Implantations et protections de mécanismes cryptographiques logiciels et matériels

Marie-Angela Cornelie

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Mathématiques**

Arrêté ministérial : 7 Août 2006

Présentée par

Marie-Angela CORNELIE

Thèse dirigée par **Philippe ELBAZ-VINCENT**

préparée au sein du Laboratoire **Institut Fourier**
dans l'École Doctorale **Ecole Doctorale MSTII**

Implantations et protections de mécanismes cryptographiques logiciels et matériels

Thèse soutenue publiquement le **12 avril 2016**,
devant le jury composé de :

M, Jean-Claude BAJARD
Professeur à l'Université Pierre et Marie Curie, Paris, Président
M, Philippe ELBAZ-VINCENT
Professeur à l'Université Grenoble Alpes, Grenoble, Directeur de thèse
M, Louis GOUBIN
Professeur à l'Université Versailles-Saint-Quentin-en-Yvelines, Versailles, Rapporteur
M, Marc JOYE
Expert cryptologie chez Technicolor, Habilité à Diriger des Recherches, Palo Alto USA, Rapporteur
M, Régis LEVEUGLE
Professeur à Grenoble INP, Grenoble, Examinateur
Mme, Marie-Laure POTET
Professeur à Grenoble INP, Grenoble, Examinatrice
M, Thomas SIRVENT
Ingénieur DGA MI, Bruz, Examinateur
M, Arnaud TISSERAND
Directeur de recherche au CNRS, IRISA, Rennes, Examinateur

Remerciements

Bôd lanmè pa lwen...
le bout du tunnel n'est pas loin...

La thèse peut parfois ressembler à un tunnel sans fin mais comme le dit ce proverbe antillais, il faut garder espoir car on finit en général par en sortir.

J'adresse tout d'abord mes remerciements à Louis Goubin et à Marc Joye pour avoir accepté de rapporter ma thèse et pour tous leurs conseils et remarques qui m'ont permis d'avoir un regard neuf sur mon travail et de mieux appréhender ce domaine de recherche. J'y associe également les autres membres du jury qui ont accepté de faire partie de cette aventure. Mes travaux de thèse ont été partiellement financés par la DGA-MI et le Labex Persyval-Lab (ANR-11-LABX-0025).

Tout au long de ces années, j'ai pu compter sur le soutien de Philippe qui, depuis mon master, a su me conseiller et m'aider malgré les quelques embûches que nous avons rencontrées. Je le remercie pour la confiance qu'il m'a accordée et pour toutes ces discussions aux envolées surprenantes. J'ai également une pensée pour Simon, Paolo, Régis et Cyril avec qui j'ai eu le plaisir de pouvoir travailler.

Je tiens également à mentionner toutes les personnes que j'ai eu la chance de côtoyer dans le bureau 34C : Le grimpeur de l'extrême, Huguette la spécialiste du `main` qui s'appelle `main`, celui qui aimait les panoramas et celle pour qui ça ne recule jamais. La bonne humeur et l'entraide ont toujours été au rendez-vous grâce à une ambiance toujours conviviale et à nos petits moments au Facto'. Je n'oublie pas le personnel administratifs de l'Institut Fourier, le service informatique et Anne-Laure qui sont toujours présents pour nous aider dans la machinerie administrative qui peut s'avérer très complexe.

Je sais que tout cela n'aurait pas été possible sans le soutien de ma famille et en particulier celui de ma mère qui m'a toujours poussée à me dépasser. Même à distance, ses recommandations étaient toujours de rigueur pour me permettre d'avancer dans mes études supérieures. Je remercie également Marine qui m'a accompagnée tout au long de cette belle aventure avec ses hauts et ses bas, ses nuits blanches, ses moments d'euphorie quand le code compilait. Elle a su me redonner du courage quand il le fallait avec ses bons petits plats et son humour. Grâce à elles, j'ai toujours su que le bout du tunnel était proche.

Table des matières

Introduction générale

Le développement des systèmes d'information a permis l'essor de la cryptographie moderne. La question de la sécurisation des données est devenue un enjeu crucial, en particulier depuis la diversification des supports de données. Les ordinateurs, les cartes à puces ne sont plus seulement les cibles à protéger car les smartphones contiennent de plus en plus de données sensibles.

Les mécanismes cryptographiques répondent bien à la problématique de sécurisation des données. Cependant, **comment peut-on sécuriser ces algorithmes et empêcher qu'ils ne soient corrompus au point qu'ils ne remplissent plus leur fonction de protection ?** Pour répondre à cette question, il faut s'intéresser aux divers domaines liés à la protection logicielle et matérielle.

Protection logicielle

La protection peut se faire à l'aide de mesures légales et techniques. La protection légale regroupe l'ensemble des droits d'auteur, brevets et licences que l'on peut attacher au logiciel.

Les droits d'auteur permettent à un auteur ou à son cessionnaire de protéger l'exploitation, pendant un certain nombre d'années, d'une œuvre littéraire, artistique ou scientifique. En cas de litiges, une preuve d'antériorité doit être apportée. Avec la rapidité de développement des logiciels, il est compliqué de fournir une telle preuve.

Les brevets protègent à la fois les inventions techniques et les fonctionnalités liées à un logiciel. Les entreprises, quelle que soit leur taille, utilisent ce moyen afin de se prémunir de leurs concurrents. Déposer un brevet leur permet de prendre le contrôle juridique et commercial d'une invention, et d'en tirer un profit optimal. C'est une solution qui protège effectivement le logiciel mais elle est très onéreuse.

Une licence de logiciel est un contrat par lequel le titulaire des droits d'auteur sur un logiciel définit avec l'utilisateur ou l'exploitant les conditions dans lesquelles il peut être utilisé, diffusé ou modifié. Il peut notamment interdire l'analyse ou la rétro-conception du programme. Cependant, en pratique il n'est pas aisé de vérifier que les termes de la licence soient bien respectés.

En général, les moyens légaux sont très coûteux et ne protègent pas contre toutes les attaques possibles. Une alternative consiste à utiliser des techniques de protection qui agissent directement au niveau du logiciel.

Exécution coté serveur L'utilisateur se connecte sur le serveur ou le site du développeur pour exécuter le programme. Ainsi un attaquant ne peut pas avoir d'accès physique au logiciel à moins de pirater le serveur. Cependant, les performances du logiciel sont impactées. Une solution partielle consiste à diviser le programme en deux parties : la partie que l'on souhaite protéger reste sur le serveur alors que l'autre partie du code est récupérable par l'utilisateur. L'exécution normale du logiciel ne peut se faire que si les deux partie sont exécutées.

Utilisation de code natif Quand l'utilisateur télécharge le logiciel, il s'identifie avec son architecture, et la version de code natif correspondante lui est transmise. Les signatures digitales assurent l'authenticité et la non-altération du logiciel. Cependant un attaquant peut malgré tout décompiler le code natif même si l'analyse est plus ardue.

Chiffrement Le chiffrement de l'application n'est une protection efficace que si l'intégralité du processus de dé-chiffrement/exécution se déroule en matériel. Si une machine virtuelle est utilisée, il sera toujours possible d'intercepter le code après déchiffrement et de l'analyser.

Obscurcissement Avant distribution, le logiciel est obscurci en utilisant un outil automatique qui donne en sortie un nouveau avec des fonctionnalités identiques mais qui rend la rétro-conception plus difficile.

Protection matérielle

Lorsqu'un mécanisme cryptographique est implanté sur une cible matérielle, que ce soit une carte à puce ou un FGPA il faut prendre en compte son environnement d'exécu-

tion afin de définir les possibilités d'attaques et les protections nécessaires.

Les attaques peuvent être réalisées selon deux méthodes :

- **Attaque passive** : Une observation des différentes propriétés physiques, comme la consommation de courant, permet de récupérer de l'information sur le fonctionnement du composant sans pour autant le perturber.
- **Attaque active** : Un comportement anormal est provoqué par l'utilisation d'une perturbation. On peut par exemple, faire varier la température ou utiliser un laser. De cette manière, il est possible de retrouver des implantations d'algorithmes ou des données stockées en mémoire. On obtient ainsi une image de circuit qu'on peut utiliser afin de le rétro-concevoir. En général, ces attaques sont destructives et le composant n'est donc plus utilisable par la suite. Cependant, si un attaquant ou une attaquante a sa disposition plusieurs exemplaires d'un même composant, il peut réaliser une attaque active sur un exemplaire afin de récupérer par exemple un algorithme, puis utiliser les autres afin de réaliser une attaque passive vérifiant les résultats obtenus.

Il paraît très complexe de se prémunir face aux attaques actives. Dans certaines situations, il est possible d'implanter un détecteur de changement de température ou d'une autre perturbation, qui provoquera l'arrêt du composant en cas d'attaque. Une détection d'injection de fautes peut également être ajoutée au composant.

Afin de se protéger des attaques passives, il est nécessaire de comprendre les fuites d'information que peut provoquer l'exécution d'un algorithme sur un composant. Par exemple, il est possible de privilégier des opérations en temps constant qui ne dépendent pas d'une valeur secrète.

Dans ce mémoire, nous allons étudier les problématiques introduites précédemment en deux parties. La première partie traitera des implantations logicielle et matérielle d'un crypto-processeur basé sur les courbes elliptiques. La seconde présentera nos travaux concernant l'obscurcissement de code source.

Acronymes

AES	Advanced Encryption Standard.
ANSI	American National Standards Institute.
BDM	Block Decomposition Method.
BRAM	Block Random Access Memory.
CC	Cyclomatic Complexity.
CCC	Coupled Cyclomatic Complexity.
CFG	Control Flow Graph.
DLP	Discrete Logarithm Problem.
DRBG	Deterministic Random Bit Generator.
DRM	Digital Rights Management.
DSA	Digital Signature Algorithm.
DSP48E	Digital Signal Processing Element.
ECC	Elliptic Curve Cryptography.
ECDSA	Elliptic Curve Digital Signature Algorithm.
FIPS	Federal Information Processing Standards Publications.
FPGA	Field Programmable Gate Arrays.
IEEE	Institute of Electrical and Electronics Engineers.
ISO	International Organization of Standardization.
NIST	National Institute of Standards and Technology.
PCP	Post Correspondence Problem.
RNS	Residue Number System.

SCA Side Channel Attacks.
SPA Simple Power Analysis.

TCC Total Cyclomatic Complexity.

Première partie

Implantation d'un crypto-processeur basé sur les courbes elliptiques

Table des matières

Notations

K	Un corps fini.
$\overline{K}$	Une clôture algébrique de K.
$K^\times$	Le groupe multiplicatif associé à K.
$K[X]$	Anneau des polynômes à coefficients dans K.
$\mathbb{F}_p$	Corps fini à p éléments.
$\mathbb{Z}/N\mathbb{Z}$	Pour $N \in \mathbb{Z}$, anneau quotient des résidus modulo N.
$(u_{n-1}, u_{n-2}, \ldots, u_0)_b$	La représentation en base b d'un entier u de n b-mots.
E/K	Une courbe elliptique E définie sur un corps fini K.
$E(L)$	Pour $K \subset L \subset \overline{K}$, ensemble des points L-rationnels de la courbe E définie sur K.
$E[m]$	Ensemble des points de m-torsion de la courbe E.
$\Delta_{E/K}$	Discriminant d'une courbe elliptique E définie sur un corps fini K.
$j_{E/K}$	j-invariant d'une courbe elliptique E définie sur un corps fini K.
$E_{W,a,b}/K$	Une courbe elliptique définie par une équation réduite de Weierstrass sur un corps fini K.

$E_{J,a,b}/K$	Une courbe quartique de Jacobi étendue définie sur un corps fini K.
$E_{E,a,b}/K$	Une courbe d'Edwards définie sur un corps fini K.

Chapitre 1

Introduction

La cryptographie basée sur les courbes elliptiques a été introduite indépendamment par Koblitz [Kob87] et Miller [Mil86] en 1985. Il s'agissait de concevoir un crypto-système asymétrique basée sur des courbes elliptiques définies sur des corps finis. Le problème mathématique qui a motivé leur utilisation est leur résistance au problème du logarithme discret.

