithme prend en entrée deux programmes écrits dans la représentation intermédiaire et génère un problème de *satisfiabilité modulo des théories* [17] (SMT).

Le problème SMT est décrit en Python grâce au module Boolector [18]. Chaque instruction de la représentation intermédiaire est traduite vers son équivalent dans l'ensemble d'opération Booléenne que Boolector mets à disposition. Boolector fait appel à un solveur SMT spécialisé pour les problèmes booléens [19]. Le solveur SMT détermine s'il existe une configuration dans laquelle les deux programmes ne sont pas équivalents. Si l'équation est non satisfiable, l'algorithme de transformation de programme donne bien deux programmes équivalents car il n'y a aucune configuration dans laquelle les résultats de f sont différents des résultats de $f_{Instable}$ et f_{Stable}.

Le problème SMT compare la représentation intermédiaire après parsing avec celles données par les deux processus de compilation comme illustré dans la Figure 16.

La vérification était une étape cruciale de mon travail et m'a permis de trouver de nombreuses erreurs qui n'avaient pas été détectées lors de précédents test unitaires.

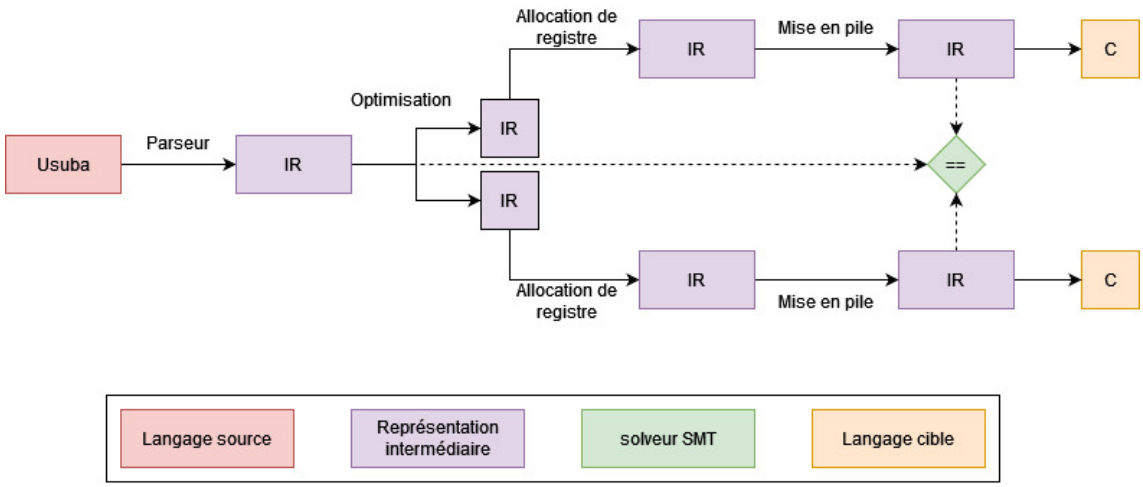

FIGURE 16 – Validation sur l'ensemble de la transformation du programme

4.8 Évaluation

J'ai calculer les gains théoriques pour différents algorithmes de chiffrement symétrique (*AES*, *Gift*, *Photon*, *Present*, *Xoodoo*, *Rectangle*). J'ai utilisé ces algorithmes car ils étaient disponibles sur Usuba.

algorithme	nombre maximum de rondes optimisés	gain théorique
AES	2	16.3%
Gift	4	7.75%
Photon	2	9.35%
Present	3	6.78%
Xoodoo	3	9.33%
Rectangle	4	7.69%

4.8.1 Résultats sur l'algorithme *AES*

AES est le seul algorithme de chifferment symétrique sur lequel il est possible de comparer les résultats de l'optimisation car les performances sont connues dans le papier FACE. Le gain maximal de performance estimé est de 16.3% pour la configuration de 8 bits instables, ce qui correspond à ce que l'on peut lire dans le papier FACE. De plus, on observe des paliers tous les 8 bits sur la Figure 17 ce qui est un résultat logique compte tenu du fonctionnement de la fonction de chiffrement de *AES*. En effet, la fonction de chiffrement combine les bits par paquet de 8 bits les rendant tous instable, cela explique qu'il n'y a pas de différence d'optimisation entre 9 et 16 bits instables.