1.1 Problème du logarithme discret (DLP)

Avant de définir le problème du logarithme discret, définissons en premier lieu le contexte. Soient G un groupe, $g \in G$, et $h \in \langle g \rangle$ tels qu'il existe $n \in \mathbb{N}$ pour lequel $h = g^n$. On dit qu'un entier $a \in \mathbb{N}$ est un logarithme discret de h relativement à g si $h = g^a$. Le problème du logarithme discret consiste à retrouver a en ayant juste la connaissance de h et de g. La complexité du problème dépend du groupe considéré.

Exemple 1.1.1
Problème du logarithme discret sur divers groupes.

1. $G = (\mathbb{Z}/N\mathbb{Z})^+$ muni de la loi de groupe additive. Le DLP est trivial car une simple division permet de retrouver le logarithme discret.
2. $G = (\mathbb{Z}/N\mathbb{Z})^\times$. Si N est suffisamment grand et que $g \in G$ est d'ordre p (grand également) alors la résolution du DLP est difficile.
3. $G = \mathbb{F}_q^\times$. Si q est suffisamment grand alors il est difficile de résoudre le DLP.

Dans le cas du groupe de points d'une courbe elliptique, le problème est également difficile. La meilleure attaque est en $O(\sqrt{n})$ où n est l'ordre du générateur. Cela justifie

par conséquent leur usage en cryptographie. Partant de ce constat, différents protocoles ont été adaptés aux courbes elliptiques car en plus de leur résistance au DLP, les tailles de clés utilisées sont moindres pour un même niveau de sécurité que celles dans le cas du problème de factorisation (RSA [RSA78]).

bits de sécurité	RSA	ECC
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

Tableau 1.1 – Comparaison des tailles de clés entre ECC et RSA [SP800-57].

1.2 ECDSA

Le protocole ECDSA, Elliptic Curve Digital Signature Algorithm, est une variante de DSA, Digital Signature Algorithm, pour les courbes elliptiques. Il a été proposé en 1992 par Vanstone [Van92] en réponse à la demande du NIST de propositions pour un système de signatures digitales. C'est devenu un standard ISO en 1998 [ISO-14888-3], puis un standard de l'ANSI [X9.62] en 1999. En 2000, il est également accepté comme standard IEEE [IEEE-P1363.3] et du FIPS [FIPS-186-4].

Domaine de paramètres

Le domaine de paramètres $\mathcal{D}$ pour ECDSA est de la forme

$$(q, FR, a, b\{, domain\_parameter\_seed\}, G, n, h),$$

où q est la taille du corps de base ; FR indique le type de base utilisée ; a et b sont deux éléments du corps qui définissent l'équation de la courbe ; $domain\_parameter\_seed$ est la graine utilisée pour définir $\mathcal{D}$, c'est un paramètre optionnel qui est utilisé quand la courbe a été générée de manière aléatoire et que cela peut être vérifié ; G est un point de base de la courbe d'ordre premier n et h est son co-facteur. Il faut également spécifier la fonction de hachage H qui devra être utilisée.

Génération de clés

Une paire de clés pour ECDSA consiste en une clé privée d et une clé publique Q associées au domaine $\mathcal{D}$. La donnée d est prévue pour une période de temps donnée alors que Q peut être utilisée tant que des signatures générées avec d doivent être vérifiées. Ces clés ne doivent être utilisées que pour le protocole ECDSA.

Signature

Supposons qu'Alice ait défini un domaine de paramètres $\mathcal{D}$ et qu'elle ait généré sa paire de clé (d, Q). Elle souhaite signer un message M qu'elle veut envoyer à Bob. La procédure de signature est la suivante.

1. Choisir un entier aléatoire k tel que $1 \leqslant k \leqslant n-1$.
2. Calculer $kG = (x_1, y_1)$ et $r = x_1 \bmod n$. Si $r = 0$ alors retourner à l'étape 1.
3. Calculer $k^{-1} \bmod n$.
4. Calculer $e = H(M)$.
5. Calculer $s = k^{-1}(e + dr) \bmod n$. Si $s = 0$ alors retourner à l'étape 1.

La signature du message M est alors la paire (r, s).

Vérification de la signature

De son côté, pour vérifier la signature Bob doit récupérer le domaine $\mathcal{D}$ ainsi que la clé publique d'Alice. Puis, il utilise la procédure suivante.

1. Vérifier que r et s sont des entiers dans $[1, n-1]$.
2. Calculer $e = H(M)$.
3. Calculer $w = s^{-1} \bmod n$.
4. Calculer $u_1 = ew \bmod n$ et $u_2 = rw \bmod n$.
5. Calculer le point $X = u_1 G + u_2 Q$. Si $X = \mathcal{O}$, où $\mathcal{O}$ est l'élément neutre, alors rejeter la signature. Sinon calculer $v = x_1 \bmod n$ où $X = (x_1, y_1)$.
6. Accepter la signature si et seulement si $v = r$.

Si la signature a bien été générée par Alice alors, l'équation suivante est vérifiée :

$$k \equiv s^{-1}(e + dr) \equiv s^{-1}e + s^{-1}dr \equiv u_1 + u_2 d (\bmod n)$$

Par conséquent $X = (u_1 + u_2 d)G = kG$ et $v = r$.

Parties critiques

On peut identifier différentes parties critiques dans ce protocole que ce soit au niveau des performances ou de la sûreté. On peut par exemple citer la génération de nombre aléatoire qui a été sujette à des attaques du protocole [fai10]. Si l'on se concentre sur les étapes faisant intervenir les courbes elliptiques, les points sensibles sont :

- la courbe utilisée et le point de base,
- la multiplication scalaire kG qui fait intervenir un nonce au moment de la signature.

Du point de vue des performances, cette opération doit également être implantée de manière efficace. Ce protocole illustre ainsi les problématiques liées à une bonne utilisation des courbes elliptiques à savoir la génération de courbes aléatoires et une multiplication scalaire rapide mais sûre.

1.3 Objectifs et contributions

Les travaux de cette partie s'inscrivent dans une collaboration avec le laboratoire TIMA dans le cadre du LabEx PERSYVAL-Lab [1] et de l'axe de recherche « Design and Analysis of Cryptographic Components » (DACC) de l'équipe-action SCCyPhy [2]. Notre objectif est d'apporter des solutions logicielle et matérielle permettant d'obtenir une implantation des courbes elliptiques sûre et efficace.

Les principales contributions de nos travaux sont les suivantes :

- Étude et implantation d'une bibliothèque logicielle dédiée à la génération aléatoire de courbes elliptiques sous plusieurs formes.
- Développement d'une bibliothèque efficace pour le traitement de courbes elliptiques données sous différentes formes. Nous avons mis en place une architecture modulaire qui permet de brancher d'autres bibliothèques. Nos résultats de performance nous placent en bonne position parmi les implantations supportant les courbes. De plus, il est aisé d'étendre les fonctionnalités en ajoutant d'autres formes de courbes.
- Implantation matérielle d'une arithmétique modulaire rapide sur FPGA. Notre design exploite la forte parallélisation des opérations afin d'avoir une exécution performante.

1. ANR-11-LABX-0025
2. Security and Cryptology for CyberPhysical systems

1.4 Plan de cette partie

Cette partie est structurée en six chapitres. Le premier chapitre introduira les courbes elliptiques permettant d'obtenir des formules d'addition unifiées. Après un rappel des propriétés classiques des courbes, nous verrons en quoi les formules unifiées constituent une contre-mesure face aux attaques par canaux cachés. Nous présenterons en détail deux formes de courbes qui présentent une telle propriété : les courbes quartiques de Jacobi étendues et celles d'Edwards. De plus, elles ont également une arithmétique plus rapide que les courbes classiques données sous forme de Weierstrass.

Le deuxième chapitre traitera de la problématique de génération aléatoire de courbes elliptiques. Nous rappellerons en premier lieu les attaques mathématiques que l'on trouve dans la littératures ainsi que les contre-mesures associées. Nous discuterons de l'état de l'art des outils de génération. Puis, nous adapterons la méthodologie Brainpool au support des courbes quartiques de Jacobi étendues et Edwards. Enfin, nous présenterons l'implantation qui a été réalisée.

Le troisième chapitre analysera l'arithmétique sur les corps finis $\mathbb{F}_p$ ainsi que ses extensions qui permettent d'obtenir une implantation des courbes elliptiques efficace. En fonction des contraintes de notre contexte, nous déciderons quels algorithmes sont les plus adaptés. Concernant les extensions de corps, nous présenterons la méthodologie des tours d'extensions et nous détaillerons les algorithmes possibles ainsi que leurs versions parallélisées pour les extensions quadratiques et cubiques.

Le quatrième chapitre présentera notre contribution à la bibliothèque `MPHELL`. Nous détaillerons ses fonctionnalités ainsi que la modularité du design. Nous donnerons les performances que l'on obtient avec les différentes configurations possibles de la bibliothèque.

Le cinquième chapitre discutera de notre implantation matérielle de l'arithmétique des courbes quartiques de Jacobi étendues définies sur $\mathbb{F}_p$ et $\mathbb{F}_{p^3}$. Après un rappel de l'état de l'art nous présenterons nos choix d'architecture ainsi que les fonctionnalités que nous avons implantées. Nous donnerons également la parallélisation de l'addition unifiée pour les courbes quartiques de Jacobi étendues que nous avons réalisée ainsi que quelques comparaisons avec l'état de l'art.

Dans le dernier chapitre, nous ferons une conclusion sur les travaux réalisés et nous suggérons différentes pistes d'amélioration et de continuation concernant les algorithmes que nous avons implantés.

Chapitre 2

Formules unifiées pour les courbes elliptiques

Sommaire

Certaines formes de représentation des courbes elliptiques définies sur les corps finis permettent d'utiliser des formules unifiées pour l'addition et le doublement de points. De ce fait, elles apparaissent comme une bonne contre-mesure face à certaines attaques par canaux cachés. De plus, les algorithmes utilisés sont plus performants que ceux classiquement utilisés pour la forme Weierstrass.

Après un rappel sur les propriétés des courbes elliptiques définies sur un corps fini K, nous présenterons en détail deux formes permettant d'avoir des formules unifiées : les courbes quartiques de Jacobi étendues et celles d'Edwards. Nous analyserons les critères qui permettent à une courbe elliptique générique de se mettre sous l'une de ces formes. Nous comparerons également les performances qu'il est possible d'obtenir.

Dans tout le chapitre, K désignera un corps fini.

2.1 Rappels sur les courbes elliptiques

Par commodité pour le lecteur ou la lectrice, nous rappelons dans cette section quelques définitions et propriétés des courbes elliptiques définies sur un corps fini [Coh+12]. Nous donnerons les algorithmes classiques d'addition et de doublement de points ainsi que les différentes manières d'implanter la multiplication scalaire. Ensuite, nous verrons certains concepts de base concernant les opérations entre courbes ainsi que que le comptage de points. Pour finir, nous discuterons de certaines contre-mesures face aux attaques par canaux cachés.

2.1.1 Définition d'une courbe elliptique

Définition 2.1.1
Une courbe elliptique E sur K, notée E/K, est une courbe projective régulière définie par l'ensemble des solutions dans $\mathbb{P}^2(\overline{K})$ de l'équation homogène

$$F(X,Y,Z) : Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 = 0, \qquad (2.1)$$

avec les coefficients $a_1, a_2, a_3, a_4, a_6 \in K$.
Cette équation est appelée forme étendue de Weierstrass.

Soit L un corps avec $K \subset L \subset \overline{K}$. On dira qu'un point de la courbe est L-rationnel si ses coordonnées sont dans L. On désigne par $E(L)$ l'ensemble des points L-rationnels de la courbe. On notera $\mathcal{O}$ le point projectif $(0 : 1 : 0)$. On remarquera que ce point de E est L-rationnel quel que soit L.

Définition 2.1.2
Le discriminant d'une courbe elliptique E/K, noté $\Delta_{E/K}$, est défini par

$$\Delta_{E/K} = -b_2^2 b_8 - 8b_4^3 - 27b_6^2 + 9b_2 b_4 b_6,$$

où

$$b_2 = a_1^2 + 4a_2, b_4 = a_1 a_3 + 2a_4,$$

$$b_6 = a_3^2 + 4a_6, b_8 = a_1^2 a_6 - a_1 a_3 a_4 + 4a_2 a_6 + a_2 a_3^2 - a_4^2.$$

Si la caractéristique de K est différente de 2 et 3, E/K est régulière si et seulement si son discriminant $\Delta_{E/K}$ est non nul. Dans ce cas, le j-invariant de E/K est défini par

$$j_{E/K} = \frac{(b_2^2 - 24b_4)^3}{\Delta_{E/K}}.$$

Il est possible de définir l'équation affine de Weierstrass :

$$F(x, y) : y^2 + a_1 xy + a_3 y - x^3 - a_2 x^2 - a_4 x - a_6 = 0. \tag{2.2}$$

Le changement entre les deux représentations se fait comme suit :

- si $Z \neq 0$, un point projectif $(X : Y : Z)$ correspond au point affine $(X/Z, Y/Z)$ et $Z = 0$ correspond au point à l'infini,
- un point affine (x, y) correspond au point projectif $(x : y : 1)$.

2.1.2 Isomorphismes et isogénies

Il est possible de caractériser les transformations entre courbes. Nous allons donc rappeler les propriétés des isomorphismes et des isogénies.

Définition 2.1.3
Soient E/K et E'/K deux courbes elliptiques définies sur K. E/K et E'/K sont isomorphes sur K si et seulement s'il existe $r, s, t \in K$ et $u \in K^\times$, tels que le changement de variables :

$$x = u^2 x' + r \text{ et } y = u^3 y' + s u^2 x' + t,$$

convertit E/K en E'/K. Un tel changement de variables est dit admissible.

On peut lier l'existence d'isomorphismes entre deux courbes à leur j-invariant respectif. On a la proposition suivante.

Proposition 2.1.4
Si E/K et E'/K ont le même j-invariant alors elles sont isomorphes sur $\overline{K}$. De plus, si deux courbes sont L-isomorphes avec $K \subset L \subset \overline{K}$, elles ont le même j-invariant.

En utilisant la notion de changement de variable admissible, la forme réduite de Weierstrass peut être définie.

Proposition 2.1.5 (Équation réduite de Weierstrass)
Soit E/K une courbe elliptique définie sur K avec $char(K) \neq 2, 3$. L'équation de E/K peut être transformée en :

$$E'/K : y^2 = x^3 - \frac{c_4}{48}x - \frac{c_6}{864}$$

où :

- $c_4 = b_2^2 - 24b_4$ *et* $c_6 = -b_2^3 + 36b_2b_4 - 216b_6$,
- $b_2 = a_1^2 + 4a_2$,
- $b_4 = 2a_4 + a_1a_3$,
- $b_6 = a_3^2 + 4a_6$.

Démonstration. La preuve repose sur deux changements de variables admissibles :

$$x \mapsto x' = x \text{ et } y \mapsto y' = y + \frac{1}{2}\left(a_1x + \frac{a_3}{2}\right)$$

et

$$x \mapsto x' = x + \frac{b_2}{12} \text{ et } y \mapsto y' = y$$

□

Remarque 2.1.6
L'isomorphisme est défini sur K, mais il est également possible de considérer un isomorphisme sur une extension de K. De plus, deux courbes non-isomorphes sur K peuvent le devenir sur une extension et on parle alors de courbes tordues.

Dans la suite, E/K sera donnée par son équation réduite avec $char(K) \neq 2, 3$ et sera notée $E_{W,a,b}/K$ avec $a = -\frac{c_4}{48}$ et $b = -\frac{c_6}{864}$. Cette forme permet de simplifier les équation pour le discriminant et le j-invariant :

$$\Delta_{E/K} = -16(4a^3 + 27b^2) \text{ et } j_{E/K} = 1728\frac{a^3}{4\Delta_{E/K}}.$$

Corollaire 2.1.7 ([Coh+12, Corrolaire 13.16])
Soit $E_{W,a,b}/K$ une courbe elliptique définie sur K par son équation réduite.

- *Si $a = 0$ alors pour tout $b' \in K^\times$, la courbe E est isomorphe à $E' : y^2 = x^3 + b'$ sur $K((b/b')^{1/6})$.*
- *Si $b = 0$ alors pour tout $a' \in K^\times$, la courbe E est isomorphe à $E' : y^2 = x^3 + a'x$ sur $K((a/a')^{1/4})$.*
- *Si $ab \neq 0$ alors pour tout $v \in K^\times$, la courbe E est isomorphe à $\widetilde{E}_v : y^2 = x^3 + a'x + b'$ avec $a' = v^2a$ et $b' = v^3b$ sur $K(v^{1/2})$.*

Les courbes $\widetilde{E}_v$ sont appelées les tordues quadratiques de E. La relation d'isomorphisme est définie sur K si et seulement si v est un carré dans $K^\times$.

Définition 2.1.8
Deux courbes E/K et E'/K sont dites isogènes sur K s'il existe une application $\psi \in Hom_K(E, E')$ telle que $Im(\psi) = E'$ et que $ker(\psi)$ soit fini.

Proposition 2.1.9
Si ψ est une isogénie sur K de deux courbes E/K et E'/K, alors elle admet une isogénie duale, $\hat{\psi} : E' \to E$ définie par :

$$\hat{\psi} \circ \psi = [m]_E \text{ et } \psi \circ \hat{\psi} = [m]_{E'}.$$

2.1.3 Addition et doublement de points

Une courbe elliptique définie sur un corps ayant la structure de groupe abélien, on note $+$ la loi de composition additive associée. Sur $\mathbb{R}$, on peut voir de manière géométrique comment se construit la somme de deux points (voir figure 2.1) .

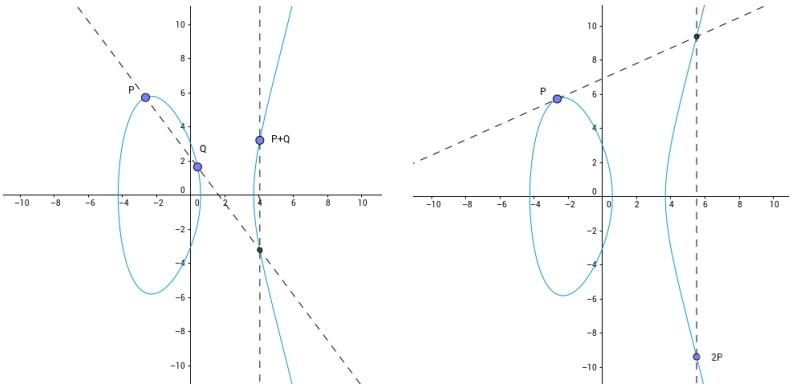


Figure. 2.1 – Loi de groupe sur $y^2 = x^3 - 16x + 9$ définie sur $\mathbb{R}$

Soient P et Q deux points de E. $P \star Q$ est défini comme étant l'intersection de la ligne droite passant par P et Q avec E. Alors le point $P + Q$ est le symétrique (par rapport à l'axe des abscisses) du point $P \star Q$. Pour $2P$, on considère la tangente à la courbe. Nous verrons par la suite les formules associées à ces opérations.

2.1.4 Multiplication scalaire

Pour tout $m \in \mathbb{N}$, on définit le morphisme de groupe $[m]$ par :

$$\begin{aligned} [m] : \quad & E \to E \\ & P \mapsto mP = \underbrace{P + \cdots + P}_{m \text{ fois}} \end{aligned}$$

Remarque 2.1.10
La généralisation au cas $m < 0$ est possible en établissant que $[-m]P = -(mP)$.

On peut adapter les algorithmes classiques pour l'exponentiation à la multiplication scalaire. Il faut remplacer les multiplications, carrés, et divisions par des additions, doublements et soustractions sur la courbe. L'algorithme I.1 est une adaptation de la mé-

Algorithme I.1 : Multiplication Left-to-Right pour les courbes elliptiques

Données : `Un point` P `de la courbe` E `et un entier positif` $n = (n_{\ell-1} \dots n_0)_2$.
Résultat : `Le point` $[n]P$.

1 $Q \leftarrow \mathcal{O}$ `et` $i = \ell - 1$
2 **tant que** $i \geqslant 0$ **faire**
3 $\quad Q \leftarrow [2]Q$
4 $\quad$ **si** $n_i = 1$ **alors** $Q \leftarrow Q + P$
5 $\quad i = i - 1$
6 **retourner** Q

thode d'exponentiation Square and multiply. On peut également déduire une version similaire qui parcourt les bits de n en sens inverse [Coh+12, Algorithme 9.2]. Avec cette méthode, le calcul de $[n]P$ se fait en $O(\log(n))$ opérations sur la courbe.

D'autres techniques permettent d'améliorer les performances, en utilisant par exemple une table de pré-calculs (voir algorithme I.2). Cependant, il faut bien déterminer la taille de celle-ci si la cible d'implantation a un espace mémoire limité. Pour d'autres méthodes d'exponentiation, nous renvoyons à [Coh+12, Chapitre 9].

Algorithme I.2 : Multiplication scalaire avec fenêtre glissante pour les courbes elliptiques

Données : Un point P de la courbe E, un entier positif $n = (n_{\ell-1} \dots n_0)_2$, un paramètre $k \geqslant 1$ et un ensemble de points pré-calculés $[3]P, [5]P, \dots, [(2^k - 1)]P$.

Résultat : Le point $[n]P$.

1 $Q \leftarrow \mathcal{O}$ et $i = \ell - 1$
2 **tant que** $i \geqslant 0$ **faire**
3 **si** $n_i = 0$ **alors**
4 $Q \leftarrow [2]Q$ et $i = i + 1$
5 **sinon**
6 $s = \max(i - k + 1, 0)$
7 **tant que** $n_s = 0$ **faire** $s = s + 1$
8 **pour** $h = 1$ to $i - s + 1$ **faire** $Q \leftarrow [2]Q$
9 $u = (n_i \dots n_s)_2$
10 $Q \leftarrow Q + [u]P$
11 $i = s - 1$
12 **retourner** Q

2.1.5 Points rationnels et points de torsion

Définition 2.1.11
Si $K = \mathbb{F}_q$ alors le morphisme de Frobenius Frob sur E est défini par

$$\begin{array}{rccc} Frob : & E & \to & E \\ & (x, y) & \mapsto & (x^q, y^q) \\ & \mathcal{O} & \mapsto & \mathcal{O} \end{array}$$

Lorsque $K = \mathbb{F}_q$ avec q une puissance d'un nombre premier, la structure de $E(K)$ peut être décrite plus précisément. $E(\mathbb{F}_q)$ est soit cyclique soit isomorphe à $\mathbb{Z}/d_1\mathbb{Z} \times \mathbb{Z}/d_2\mathbb{Z}$ avec $d_1|d_2$ et $d_1|(q-1)$. De plus, le théorème de Hasse donne une borne sur le cardinal de ce groupe en fonction de t, la trace de l'endomorphisme de Frobenius.

Théorème 2.1.12 (Hasse[Has36])
Soit $E/\mathbb{F}_q$ une courbe elliptique définie sur $\mathbb{F}_q$. Alors

$$|E(\mathbb{F}_q)| = q + 1 - t \text{ et } |t| \leqslant 2\sqrt{q}.$$

Remarque 2.1.13
André Weil a ensuite donné une généralisation pour les courbes hyperelliptiques [Wei49].

On peut lier l'existence d'une isogénie entre deux courbes à leurs anneaux d'endomorphisme et au morphisme de Frobenius. On notera $End(E)/K$ l'ensemble des endomorphismes de E définis sur K. Nous avons le théorème suivant :

Théorème 2.1.14 ([Koh96])
Soient E et E' deux courbes elliptiques ordinaires définies sur $\mathbb{F}_p$ qui sont isogènes sur $\mathbb{F}_p$. Soit F un corps quadratique complexe contenant $End(E)/K$, et soit $\mathcal{O}_F$ l'anneau d'entier de F.