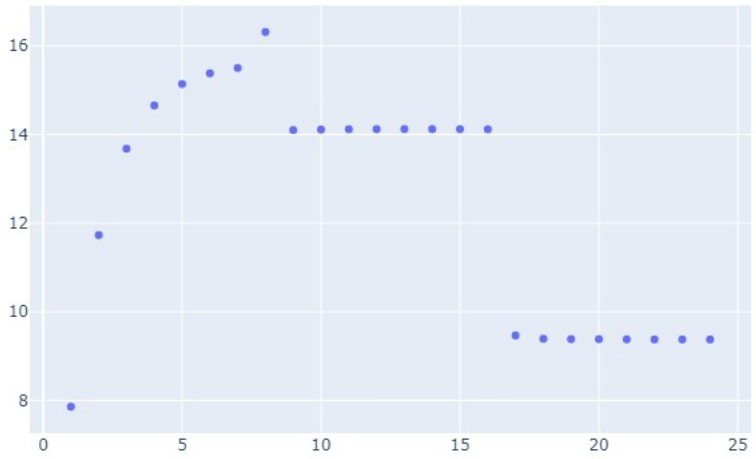

FIGURE 17 – Résultat pour *AES*

J'ai également mis en place un benchmark pour la configuration 8 bits instables de *AES* donnant un gain de performance de 11.7% entre *AES* et *AES* avec l'optimisation CTR. L'implémentation en C de *AES* et la version d'*AES* optimisé que j'ai faites pour le benchmark sont la cause des différences de performance avec le papier FACE qui donne une optimisation de 20%.

4.8.2 Résultats sur l'algorithme $Gift$

Les résultats sur l'algorithme de chiffrement $Gift$ donne un gain maximal de performance de 7.75%. Ce gain maximum est obtenu pour la configuration de 9 bits instables. Les optimisations CTR sont appliquées sur 3 à 4 rondes de l'algorithme $Gift$, cela s'explique par un plus grand nombre de rondes (40 rondes).

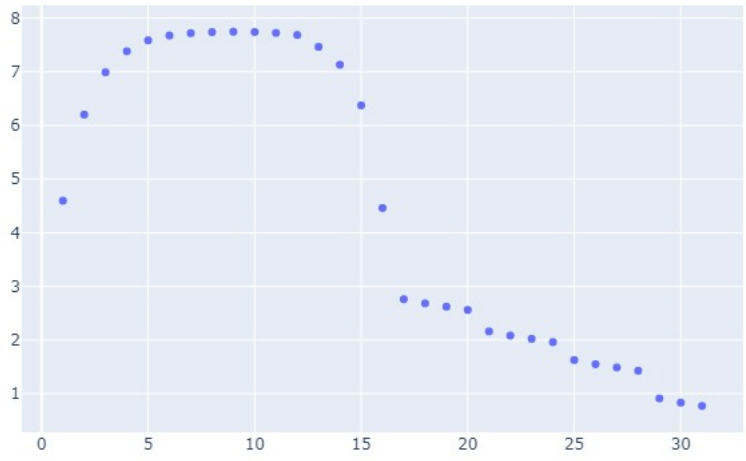

FIGURE 18 – Résultat pour $Gift$

4.8.3 Résultats sur l'algorithme *Photon*

Pour l'algorithme *Photon*, on peut observer une périodicité pour les gains de performances tous les 4 bits. Avec de plus amples analyses, on s'aperçoit que la périodicité est corrélée avec la deuxième partie de l'optimisation CTR (la mémoisation des données instables). La meilleure configuration est située à 8 bits instables avec un gain de performance de 9.35%. *Photon* est optimisé sur le même nombre de rondes que *AES*.