1. *L'ordre de $End(E)/K$ vérifie $\mathbb{Z}[Frob] \subseteq End(E)/K \subseteq \mathcal{O}_F$.*
2. *L'ordre de $End(E')/K$ vérifie également $End(E')/K \subset F$ et $\mathbb{Z}[Frob] \subseteq End(E')/K \subseteq \mathcal{O}_F$.*
3. *Les propositions suivantes sont équivalentes :*
   - *$End(E)/K = End(E')/K$.*
   - *Il existe deux isogénies $\phi : E \to E'$ et $\psi : E \to E'$ de degré premier.*
   - *$[\mathcal{O}_F : End(E)/K] = [\mathcal{O}_F : End(E')/K]$.*
   - *$[End(E)/K : \mathbb{Z}[Frob]] = [End(E')/K : \mathbb{Z}[Frob]]$.*
4. *Soit $\phi : E \to E'$ une isogénie de E vers E' de degré premier l. Alors soit $End(E)/K$ contient $End(E')/K$, soit $End(E')/K$ contient $End(E)/K$, et l'index du plus petit dans le plus grand divise l.*
5. *Si l'on suppose que l est un nombre premier qui divise soit $[\mathcal{O}_F : End(E)/K]$, soit $[\mathcal{O}_F : End(E')/K]$, alors toute isogénie $\phi : E \to E'$ a un degré égal à un multiple de l.*

Le conducteur de $End(E)/K$ sera noté c_E et celui de $\mathbb{Z}[Frob]$, c_{Frob}. En appliquant un résultat de [Cox89], on peut déduire que $End(E)/K = \mathbb{Z} + c_E\mathcal{O}_F$ et les discriminants D de $End(E)/K$ et d_F de $\mathcal{O}_F$ vérifient

$$D = c_E^2 d_F.$$

On peut également écrire

$$t^2 - 4p = \text{disc}(\mathbb{Z}[Frob]) = c_{Frob}^2 d_F = c_E^2 [End(E)/K : \mathbb{Z}[Frob]]^2 d_F.$$

Un autre groupe intéressant dans l'analyse d'une courbe elliptique concerne les points de torsion.

Définition 2.1.15
Un point P est d'ordre fini s'il existe m tel que $[m]P = \mathcal{O}$. L'ordre de P est alors le plus petit m pour lequel l'équation est vérifiée. Le noyau de $[m]$ noté $E[m]$, est défini par

$$E[m] = \{P \in E(\overline{K}) | [m]P = \mathcal{O}\}.$$

Les éléments de $E[m]$ sont appelés les points de m-torsion.

Théorème 2.1.16 ([Deu41])
Soit E/K une courbe elliptique définie sur K. Si la caractéristique de K est soit 0 soit premier avec m alors

$$E[m] \cong \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}.$$

Sinon, si $char(K) = p$ et $m = p^r$ alors

$$E[p^r] = \{\mathcal{O}\}, \text{ pour tout } r \geqslant 1 \text{ ou } E[p^r] \cong \mathbb{Z}/p^r\mathbb{Z} \text{ pour tout } r \geqslant 1.$$

Si $E[p^r] = \{\mathcal{O}\}$ pour tout r positif alors on dit que la courbe est *super-singulière* sinon elle est dite *ordinaire*.

Remarque 2.1.17
Pour une courbe définie sur un corps premier $\mathbb{F}_p$ avec $p > 3$, la condition d'être super-singulière est liée au cardinal du groupe des points $\mathbb{F}_p$-rationnel. Elle est super-singulière si et seulement si $|E(\mathbb{F}_p)| = p + 1$.

2.1.6 Arithmétique sur $\mathbb{F}_p$

Nous allons analyser maintenant les formules d'addition et de doublement sur les courbes elliptiques $E_{W,a,b}/K$ définies sur $\mathbb{F}_p$ avec $p > 3$ par leur forme réduite de Weierstrass. Les coûts des opérations seront donnés en fonction du nombre de multiplications (M), d'élévation au carrée (S), de multiplication par une constante (B) et d'additions (A) sur $\mathbb{F}_p$. Nous étudierons trois systèmes de coordonnées, à savoir affines, projectives, et jacobiennes. Nous verrons également les formules co-Z pour les coordonnées jacobiennes. L'ordonnancement des opérations permettant d'atteindre les coûts présentés est donné pour chaque opération dans l'annexe A et proviennent de [BL].

Coordonnées affines

Soient $P = (x_1, y_1)$ et $Q = (x_2, y_2)$ deux points donnés en coordonnées affines. Pour calculer $P + Q = (x_3, y_3)$ avec $P \neq \pm Q$, nous avons les formules suivantes :

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2, \\ y_3 &= \lambda(x_1 - x_3) - y_1, \end{aligned} \tag{2.3}$$

avec $\lambda = \frac{y_1 - y_2}{x_1 - x_2}$.

Pour calculer $[2]P = (x_3, y_3)$, nous avons les formules suivantes :

$$\begin{aligned} x_3 &= \lambda^2 - 2x_1, \\ y_3 &= \lambda(x_1 - x_3) - y_1, \end{aligned} \tag{2.4}$$

avec $\lambda = \frac{3x_1^2+a}{2y_1}$.

Les coûts respectifs de l'addition et du doublement sont donc de $I+2M+S$ et $I+2M+2S$. L'inversion étant, en général, une opération très coûteuse, ce système de représentation est rarement utilisé pour les calculs.

Coordonnées projectives

Pour rappel, en coordonnées projectives, l'équation de la courbe devient

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

Un point projectif $(X_1 : Y_1 : Z_1)$ correspond au point affine $(X_1/Z_1, Y_1/Z_1)$ quand $Z_1 \neq 0$ et au neutre $\mathcal{O} = (0 : 1 : 0)$ sinon. L'opposé de $(X_1 : Y_1 : Z_1)$ est $(X_1 : -Y_1 : Z_1)$.

Pour calculer $P + Q = (X_3 : Y_3 : Z_3)$ avec $P \neq \pm Q$, nous avons les formules suivantes :

$$\begin{aligned} A &= Y_2Z_1 - Y_1Z_2, \\ B &= X_2Z_1 - X_1Z_2, \\ C &= A^2Z_1Z_2 - B^3 - 2B^2X_1Z_2, \\ X_3 &= BC, \\ Y_3 &= A(B^2X_1Z_2 - C) - B^3Y_1Z_2, \\ Z_3 &= B^3Z_1Z_2. \end{aligned} \tag{2.5}$$

Le coût est de $12M + 2S + 6A$.

Pour calculer $[2]P = (X_3 : Y_3 : Z_3)$, nous avons les formules suivantes :

$$\begin{aligned} A &= aZ_1^2 + 3X_1^2, \\ B &= Y_1Z81, \\ C &= X_1Y_1B, \\ D &= A^2 - 8C, \\ X_3 &= 2BD, \\ Y_3 &= A(4C - D) - 8Y_1^2B^2, \\ Z_3 &= 8B^3. \end{aligned} \tag{2.6}$$

Le coût est de $5M + 6S + 1B + 7A$. La constante a intervient dans les calculs. En prenant $a = -3$, on arrive à améliorer le coût précédent en $7M + 3S + 5A$.

Coordonnées jacobiennes

Dans ce système de coordonnées, l'équation de courbe considérée est

$$Y^2 = X^3 + aXZ^4 + bZ^6.$$

Un point $(X_1 : Y_1 : Z_1)$ correspond au point affine $(X_1/Z_1^2, Y_1/Z_1^3)$ quand $Z_1 \neq 0$ et au neutre $\mathcal{O} = (1 : 1 : 0)$ sinon. L'opposé de $(X_1 : Y_1 : Z_1)$ est $(X_1 : -Y_1 : Z_1)$.

Pour calculer $P + Q = (X_3 : Y_3 : Z_3)$ avec $P \neq \pm Q$, nous avons les formules suivantes :

$$\begin{aligned} A &= X_1 Z_2^2, \\ B &= X_2 Z_1^2, \\ C &= Y_1 Z_2^3, \\ D &= Y_2 Z_1^3, \\ E &= B - A, \\ F &= D - C, \\ X_3 &= -E^3 - 2AE^2 + F^2, \\ Y_3 &= -CE^3 + F(AE^2 - X_3), \\ Z_3 &= Z_1 Z_2 E. \end{aligned} \tag{2.7}$$

Le coût est de $11M + 5S + 9A$.

Pour calculer $[2]P = (X_3 : Y_3 : Z_3)$, nous avons les formules suivantes :

$$\begin{aligned} A &= 4X_1 Y_1^2, \\ B &= 3X_1^2 + aZ_1^4, \\ X_3 &= -2A + B^2, \\ Y_3 &= -8Y_1^4 + B(A - X_3), \\ Z_3 &= 2Y_1 Z_1. \end{aligned} \tag{2.8}$$

Le coût est de $1M + 8S + 1B + 10A$. La constante a intervient dans les calculs. En prenant $a = -3$, on arrive à améliorer le coût précédent en $3M + 5S + 8A$.

Si on compare les coûts pour les coordonnées projectives et jacobiennes, on remarque que pour l'addition de points les formules en projective sont plus rapides. Pour le doublement, c'est l'inverse qui est observé.

D'autres systèmes de coordonnées ont été dérivés des coordonnées jacobiennes. On retrouve ainsi les coordonnées jacobiennes Chudnovsky [CC86] et les coordonnées jacobiennes modifiées [CMO98] qui augmentent le nombre de coordonnées afin de précalculer certaines variables qui interviennent dans les formules d'addition et de doublement.

Addition co-Z

Les coordonnées co-Z sont basées sur le systèmes de coordonnées jacobiennes mais on se restreint au cas où les points possèdent la même coordonnée Z. Elles ont été introduites par Meloni dans [Mel07]. Il a remarqué que l'on pouvait obtenir une addition plus rapide en utilisant les formules suivantes : Soient $P = (X_1 : Y_1 : Z)$ et $Q = (X_2 : Y_2 : Z)$. $R = P + Q = (X_3 : Y_3 : Z_3)$ est calculé avec

$$\begin{aligned} A_1 &= Y_1(W_1 - W_2), \\ W_1 &= X_1 C, \\ W_2 &= X_2 C, \\ C &= (X_1 - X_2)^2, \\ D &= (Y_1 - Y_2)^2, \\ X_3 &= D - W_1 - W_2, \\ Y_3 &= (Y_1 - Y_2)(W_1 - X_3) - A_1, \\ Z_3 &= Z(X_1 - X_2). \end{aligned} \tag{2.9}$$

Il a également remarqué que l'on pouvait mettre à jour, sans coût, la coordonnée Z de P, de manière à ce que P et R aient le même Z. Plus précisément, on a :

$$(X_1(X_1 - X_2)^2 : Y_1(X_1 - X_2)^3 : Z_3) = (W_1 : A_1 : Z_3) = P.$$

Le coût d'une telle addition est de $5M + 2S + 7A$.

2.1.7 Attaques par canaux cachés

Les attaques par canaux cachés (SCA [1]) regroupent l'ensemble des méthodes qui exploitent le comportement de la cible d'implantation pendant un calcul critique de manière passive. On peut par exemple, analyser la consommation de courant, les émissions électro-magnétiques, le temps de calcul. Tout cela permet de révéler des informations sur les clés ou des nonces présents en mémoire.

Dans les cas d'utilisation des courbes elliptiques, la clé peut apparaître au niveau de la multiplication scalaire. Selon l'algorithme utilisé, il faut mettre en place plus ou moins de contre-mesures. Par exemple, dans l'algorithme I.1, selon les bits de la clé, une addition est réalisée. En espionnant une trace de courant par exemple, on peut récupérer l'ordre des opérations et par suite la clé. Avec le fenêtrage, l'attaque est plus complexe mais il y a encore un test sur une valeur dérivée de la clé. Un attaquant peut donc exploiter cette

1. Side-Channel Attacks

faille. Une contre-mesure serait d'utiliser de fausses opérations durant le calcul [Cor99]. Cependant, une telle implantation favorise d'autre types d'attaques [YJ00 ; SMKLM01].

Afin d'éviter l'insertion de fausses opérations, on peut utiliser l'échelle de Montgomery [Mon87].

Algorithme I.3 : Échelle de Montgomery pour les courbes elliptiques

Données : Un point P de la courbe E et un entier positif $n = (n_{l-1} \dots n_0)_2$.
Résultat : Le point $[n]P$.

1 $Q_0 = \mathcal{O}$ et $Q_1 = P$
2 **pour** $i = l - 1$ à 0 **faire**
3 $\quad b = n_i$
4 $\quad Q_{1-b} = Q_{1-b} + Q_b$
5 $\quad Q_b = [2]Q_b$
6 **retourner** Q_0

À chaque itération de la boucle, il y a à la fois une addition et un doublement. De ce fait, le nombre de bits à 1 de la clé n'influe pas sur l'ordre et la quantité d'opérations (voir algorithme I.30). Pour ce faire, deux registres sont utilisés et un invariant est préservé dans la boucle. En effet, on peut vérifier que

$$Q_b^{(new)} - Q_{1-b}^{(new)} = (Q_b + Q_{1-b}) - [2]R_{1-b} = Q_b - Q_{1-b}.$$

Cela permet également de réaliser les multiplications scalaires en utilisant seulement la coordonnée x.

Joye a également proposé un algorithme dans le même esprit que l'échelle de Montgomery [Joy07]. Les différences résident dans le fait que les bits du scalaire sont parcourus en ordre inverse, et une seule opération apparaît à chaque itération de la boucle.

Dans [GJM10], Goundar, Joye et Miyaji ont proposé une implantation efficace basée sur l'algorithme de Joye en utilisant les formules d'addition co-Z. Le coût d'une multiplication scalaire par bit est de $9M + 7S$.

Au lieu de se concentrer sur l'algorithme de multiplication, certains ont étudié les formules des opérations sur les courbes. L'idée d'utiliser des représentations de courbes elliptiques qui permettent d'avoir un même algorithme pour l'addition et le doublement (formules unifiées) a été introduite indépendamment dans [LS01] et [JQ01a]. Dans [JQ01a], ils proposent d'utiliser des courbes Hessiennes alors que [LS01] suggère de représenter les courbes elliptiques comme l'intersection de deux surfaces quadriques. Cependant, il est également possible d'obtenir de telles formules sous la forme Weierstrass.

Algorithme 1.4 : Algorithme de multiplication scalaire de Joye pour les courbes elliptiques

Données : Un point P de la courbe E et un entier positif $n = (n_{l-1} \dots n_0)_2$.
Résultat : Le point $[n]P$.

$Q_0 = \mathcal{O}$ et $Q_1 = P$
pour $i = 0$ à $l - 1$ **faire**
  $b = n_i$
  $Q_{1-b} = [2]Q_{1-b} + Q_b$
retourner Q_0

Dans [RCB15], les auteurs proposent une addition ayant un coût de $12M + 5B + 23A$ en utilisant des coordonnées projectives. Soit $E_{W,a,b}/\mathbb{F}_p$ une courbe elliptique donnée sous forme de Weierstrass et définie sur $\mathbb{F}_p$. Soient $P_1 = (X_1 : Y_1 : Z_1)$ et $P_2 = (X_2 : Y_2 : Z_2)$ deux points projectifs de $E_{W,a,b}/\mathbb{F}_p$. La somme $P_3 = (X_3 : Y_3 : Z_3)$ est calculée avec :

$$\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2) - \\
&\quad (Y_1Z_2 + Y_2Z_1)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2), \\
Y_3 &= (3X_1X_2 + aZ_1Z_2)(aX_1X_2 + 3b(X_1Z_2 + X_2Z_1) - a^2Z_1Z_2) + \\
&\quad (Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2)(Y_1Y_2 - a(X_1Z_2 + X_2Z_1) - 3bZ_1Z_2), \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)(Y_1Y_2 + a(X_1Z_2 + X_2Z_1) + 3bZ_1Z_2) + \\
&\quad (X_1Y_2 + X_2Y_1)(3X_1X_2 + aZ_1Z_2).
\end{aligned} \tag{2.10}$$

Nous allons présenter en détail dans la section suivante les courbes quartiques de Jacobi étendues, puis les courbes d'Edwards qui permettent également d'avoir des formules unifiées.

2.2 Courbes quartiques de Jacobi étendues

Cette section présente l'historique des courbes quartiques de Jacobi étendues ainsi que l'évolution des coûts des formules d'addition unifiées que l'on peut trouver dans la littérature. Nous comparerons ces coûts à ceux que l'on peut obtenir classiquement avec la représentation de Weierstrass. Ensuite, nous analyserons les possibilités de transfert entre la forme Weierstrass et celle-ci.

2.2.1 Intersection de quadriques

Il est possible de représenter une courbe elliptique sur K par une intersection de deux quadriques dans $\mathbb{P}^3(K)$ [CF96, chapitre 7]. On appelle cette représentation la forme de Jacobi. Le changement se fait comme suit :

Un point (x, y) appartenant à une courbe $E_{W,a,b}$ correspond au point (X, Y, Z, T) de l'intersection

$$\begin{aligned} X^2 - TZ &= 0, \\ Y^2 - aXZ - bZ^2TX &= 0, \end{aligned} \tag{2.11}$$

en utilisant le changement de variable $(x, y) \mapsto (x, y, 1, x^2)$. Comme pour la forme Weierstrass, on peut définir la somme de deux points.

Soient $P_1 = (X_1, Y_1, Z_1, T_1)$ et $P_2 = (X_2, Y_2, Z_2, T_2)$ deux points de l'intersection définie par 2.11. Alors la somme $P_3 = (X_3, Y_3, Z_3, T_3)$ est obtenue par :

$$\begin{aligned} X_3 &= \mu_1(P_1, P_2)\mu_3(P_1, P_2), \\ Y_3 &= Y_1\mu_2(P_2, P_1) + Y_2\mu_2(P_1, P_2), \\ Z_3 &= \mu_3(P_1, P_2)^2, \\ T_3 &= \mu_1(P_1, P_2)^2, \end{aligned} \tag{2.12}$$

avec

$$\begin{aligned} \mu_1(P_1, P_2) &= T_1T_2 - 2aX_1X_2 - 4b(X_1Z_2 + Z_1X_2) + a^2Z_1Z_2, \\ \mu_2(P_1, P_2) &= T_1^2T_2 + 2aX_1T_1X_2 + 4bX_1T_1Z_2 + 3aZ_1T_1T_2 \\ &\quad + 12bZ_1T_1X_2 - 3a^2Z_1T_1Z_2 + 4bZ_1X_1T_2 - 2a^2X_1Z_1X_2 \\ &\quad - 4abX_1Z_1Z_2 - a^3Z_1^2Z_2 - 8b^2Z_1^2Z_2, \\ \mu_3(P_1, P_2) &= 2Y_1Y_2 + X_1T_2 + T_1X_2 + a(X_1Z_2 + Z_1X_2) + 2bZ_1Z_2. \end{aligned} \tag{2.13}$$

Ces formules sont valables à la fois pour l'addition et le doublement de points. De ce fait, l'intersection de quadriques présente une contre-mesure naturelle aux attaques SPA. Cependant comme le remarque Liardet et Smart dans [LS01], ces formules sont trop complexes pour être utilisées dans des implantations réelles. Partant de cela, ils ont étudiés une classe particulière des courbes elliptiques qui permettent d'obtenir de meilleures performances tout en garantissant cette résistance à la SPA.

Ils se sont intéressés aux courbes ayant trois points d'ordre deux définis sur K. Dans ce contexte, il faut que l'ordre des points K-rationnels soit divisible par 4. Cela restreint donc les courbes à étudiées et en particulier les courbes du NIST. Soient donc $E'_{W,\lambda}$ les nouvelles courbes considérées d'équation affine :

$$y^2 = x(x + 1)(x + \lambda).$$

L'intersection de quadriques correspondante est donnée par :

$$\begin{aligned} X^2 + Y^2 - T^2 &= 0, \\ (1-\lambda)X^2 + Z^2 - T^2 &= 0, \end{aligned} \tag{2.14}$$

avec les changements de coordonnées :

$$(x, y) \mapsto (-2y, x^2 - \lambda, x^2 + 2x\lambda + \lambda, x^2 + 2x + \lambda), O \mapsto (0, 1, 1, 1),$$

et

$$(X, Y, Z, T) \mapsto \begin{cases} O & \text{si } X = 0 \text{ et } Y = Z = T, \\ \left(\frac{\lambda(Z-T)}{(1-\lambda)Y - Z + \lambda T}, \frac{\lambda(1-\lambda)X}{(1-\lambda)Y - Z + \lambda T} \right) & \text{sinon.} \end{cases}$$

L'opposé du point $(X : Y : Z)$ est obtenu en remplaçant la première coordonnée par son opposé. Les formules d'addition de points pour $P_3 = P_1 + P_2$ deviennent :

$$\begin{aligned} X_3 &= T_1 Y_2 X_1 Z_2 + Z_1 X_2 Y_1 T_2, \\ Y_3 &= T_1 Y_2 Y_1 T_2 - Z_1 X_2 X_1 Z_2, \\ Z_3 &= T_1 Z_1 T_2 Z_2 - k^2 X_1 Y_1 X_2 Y_2, \\ T_3 &= (T_1 Y_2)^2 + (Z_1 X_2)^2. \end{aligned} \tag{2.15}$$

Le coût de l'addition est donc de 16 multiplications plus une multiplication par la constante k^2 et ces formules sont unifiées. Afin d'obtenir de meilleures performances, Liardet et Smart ont proposé une implantation du doublement utilisant seulement 7 multiplications. Afin de concilier la résistance face aux attaques SPA et la performance, une stratégie invoquée est d'utiliser aléatoirement soit le doublement avec la version unifiée (en 16 multiplications) soit la version plus rapide (en 7 multiplications).

2.2.2 Courbes quartiques de Jacobi étendues

Dans [BJ03a], des quartiques plus générales sont étudiées. On va considérer les courbes $E_{J,d,a}/K$ données par l'équation

$$y^2 = dx^4 + 2ax^2 + 1, \tag{2.16}$$

avec $a, d \in K$ tels que $256d(a^2 - d)^2 \neq 0$.

Les courbes initiales de Jacobi sont alors un cas particulier avec $d = k^2$ et $a = -(1 + k^2)/2$. Dans la suite, ce type de courbes sera désigné par le terme quartique de Jacobi étendue ou simplement quartique de Jacobi quand $d = 1$. En version projective, on obtient l'équation suivante :

$$Y^2 = dX^4 + 2aX^2Z^2 + Z^4. \tag{2.17}$$

Les points sont donnés en coordonnées projectives pondérées. Deux triplets $(X_1 : Y_1 : Z_1)$ et $(X_2 : Y_2 : Z_2)$ représentent le même point si et seulement s'il existe $t \in K^\times$ tel que

$$X_1 = tX_2, Y_1 = t^2 Y_2 \text{ et } Z_1 = tZ_2.$$

L'opposé du point $(X : Y : Z)$ est obtenu en remplaçant la première coordonnée par son opposé. La somme $(X_3 : Y_3 : Z_3)$ de deux points peut être calculée avec la formule unifiée suivante utilisant 10 multiplications, 3 carrés, 3 multiplications par une constante et 13 additions.

$$\begin{aligned} X_3 &= X_1Z_1Y_2 + Y_1X_2Z_2, \\ Y_3 &= [(Z_1Z_2)^2 + d(X_1X_2)^2][Y_1Y_2 + 2aX_1X_2Z_1Z_2] + 2dX_1X_2Z_1Z_2(X_1^2Z_2^2 + Z_1^2X_2^2), \\ Z_3 &= (Z_1Z_2)^2 - d(X_1X_2)^2. \end{aligned} \tag{2.18}$$

La correspondance avec la forme de Weierstrass est donnée par le théorème suivant :

Théorème 2.2.1 ([BJ03a])
Soient $E_{W,a,b}/K$ une courbe elliptique définie sur K donnée par l'équation de Weierstrass réduite $y^2 = x^3 + ax + b$ avec $a, b \in K$ et $\mathcal{O}$ son point à l'infini. Si on suppose que E admet un point de 2-torsion $(\theta, 0) \in E_{W,a,b}(K)$, alors $E_{W,a,b}/K$ est birationnellement équivalente à la courbe quartique de Jacobi étendue $E_{J,-(3\theta^2+4a)/16,-3\theta/4}/K$. Les changements de coordonnées sont donnés par les transformations suivantes :

$$\begin{cases} (\theta, 0) & \longmapsto (0 : -1 : 1), \\ (x, y) & \longmapsto (2(x-\theta) : (2x+\theta)(x-\theta)^2 - y^2 : y), \\ \mathcal{O} & \longmapsto (0 : 1 : 1), \end{cases} \tag{2.19}$$

et

$$\begin{cases} (0 : 1 : 1) & \longmapsto \mathcal{O}, \\ (0 : -1 : 1) & \longmapsto (\theta, 0), \\ (X, Y, Z) & \longmapsto \left(2\frac{(Y+Z^2)}{X^2} - \frac{\theta}{2}, Z\frac{4(Y+Z^2)-3\theta X^2}{X^3}\right). \end{cases} \tag{2.20}$$

Nous donnons la preuve de ce résultat car il permet d'illustrer comment la conversion de courbes est réalisée et l'on peut déduire des simplifications quand certains paramètres sont fixés ou que l'on a trois points d'ordre 2.