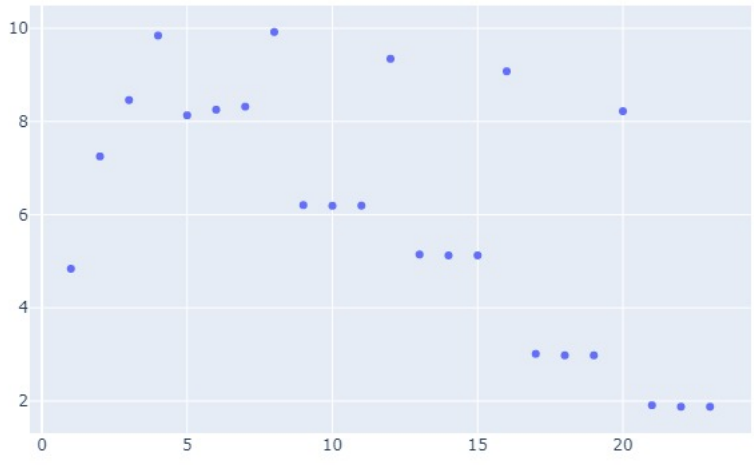

FIGURE 19 – Résultat pour *Photon*

4.8.4 Résultats sur l'algorithme *Rectangle*

L'algorithme de chiffrement *Rectangle* possède 25 rondes et l'optimisation CTR s'applique au maximum sur 4 rondes. Le meilleur gain est donné pour 5 bits instables, cela correspond au plus grand buffer de donnée instable pour une optimisation de 4 rondes. En effet, pour 6 bits instables et plus l'optimisation s'applique pour moins de 4 rondes. Le gain de performance maximum est de 7.69% pour *Rectangle*.

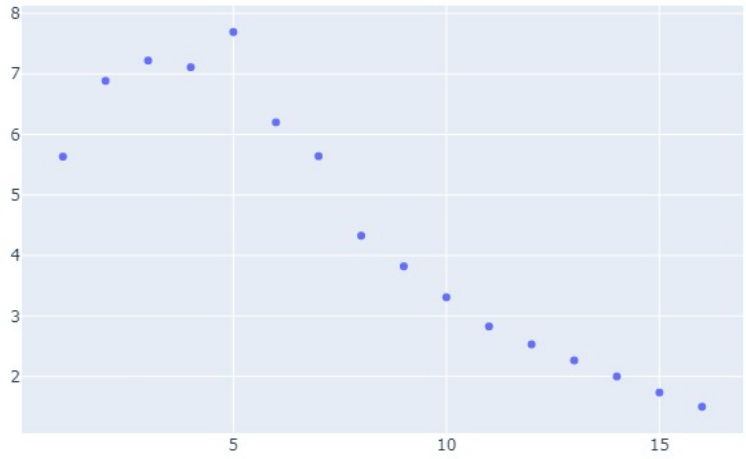

FIGURE 20 – Résultat pour *Rectangle*

4.8.5 Résultats sur l'algorithme *Present*

Comme on peut le voir sur la Figure 21, le maximum de gain pour l'algorithme de chiffrement *Present* est situé à 4 bits instables. Ce gain est de 6.78%. Cela s'explique par le fait que l'optimisation CTR s'applique sur seulement 3 des 31 rondes de *Present*.

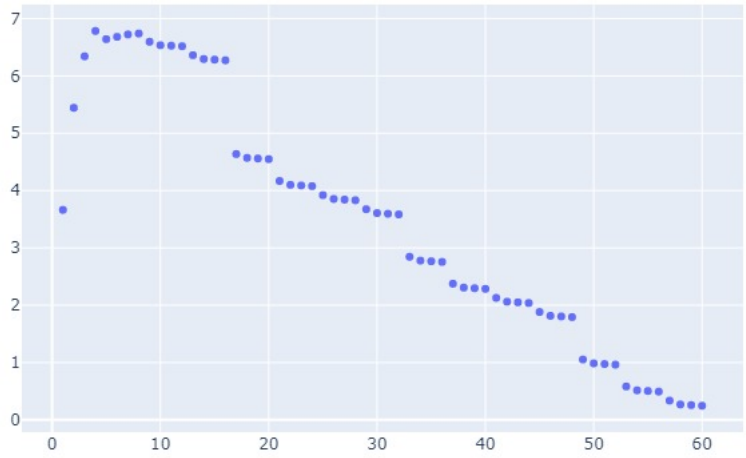

FIGURE 21 – Résultat pour *Present*

4.8.6 Résultats sur l'algorithme *Xoodoo*

Le gain maximal sur l'algorithme *Xoodoo* est de 9.33% avec seulement 3 bits instables. Cela s'explique par le manque de données pouvant être mémoïsé par la première partie de l'optimisation CTR (la mémoisation des données stables).