Démonstration. Soient $P(x, y)$ un point de $E_{W,a,b}$, $X = 2(x-\theta)$, $Y = (2x+\theta)(x-\theta)^2 - y^2$ et $Z = y$. On va montrer que X, Y, Z vérifient l'équation 2.16 et donc que le point $(X : Y : Z)$ appartient à la courbe $E_{J,-(3\theta^2+4a)/16,-3\theta/4}$.

Soient $d = -(3\theta^2 + 4a)/16$ et $a_j = -3\theta/4$.

On a :

$$Y^2 = (2x + \theta)^2(x - \theta)^4 + y^4 - 2y^2(2x + \theta)(x - \theta)^2,$$

$$dX^4 + 2a_jX^2Z^2 + Z^4 = -(3\theta^2 + 4a)(x - \theta)^4 - 6\theta(x - \theta)^2y^2 + y^4,$$

donc

$$\begin{aligned} Y^2 - dX^4 - 2a_jX^2Z^2 - Z^4 &= 4(x - \theta)^4(x^2 + x\theta + \theta^2 + a) - 4y^2(x - \theta)^3, \\ &= 4(x - \theta)^3((x - \theta)(x^2 + x\theta + \theta^2 + a) - y^2), \\ &= 4(x - \theta)^3(x^3 + ax - y^2 - \theta^3 - a\theta). \end{aligned}$$

Puisque $(\theta, 0)$ est un point de 2-torsion, on a $\theta^3 + a\theta + b = 0$. Or (x, y) étant un point de $E_{W,a,b}$, on peut en déduire que

$$Y^2 - dX^4 - 2a_jX^2Z^2 - Z^4 = 0.$$

Ainsi, X, Y, Z vérifient l'équation 2.16.

Considérons maintenant un point $(X : Y : Z)$ de $E_{J,-(3\theta^2+4a)/16,-3\theta/4}$, et montrons que $\left(2\frac{(Y+Z^2)}{X^2} - \frac{\theta}{2}, Z\frac{4(Y+Z^2)-3\theta X^2}{X^3}\right)$ est un point de $E_{W,a,b}$.

Soient donc $x = 2\frac{(Y+Z^2)}{X^2} - \frac{\theta}{2}$ et $y = Z\frac{4(Y+Z^2)-3\theta X^2}{X^3}$. Nous allons réécrire $x^3 + ax + b$.

$$\begin{aligned} x^3 + ax + b &= (2\frac{(Y + Z^2)}{X^2} - \frac{\theta}{2})^3 + a(2\frac{(Y + Z^2)}{X^2} - \frac{\theta}{2}) + b, \\ &= \frac{(Y + Z^2)}{X^6}[8(Y + Z^2)^2 - 6\theta X^2(Y + Z^2) + (\frac{3\theta^2}{2} + 2a)X^4] - \frac{3\theta}{8}(3\theta^2 + 4a). \end{aligned}$$

On sait que $(X : Y : Z)$ de $E_{J,-(3\theta^2+4a)/16,-3\theta/4}$ donc :

$$X^4(3\theta^2 + 4a) = -16Y^2 - 24\theta X^2Z^2 + 16Z^4.$$

On peut alors écrire :

$$\begin{aligned}
x^3+ax+b &= \frac{(Y+Z^2)}{X^6}[8(Y+Z^2)^2-6\theta X^2(Y+Z^2)+8Z^4-8Y^2-12\theta X^2Z^2]- \\
&\frac{3\theta}{X^4}(2Z^4-2Y^2-3\theta X^2Z^2), \\
&= \frac{Z^2}{X^6}[16Z^2(Y+Z^2)+16Y(Y+Z^2)-18\theta X^2(Y+Z^2)-6\theta X^2Z^2+9\theta^2X^4]- \\
&\frac{6\theta X^2Y(Y+Z^2)-6\theta X^2Y^2}{X^6}, \\
&= \frac{Z^2}{X^6}[16(Y^2+2YZ+Z^4)-24\theta X^2(Y+Z^2)+9\theta^2X^4], \\
&= \frac{Z^2}{X^6}[(4(Y+Z^2))^2-24\theta X^2(Y+Z^2)+(3\theta X^2)^2], \\
&= \frac{Z^2}{X^6}[4(Y+Z^2)-3\theta X^2]^2, \\
&= \left(Z\frac{4(Y+Z^2)-3\theta X^2}{X^3}\right)^2, \\
&= y^2.
\end{aligned}$$

(x,y) est par conséquent un point de $E_{W,a,b}$. □

2.2.3 Représentation redondante

La forme quartique de Jacobi étendue permet d'utiliser différentes représentations de points qui améliorent le coût des opérations. Nous allons voir dans ce paragraphe les différents systèmes qui sont apparus dans la littérature et qui ont la particularité de rajouter des coordonnées aux points afin d'obtenir de meilleures performances. Nous présenterons en particulier les résultats obtenus pour des formules unifiées mais nous donnerons malgré tout les références pour des implantations plus rapides mais non unifiées. Cela peut avoir de l'intérêt pour la partie de vérification de signature par exemple, qui n'utilise que des données publiques.

Une première représentation redondante a été donnée par Duquesne dans [Duq07]. Il utilise quatre coordonnées au lieu de trois. Ainsi un point $(X:Y:Z)$ est représenté par $(X^2:XZ:Z^2:Y)$. Avec ce système, il obtient une formule d'addition unifiée en $9M+2S+3D+13A$.

Bernstein et Lange [BL] ont également proposé deux représentations avec six et cinq coordonnées, $(X:Y:Z:X^2:2XZ:Z^2)$ et $(X:Y:Z:X^2:2XZ:X^2+Z^2)$ qui permettent de transformer une multiplication en une élévation au carré par rapport à la version précédente.

La même année, Hisil et al [HWCD09b] ont également donné une nouvelle formule unifiée utilisant cette fois ci une représentation avec six coordonnées. Ils représentent un point avec $Z \neq 0$ par un sextuplet $(X : Y : Z : X^2 : Z^2 : XZ)$. L'algorithme I.5 permet d'obtenir une addition unifiée en $7M + 3S + 1D$.

Algorithme I.5 : Addition unifiée pour la forme quartique de Jacobi

Données : Deux points $(X_1 : Y_1 : Z_1 : U_1 : V_1 : W_1)$ et $(X_2 : Y_2 : Z_2 : U_2 : V_2 : W_2)$ sur $E_{J,1,a}$ avec $U_i = X_i^2$, $V_i = Z_i^2$ et $W_i = X_i Z_i$

Résultat : La somme $(X_3 : Y_3 : Z_3 : U_3 : V_3 : W_3)$

1 $A = U_1U_2$, $B = V_1V_2$, $C = W_1W_2$, $D = Y_1Y_2$
2 $X_3 = (W_1 + Y_1)(W_2 + Y_2) - C - D$, $Z_3 = B - A$
3 $U_3 = X_3^2$, $V_3 = Z_3^2$, $F = A + B + 2C$
4 $G = (U_1 + V_1)(U_2 + V_2) + 2(a-1)C + D$, $H = U_3 + V_3$
5 $Y_3 = FG - H$, $W_3 = ((X_3 + Z_3)^2 - H)/2$

Une version moins redondante permet d'obtenir un coût en $7M + 4S + 1D$. Ils utilisent le système de coordonnées introduit dans [HCD07], qui représente un point par le quintuplet $(X : Y : Z : X^2 : Z^2)$.

Les mêmes auteurs ont encore amélioré leurs formules en utilisant uniquement quatre coordonnées [HWCD09a]. Ils utilisent une représentation homogène projective étendue. Pour rappel, dans le système homogène projectif, un point (x, y) est représenté par le triplet $(X : Y : Z)$ qui vérifie l'équation projective

$$Y^2Z^2 = dX^4 + 2aX^2Z^2 + Z^4.$$

Les propriétés suivantes sont vérifiées :

- Le point $(X : Y : Z)$ correspond au point affine $(X/Z, Y/Z)$ pour $Z \neq 0$.
- L'élément neutre est représenté par $(0 : 1 : 1)$.
- L'opposé du point $(X : Y : Z)$ est $(-X : Y : Z)$.

Dans sa version étendue, une quatrième coordonnée T est introduite. Elle vérifie $T = X^2/Z$. Par conséquent, un point affine (x, y) est représenté par $(X : Y : T : Z) = (x : y : x^2 : 1)$. Chaque quadruplet $(X : Y : T : Z)$ avec $Z \neq 0$ vérifie les équations suivantes :

$$\begin{aligned} X^2 - TZ &= 0, \\ Y^2 - dT^2 - 2aX^2 - Z^2 &= 0. \end{aligned} \tag{2.21}$$

Dans ce système, l'élément neutre est représenté par $(0 : 1 : 0 : 1)$ et l'opposé un point est obtenu en changeant la première coordonnée par son opposé.

Les auteurs ont fourni des formules dédiées à l'addition et au doublement qui améliorent les précédents résultats (voir [HWCD09a, sections 4.1 et 4.2]) mais ils proposent également une formule unifiée.

Soient deux points $(X_1 : Y_1 : T_1 : Z_1)$ et $(X_2 : Y_2 : T_2 : Z_2)$ appartenant à la courbe $E_{J,a,d}$ avec $Z_1 \neq 0$ et $Z_2 \neq 0$. Une addition unifiée pour le calcul de

$$(X_3 : Y_3 : T_3 : Z_3) = (X_1 : Y_1 : T_1 : Z_1) + (X_2 : Y_2 : T_2 : Z_2)$$

peut être obtenue avec les formules suivantes :

$$\begin{aligned} X_3 &= (X_1Y_2 + Y_1X_2)(Z_1Z_2 - dT_1T_2), \\ Y_3 &= (Y_1Y_2 + 2aX_1X_2)(Z_1Z_2 + dT_1T_2) + 2dX_1X_2(T_1Z_2 + Z_1T_2), \\ T_3 &= (X_1Y_2 + Y_1X_2)^2, \\ Z_3 &= (Z_1Z_2 - dT_1T_2)^2. \end{aligned} \tag{2.22}$$

Elles sont complètes selon le lemme 2.1 ([HWCD09a]) quand d n'est pas un carré dans K. Si d est un carré dans K ($d = s^2$), une nouvelle équation pour le calcul de Y^3 permet d'être encore plus rapide. En effet, on peut écrire

$$Y_3 = (Z_1Z_2 + dT_1T_2 + 2sX_1X_2)(Y_1Y_2 + 2aX_1X_2 + sT_1Z_2 + sZ_1T_2) - sT_3.$$

On obtient ainsi une addition en $7M + 3S + 5D + 18A$. Le seul inconvénient est que la formule n'est plus complète. Le lemme 2.2 ([HWCD09a]) permet cependant de passer outre ce problème quand les points considérés sont d'ordre impair. En particulier, les points utilisés en cryptographie sont d'ordre premier impair.

Si l'on résume les différents coûts que l'on a pour une formule unifiée en utilisant une courbe quartique de Jacobi étendue, on obtient les tableaux 2.1 et 2.2.

Articles	Représentation	Coûts
[BJ03a]	3 coordonnées	$10M + 3S + 3D + 13A$
[Duq07]	4 coordonnées	$9M + 2S + 3D + 13A$
[HWCD09a]	4 coordonnées	$8M + 3S + 3D + 17A$ ou $9M + 2S + 3D + 13A$
[HWCD09a]	d est un carré dans dans K	$7M + 3S + 5D + 18A$ ou $8M + 2S + 5D + 14A$

Tableau 2.1 – Coût d'addition unifiée pour la forme quartique de Jacobi étendue

Le choix de la formule optimale ne dépend pas uniquement du nombre de multiplications ou d'élévations au carré de l'algorithme. En effet, il faut prendre en compte le rapport entre le coût d'une multiplication et celui d'une addition car cela peut avantager

des formules différentes selon les cibles d'implantation. Par exemple, dans [HWCD09a], ils fournissent différents agencements des opérations qui augmentent plus ou moins le nombre d'addition.

Articles	Représentation	Coûts
[BJ03a]	3 coordonnées	$10M + 3S + 1D$
[Duq07]	4 coordonnées	$9M + 2S + 1D$
[BL]	6 coordonnées	$8M + 3S + 1D$
[HWCD09b]	5 coordonnées	$7M + 4S + 1D$
[HWCD09b]	6 coordonnées	$7M + 3S + 1D$
[HWCD09a]	4 coordonnées	$7M + 3S + 1D$

Tableau 2.2 – Coût d'addition unifiée pour la forme quartique de Jacobi

2.2.4 Transfert Weierstrass

Soit $E_{W,a,b}/\mathbb{F}_p$ une courbe elliptique définie sur K par sa forme réduite de Weierstrass. Nous avons vu précédemment, que si cette courbe possédait au moins un point de 2-torsion sur K alors, on pouvait la transformer en une courbe quartique de Jacobi étendue (voir Théorème 2.2.1). De plus, si l'on trouve trois points de 2-torsion, on peut se ramener au cas où $d = 1$ [BJ03a]. Si aucun point de 2-torsion n'est trouvé, il est nécessaire de travailler sur $\mathbb{F}_{p^3}$.

La problématique reste à déterminer la proportion de courbes pouvant se mettre sous la forme quartique de Jacobi étendue (avec ou non $d = 1$). Les travaux de Plût permettent de répondre à celle-ci [Plu11]. Il utilise l'action du morphisme de Frobenius sur le sous-groupe des points de 4-torsion afin de caractériser les proportions de courbes pouvant se mettre sous cette forme. Il donne également les proportions pour les tordues d'Edwards que nous verrons dans la section suivante 2.3

Courbes	p impair	$p = 1 \bmod 4$	$p = -1 \bmod 4$
quartique de Jacobi étendue	2/3	2/3	2/3
quartique de Jacobi étendue ($d = 1$)	5/32	7/48	1/6
Tordue d'Edwards	17/48	1/3	3/8

Tableau 2.3 – Proportions de courbes elliptiques pouvant se mettre sous forme quartique de Jacobi étendue et Edwards [Plu11]

2.3 Courbes d'Edwards

En se basant sur des résultats d'Euler et de Gauss, Edwards a introduit une nouvelle forme de courbe elliptique [Edw07]. Il a montré que toute courbe elliptique définie sur un corps de nombre pouvait se mettre sous la forme

$$x^2 + y^2 = c^2 + c^2x^2y^2,$$

avec $(0, c)$ comme élément neutre.

Dans [BL07a], Bernstein et Lange ont donné une version plus générale des courbes introduites par Edwards ainsi que des algorithmes pour calculer les opérations en utilisant des coordonnées projectives.

Définition 2.3.1
Une courbe d'Edwards affine $E_{E,c,d}$ définie sur K est donné par l'équation suivante

$$x^2 + y^2 = c^2(1 + dx^2y^2),$$

avec $c, d \in K$ tels que $c, d \neq 0$ et $dc^4 \neq 1$.

C'est une courbe elliptique car son discriminant, $cd(1 - dc^4)$ est différent de 0. L'élément neutre est le point $(0, 1)$ et l'inverse du point (x_1, y_1) est $(-x_1, y_1)$.

La somme de deux points (x_1, y_1), (x_2, y_2) de $E_{E,d}$ donnés en coordonnées affines est :

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{c(1 + dx_1x_2y_1y_2)}, \frac{y_1y_2 - x_1x_2}{c(1 - dx_1x_2y_1y_2)} \right).$$

Ces formules sont unifiées. Elles sont également complètes si d n'est pas un carré dans K [BL07a, Théorème 3.3].

En version projective, l'équation de la courbe $E_{E,c,d}$ se réécrit sous la forme

$$(X^2 + Y^2)Z^2 = c^2(Z^4 + dX^2Y^2).$$

Un point $(X_1 : Y_1 : Z_1)$ avec $Z_1 \neq 0$ vérifiant cette équation, correspond au point affine $(X_1/Z_1, Y_1/Z_1)$. L'élément neutre est le point $(0 : c : 1)$ et l'opposé d'un point $(X_1 : Y_1 : Z_1)$ est $(-X_1 : Y_1 : Z_1)$.

Les formules suivantes permettent de calculer la somme $(X_3 : Y_3 : Z_3)$ des points

$(X_1 : Y_1 : Z_1)$ et $(X_2 : Y_2 : Z_2)$.

$$\begin{aligned} A &= Z_1 Z_2, \\ B &= A^2, \\ C &= X_1 X_2, \\ D &= Y_1 Y_2, \\ E &= dCD, \\ F &= B - E, \\ G &= B = E, \\ X_3 &= AF((X_1 + Y_1)(X_2 + Y_2) - C - D), \\ Y_3 &= AG(D - C), \\ Z_3 &= cFG. \end{aligned} \tag{2.23}$$

On obtient un coût de $10M + 1S + 2B$.

Dans [BL07b], Bernstein et Lange ont introduit un nouveau système de coordonnées. Un point $(X_1 : Y_1 : Z_1)$ correspond au point affine $(Z_1/X_1, Z_1/Y_1)$ avec $X_1, Y_1, Z_1 \neq 0$. L'équation de la courbe devient

$$(X_1^2 + Y_1^2)Z_1^2 = X_1^2 Y_1^2 + dZ_1^4.$$

Les formules suivantes permettent de calculer la somme $(X_3 : Y_3 : Z_3)$ de deux points $(X_1 : Y_1 : Z_1)$ et $(X_2 : Y_2 : Z_2)$.

$$\begin{aligned} A &= Z_1 Z_2, \\ B &= dA^2, \\ C &= X_1 X_2, \\ D &= Y_1 Y_2, \\ H &= C - D, \\ I &= (X_1 + Y_1)(X_2 + Y_2) - C - D, \\ X_3 &= (E + B)H, \\ Y_3 &= (E - B)I, \\ Z_3 &= AHI. \end{aligned} \tag{2.24}$$

On obtient un coût de $9M + 1S + 1B$. Cela permet donc de gagner une multiplication par rapport aux formules précédentes.

Dans [BBJLP08], les auteurs introduisent les courbes tordues d'Edwards.

Définition 2.3.2
Soit K un corps tel que $char(K) \neq 2$. Soient a et d deux éléments non-nuls de K. La courbe tordue d'Edwards avec les coefficients a et d est la courbe

$$E_{E,a,d}/K : ax^2 + y^2 = 1 + dx^2y^2.$$

Avec cette définition, une courbe d'Edwards peut être vue comme une courbe tordue d'Edwards avec $a = 1$. La somme de deux points (x_1, y_1), (x_2, y_2) de $E_{E,d}$ donnés en coordonnées affines est :

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + y_1 x_2}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right).$$

Ces formules sont unifiées, et complètes si d n'est pas un carré dans K [BBJLP08].

En coordonnées homogènes projectives, un point (x, y) sur $E_{E,a,d}/K$, correspond au point projectif $(x : y : 1)$ et pour $Z \neq 0$, un point $(X : Y : Z)$ correspond au point affine $(X/Y, X/Z)$. L'élément neutre est le point $(0 : 1 : 1)$ et l'opposé d'un point $(X_1 : Y_1 : Z_1)$ est $(-X_1 : Y_1 : Z_1)$. Les points projectifs vérifient l'équation

$$aX^2Z^2 + Y^2Z^2 = Z^4 + dX^2Y^2.$$

Les formules suivantes permettent de calculer la somme $(X_3 : Y_3 : Z_3)$, des points $(X_1 : Y_1 : Z_1)$ et $(X_2 : Y_2 : Z_2)$.

$$\begin{aligned} X_3 &= Z_1 Z_2 (X_1 Y_2 + Y_1 X_2)(Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2), \\ Y_3 &= Z_1 Z_2 (Y_1 Y_2 - a X_1 X_2)(Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2), \\ Z_3 &= (Z_1^2 Z_2^2 - d X_1 X_2 Y_1 Y_2)(Z_1^2 Z_2^2 + d X_1 X_2 Y_1 Y_2). \end{aligned} \tag{2.25}$$

Elles sont également unifiées, et complètes si d n'est pas un carré dans K [BBJLP08]. On obtient un coût de $10M + 1S + 2B$.

Dans [HWCD08], les auteurs ont également travaillé sur les tordues d'Edwards en donnant des formules encore plus performantes en rajoutant une coordonnée. Un point $(X : Y : T : Z)$ avec $Z \neq 0$ correspond au point affine étendu $(X/Z, Y/Z, T/Z)$ où la troisième coordonnée correspond au produit xy. Dans ce système de représentation, l'élément neutre est le point $(0 : 1 : 0 : 1)$ et l'opposé d'un point $(X_1 : Y_1 : T_1 : Z_1)$ est $(-X_1 : Y_1 : -T_1 : Z_1)$. Les formules suivantes permettent de calculer la somme $(X_3 : Y_3 : T_3 : Z_3)$ des points $(X_1 : Y_1 : T_1 : Z_1)$ et $(X_2 : Y_2 : T_2 : Z_2)$.

$$\begin{aligned} X_3 &= (X_1 Y_2 + Y_1 X_2)(Z_1 Z_2 - d T_1 T_2), \\ Y_3 &= (Y_1 Y_2 - a X_1 X_2)(Z_1 Z_2 + d T_1 T_2), \\ T_3 &= (Y_1 Y_2 - a X_1 X_2)(X_1 Y_2 + Y_1 X_2), \\ Z_3 &= (Z_1 Z_2 - d T_1 T_2)(Z_1 Z_2 + d T_1 T_2). \end{aligned} \tag{2.26}$$

On obtient un coût de $9M + 2B$ pour des formules qui sont à la fois unifiées et complètes si d n'est pas un carré dans K et si a est un carré dans K.

En résumé, le tableau 2.4 donne les différents coûts que l'on peut obtenir sur les courbes tordues d'Edwards ou non.

Edwards	3 coordonnées [BL07a]	$10M + 1S + 2B$
Edwards	3 coordonnées [BL07b]	$9M + 1S + 1B$
Tordue d'Edwards	3 coordonnées [BBJLP08]	$10M + 1S + 2B$
Tordue d'Edwards	4 coordonnées [HWCD08]	$9M + 2B$

Tableau 2.4 – Coûts de l'addition unifiée sur les courbes d'Edwards (tordues ou non)

2.4 Comparaison des différentes formes

Certains standards cryptographiques, comme [FIPS-186-4], préconisent l'utilisation de courbes présentant un « petit » co-facteur h. En pratique, les contraintes peuvent différer d'un standard à un autre. Par exemple, la première version de [SEC1] (2000) imposait un $h \leqslant 4$, alors que celle de 2009 préconisent plutôt $h \leqslant 2^{\alpha}$ pour un niveau de sécurité α. Dans la suite, nous considérons que la contrainte est $h \leqslant 4$. Dans ce contexte, quelle forme de courbe et quelle représentation utiliser afin d'atteindre les meilleures performances tout en garantissant des formules unifiées et complètes pour l'addition ?

Le choix doit tout d'abord prendre en compte les critères imposés sur le cardinal de la courbe que nous avons vu dans les sections précédentes. Le tableau 2.5 présente une première classification. Nous renvoyons à [JQ01b ; Hıs10] pour plus de détails sur les courbes Hessiennes.

co-facteur h	Forme
1	Weierstrass
2	Quartique de Jacobi étendue
3	Hessienne généralisée
4	Quartique de Jacobi ou tordue d'Edwards

Tableau 2.5 – Formes de courbes elliptiques sur $\mathbb{F}_p$ utilisables en fonction du co-facteur

Cependant, les différences de coûts pour l'addition entre les formes de courbe peuvent justifier dans certains cas le passage sur une extension de corps. Prenons par exemple le cas d'une courbe sous forme Weierstrass de cardinal premier définie sur $\mathbb{F}_p$. Afin d'utiliser la représentation sous forme quartique de Jacobi ou tordue d'Edwards, il est nécessaire de travailler sur l'extension $\mathbb{F}_{p^3}$ afin de récupérer des points de torsion. Le passage sous l'une de ces formes est donc intéressant si et seulement si le coût de l'addition de point sur $\mathbb{F}_{p^3}$ est meilleur que celui sur $\mathbb{F}_p$ pour la forme Weierstrass. Si l'on simplifie le coût pour une addition pour la forme quartique de Jacobi sur $\mathbb{F}_{p^3}$ en prenant en compte que la multiplication, on obtient $10M_3$ où M_3 représente le coût d'une multiplication sur $\mathbb{F}_{p^3}$. Pour une addition Weierstrass unifiée et complète, nous avons un coût de $12M$. Par conséquent, si $M_3 < 1,2M$, il est plus avantageux de travailler sur l'extension.