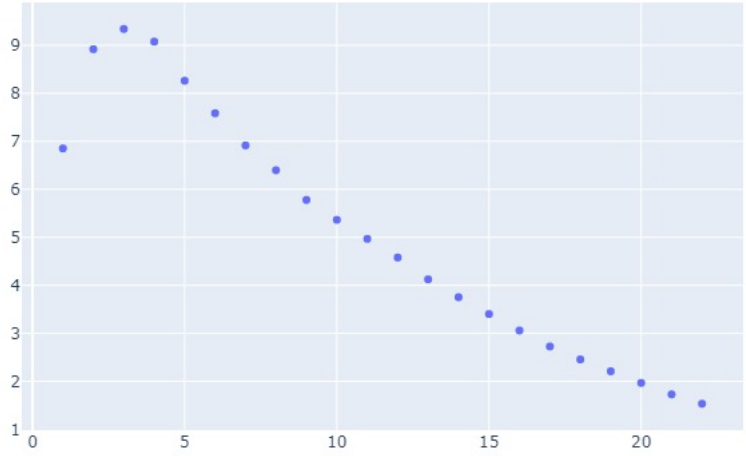

FIGURE 22 – Résultat pour *Xoodoo*

4.9 Conclusion CTR

Pendant ce stage, j'ai pu étudier et implémenter une optimisation spécifique aux algorithmes de chiffrement symétrique utilisant le mode de chiffrement CTR. Les différentes analyses que j'ai développées ont permis d'évaluer des gains de performance de l'optimisation sur plusieurs algorithmes. La méthode mise en place et les différentes analyses sont génériques et fonctionnelles pour de futurs algorithmes de chiffrement.

J'ai beaucoup apprécié de travailler sur cette optimisation, pour laquelle il fallait analyser un article scientifique, comprendre le fonctionnement et chercher à le généraliser. C'était très motivant de pouvoir obtenir des résultats de recherche sur des algorithmes qui n'avaient pas encore été testés.

5 Bilan personnel

Ce stage a été l'occasion pour moi de découvrir le milieu de la recherche. J'ai acquis de l'expérience pour lire et analyser des articles scientifiques ainsi que pour appréhender de nouveaux concepts. J'ai pu mettre en pratique les enseignements de ma formation sur la compilation des langages. J'ai pu employer différentes méthodes pour évaluer avec rigueur les résultats de mes travaux.

J'ai assisté à différents séminaires de recherche qui m'ont permis de développer mes compétences sur des sujets techniques et variés. J'ai aussi pu travailler dans une équipe et bénéficier des compétences de chacun.

Références

[1] “Usubac.” [Online]. Available : https://github.com/usubalang/usuba

[2] “Usuba, optimizing bitslicing compiler.” [Online]. Available : https://usubalang.github.io/usuba/

[3] D. Mercadier, “Bitslicing,” 2020. [Online]. Available : https://usubalang.github.io/usuba/2020/01/14/bitslicing.html

[4] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, “Jasmin : High-assurance and high-speed cryptography,” in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS ’17. New York, NY, USA : Association for Computing Machinery, 2017, p. 1807–1823. [Online]. Available : https://doi.org/10.1145/3133956.3134078

[5] “Jasminc.” [Online]. Available : https://github.com/jasmin-lang/jasmin

[6] The Coq development team, The Coq proof assistant reference manual, 2004, Version 8.0. [Online]. Available : /brokenurl#{http://coq.inria.fr/}

[7] “Exemple de code simplifié.” [Online]. Available : http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/slp_630.txt

[8] “Autres circuits.” [Online]. Available : http://cs-www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html

[9] R. C. Lozano and C. Schulte, “Survey on combinatorial register allocation and instruction scheduling,” 2014. [Online]. Available : https://arxiv.org/abs/1409.7628

[10] A. W. Appel, Modern Compiler Implementation in ML, new ed ed. Cambridge University Press, July 2004.

[11] Google, “Benchmark.” [Online]. Available : https://github.com/google/benchmark

[12] Wikipédia, “Advanced encryption standard — wikipédia, l’encyclopédie libre,” 2021, [En ligne ; Page disponible le 12-avril-2021]. [Online]. Available : http://fr.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard

[13] J. H. Park and D. H. Lee, “Face : Fast aes ctr mode encryption techniques based on the reuse of repetitive data,” IACR Transactions on Cryptographic Hardware and Embedded Systems, vol. 2018, no. 3, p. 469–499, Aug. 2018. [Online]. Available : https://tches.iacr.org/index.php/TCHES/article/view/7283

[14] D. Mercadier, “Usuba, optimizing bitslicing compiler,” 2020. [Online]. Available : https://dariusmercadier.com/assets/documents/thesis.pdf

[15] Wikipedia contributors, “Block cipher mode of operation — Wikipedia, the free encyclopedia,” 2022. [Online]. Available : https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation

[16] Wikipédia, “Mode d’opération (cryptographie) — wikipédia, l’encyclopédie libre,” 2022. [Online]. Available : http://fr.wikipedia.org/w/index.php?title=Mode_d%27op%C3%A9ration_(cryptographie)

[17] ——, “Satisfiability modulo theories — wikipédia, l’encyclopédie libre,” 2017, [En ligne ; Page disponible le 10-octobre-2017]. [Online]. Available : http://fr.wikipedia.org/w/index.php?title=Satisfiability_modulo_theories

[18] A. Niemetz, M. Preiner, and A. Biere, “Boolector 2.0,” J. Satisf. Boolean Model. Comput., vol. 9, no. 1, pp. 53–58, 2014. [Online]. Available : https://doi.org/10.3233/sat190101

[19] Wikipedia contributors, “Boolean satisfiability problem — Wikipedia, the free encyclopedia,” 2022. [Online]. Available : https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem

6 Annexe

6.1 Ast du langage intermédiaire

```
type _var = string
type _val = int

type src =
    | IO of _var * string * int
    | Var of _var
    | Val of _val

type dst =
    | DVar of _var
    | DIO of _var * string * int

type inst =
    | Oper of op * dst * src * src
    | Move of dst * src
```

6.2 Propagations *AES*

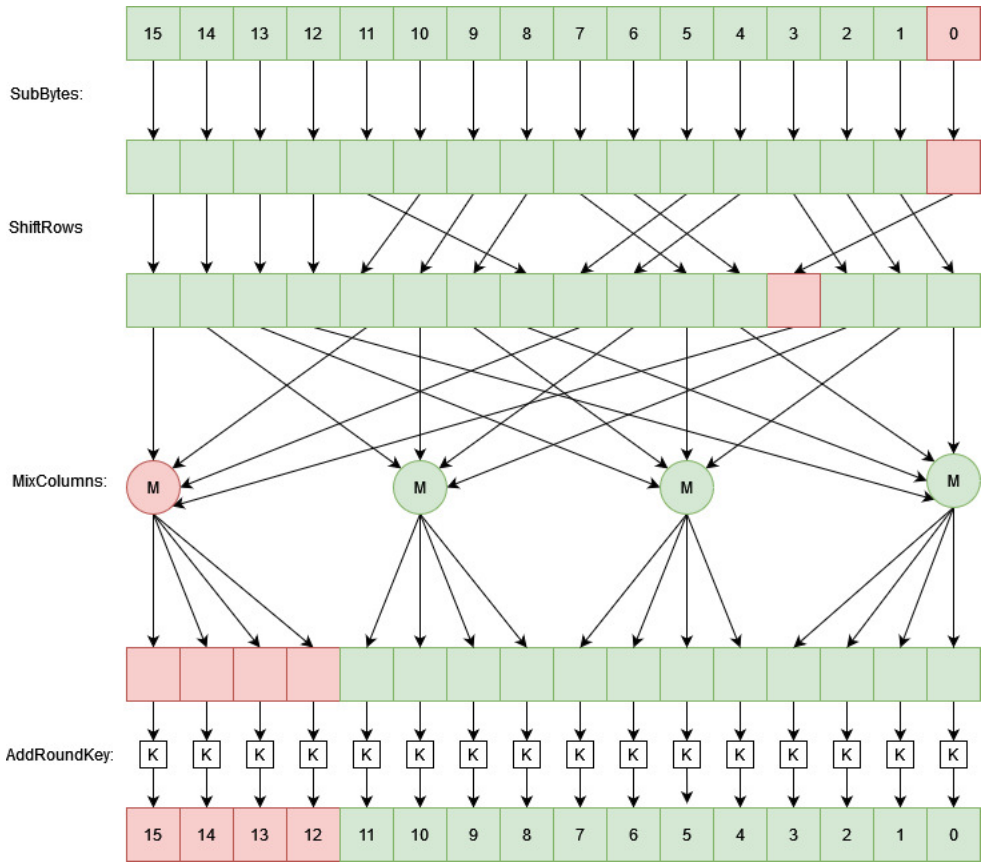

FIGURE 23 – Propagation de l'instabilité dans *AES*, découpage par octet

6.3 Interférences *AES*

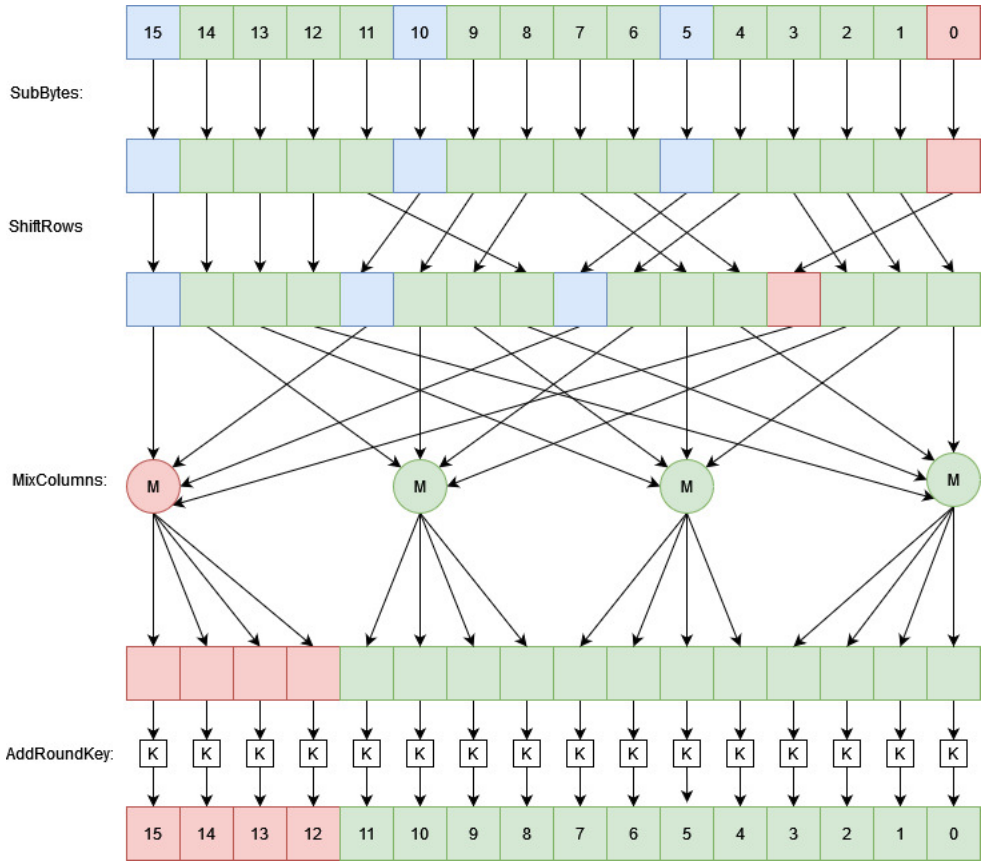

FIGURE 24 – Interférences dans *AES*, découpage par octet