Nous pouvons également remarqué que pour le cas $h = 4$, deux choix sont a priori possibles, l'utilisation d'une forme quartique de Jacobi ou celle de tordue d'Edwards. Même si les formules d'additions sont légèrement meilleures dans le cas d'une tordue d'Edwards, la proportion de courbes pouvant se mettre sous forme quartique de Jacobi est plus grande (voir section 2.2.4).

2.5 Conclusion

Nous avons revu dans ce chapitre deux formes de courbes elliptiques, les courbes quartiques de Jacobi étendues et celles d'Edwards, permettant d'utiliser des formules unifiées pour les opérations sur les points. Cette propriété leur offre une contre-mesure intrinsèque face à certaines attaques par canaux cachés. De plus, les formules sont très performantes par rapport aux algorithmes à notre disposition pour des systèmes de représentations plus classiques.

En comparant différentes formes de courbes en fonction des contraintes sur le co-facteur h ou l'efficacité des formules d'addition, nous avons constaté que le choix de la représentation utilisé pouvait différer en fonction du contexte. Si la courbe n'est pas imposée et qu'il n'y a pas de contraintes fortes sur le co-facteur, comme par exemple $h = 1$, on peut se concentrer sur les coûts des opérations et les possibilités de parallélisation et choisir la représentation la plus adaptée. Si la courbe est imposée, il faut se poser la question du passage sur une extension qui permettrait d'utiliser malgré tout une autre forme.

Nous avons décidé d'implanter dans notre bibliothèque logicielle les formes suivantes :

- Weierstrass avec coordonnées projectives et jacobiennes,
- Weierstrass avec formules unifiées,
- quartique de Jacobi étendue,
- Tordue d'Edwards.

Cependant, nous nous sommes concentrés sur les courbes quartiques de Jacobi pour notre cible matérielle et nous avons également travaillé sur le cas des calculs sur l'extension cubique afin de d'examiner l'intérêt de ces courbes même dans le cas où la conversion nécessite un travail sur cette extension.

Chapitre 3

Génération de courbes elliptiques

Sommaire

Dans ce chapitre, il sera question de la problématique liée à la génération des courbes elliptiques. En effet, certains standard préconisent des courbes qui pourraient être utilisées dans les protocoles cryptographiques [FIPS-186-4]. Cependant, l'existence ou non de failles dans ces courbes, la non-justification des graines utilisées lors de la génération et la volonté d'avoir des courbes dont on pourrait reproduire la construction ont motivé la recherche dans ce domaine. De plus, cela permet d'augmenter la confiance que les utilisateurs peuvent avoir dans les courbes générées.

Il existe deux méthodes pour générer des courbes elliptiques :

- la théorie de la multiplication complexe permet de générer directement une courbe sur un corps fini ayant un certain anneau d'endomorphisme de discriminant D tant que D n'est pas « trop grand ». Des propriétés spécifiques sont alors préservées lors de la construction [BLS03].
- la méthode dite « aléatoire » consiste à générer une courbe elliptique de manière aléatoire puis de vérifier si elle respecte les critères de sécurité qui assurent qu'elle résistera à la plupart des attaques connues. La complexité de cette méthode réside dans le fait qu'elle nécessite un calcul de comptage de points sur la courbe et parfois des factorisations de grands entiers.

Deux problématiques apparaissent lors de la génération aléatoire des courbes elliptiques :

1. Déterminer les tests à effectuer sur la courbe afin de vérifier qu'elle est robuste par rapport aux attaques connues,
2. Générer aléatoirement les paramètres de la courbe (méthode de génération, graine à utiliser, etc).

Concernant les tests à effectuer, on peut parcourir l'ensemble des attaques connues et mettre en place les contre-mesures proposées. Par exemple, dans [FPRE15 ; LM10], les auteurs discutent des différents critères de sécurité qu'il faudrait mettre en place afin de valider la robustesse d'une courbe. Les notions de certificat de validité ou plus simplement de données de validation permettent a posteriori de vérifier tous les critères sans avoir à refaire des calculs coûteux comme la factorisation.

Pour la méthode de génération des paramètres, différentes stratégies sont possibles. On peut :

- utiliser un algorithme prouvé de génération basée sur des fonctions de hachage [X9.62 ; FIPS-186-4],
- utiliser un autre algorithme déterministe pour lequel on fournit une description [LM10 ; BDFGLR16].

Cependant il est reste à décider quelle graine utiliser. De plus, il faut se demander com-

ment on peut justifier son utilisation. Cette question est abordée dans [BDFGLR16 ; LW15]. Dans [BDFGLR16], les auteurs ont proposé une source d'entropie qui est non seulement imprédictible, non biaisée, observable publiquement mais également vérifiable dans le futur. Ils utilisent pour ce faire des résultats de loteries nationales. Dans [LW15], ils utilisent un protocole en ligne où un groupe de personnes peuvent se mettre d'accord sur un nombre pseudo-aléatoire.

La première section rappellera les différents types d'attaques sur les courbes elliptiques ainsi que leurs contre-mesures. La distinction sera faite entre les attaques génériques et les attaques spécifiques. Les attaques basées sur les couplages seront expliquées plus en détail. De plus, les attaques liées à l'indice de classe seront également discutées. Même s'il n'existe pas encore de réalisation pratique de ces attaques, certains articles considèrent qu'ils faut malgré tout les prendre en compte lors de la définition de critères de sécurité pour une courbe elliptique.

La seconde section présentera la méthode de génération de Brainpool qui fait l'objet d'un standard. Cette méthode permet de générer des courbes elliptiques aléatoires qui respectent des critères de sécurité et de performance. La version d'origine ne traite que les courbes sous forme de Weierstrass mais il est possible de l'adapter à d'autres systèmes de représentation de courbes. En particulier, la version adaptée aux courbes quartiques de Jacobi étendues et Edwards sera analysée dans cette section.

La dernière section détaillera l'implantation logicielle qui a été réalisée. Celle-ci s'appuie sur la méthode de Brainpool mais elle permet également de générer des courbes quartiques de Jacobi étendues et Edwards. Les choix d'implantation seront discutés ainsi que les temps d'exécution obtenus pour la génération de différentes tailles de courbes.

3.1 Cryptanalyse en boîte noire

Les attaques peuvent être séparées en deux groupes, les attaques dites génériques qui sont des versions adaptées des algorithmes classiques, et les attaques spécifiques qui prennent en compte les propriétés des courbes elliptiques.

3.1.1 Méthodes de résolution génériques

Les attaques génériques considèrent les courbes elliptiques comme un groupe quelconque G. Afin de présenter ces attaques, le contexte suivant est défini : Soit G un sous-groupe cyclique de points de E(K) et soient $P, Q \in G$. Les attaques consistent à résoudre l'équation $Q = kP$ en assumant que l'ordre N de G est connu et que P est un générateur

de G.

Pas de bébés - Pas de géants

La méthode dite « Pas de bébés - Pas de géants » a été proposée par D.Shanks [Sha71]. Étant donné qu'elle nécessite dans le pire cas $\sqrt{N}$ opérations et $\sqrt{N}$ de stockage, elle n'est utilisée que sur des tailles modérées de N. L'attaque se déroule comme suit :

1. Choisir un entier $m \geqslant \sqrt{N}$ et calculer mP.
2. Construire et stocker la liste des points iP pour $0 \leqslant i < m$.
3. Calculer les points $Q - jmP$ pour $j = 0, 1, \ldots, m-1$ jusqu'à avoir une correspondance avec un élément de la liste stockée.
4. Si $iP = Q - jmP$, alors $Q = kP$ avec $k \equiv i + jm \mod N$.

Il est possible d'améliorer le coût de cette attaque en calculant seulement les points iP pour $0 \leqslant i \leqslant m/2$ et en vérifiant $Q - jmP = \pm iP$. Le désavantage de cette méthode est la quantité de stockage qui est requise. D'autres attaques ont donc été développées afin de réduire cette complexité.

Les méthodes de ρ et λ Pollard

La méthode de ρ Pollard et sa généralisation (λ) ont été introduites par J.M. Pollard [Pol78]. Le temps d'exécution est similaire à celui de l'attaque « Pas de bébés - Pas de géants » mais elles nécessitent moins de stockage.

La base de l'attaque est une fonction $f : G \to G$ qui est utilisée comme fonction d'itération. En effet, en partant d'un point initial P_0, les itérations $P_{i+1} = f(P_i)$ sont calculées. G étant un ensemble fini, il existe des indices i_0 et j_0 tels que $P_{i_0} = P_{j_0}$. De ce fait, en continuant les calculs des P_i, une séquence périodique apparaît de période $j_0 - i_0$ (ou un diviseur de $j_0 - i_0$).

On peut représenter graphiquement ce processus (Figure 3.1) et la figure obtenue ressemble à la lettre grecque ρ qui donne ainsi son nom à l'algorithme. Si la fonction f est choisie de manière aléatoire, il possible d'espérer une correspondance avec j_0 en $O(\sqrt{N})$ opérations.

Le coût de stockage de l'attaque peut varier en fonction de l'implantation réalisée. En effet, une implantation naïve va nécessiter $\sqrt{N}$ de stockage. Or, comme l'indique R.W. Floyd, il est possible d'obtenir un meilleur coût au prix d'un peu plus de calculs (voir [Was08, chapitre 5]).

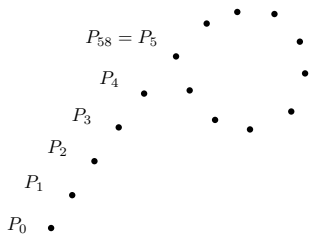


Figure. 3.1 – Méthode de ρ Pollard

La version généralisé λ de l'attaque suit le même procédé que la version ρ mais utilise plusieurs point de départ. Les étapes d'itérations peuvent être calculées en parallèle afin d'améliorer le coût de l'attaque.

Outre l'amélioration du coût de stockage, les méthodes ρ et λ Pollard diffèrent de la méthode « Pas de bébés - Pas de géants » sur le principe de terminaison. En effet, la méthode Pas de bébés - Pas de géants est déterministe et donc il est certain que l'attaque se terminera en un temps $O(\sqrt{N})$. À la différence, les méthodes de Pollard sont probabilistes, il y a une grande probabilité qu'elles terminent dans les temps prévus mais ce n'est pas une certitude.

Méthode Pohlig-Hellman

Cette méthode nécessite la connaissance de la factorisation de N, soit donc $N = \prod_i q_i^{e_i}$. L'idée de base de l'attaque est de trouver $k(\mathrm{mod} q_i^{e_i})$ pour chaque i et d'utiliser le théorème des Restes Chinois pour retrouver $k \mod N$.

Afin de calculer $k(\mathrm{mod} q_i^{e_i})$, une expansion en base q_i est utilisée et k s'écrit alors comme suit :

$$k = k_0 + k_1 q_i + k_2 q_i^2 + \dots.$$

Pour chaque i, la procédure d'évaluation est la suivante

1. Calculer $T = \{j(\frac{N}{q_i}P) | 0 \leqslant j \leqslant q_i - 1\}$.
2. Calculer $k_0 = \frac{N}{q_i}Q$.
3. Si $e_i = 1$, c'est terminé sinon il faut continuer.
4. Soit $Q_1 = Q - k_0 P$.
5. Calculer $k_1 = \frac{N}{q_i^2}Q_1$.
6. Si $e_i = 2$, alors c'est terminé sinon il faut continuer.
7. Les éléments $k_0, k_1, \dots, k_{r-1}$ et $Q_1, \dots, Q_{r-1}$ ont été calculés.
8. Soit $Q_r = Q_{r-1} - k_{r-1} q_i^{r-1} P$.

9. Déterminer k_r tel que $\frac{N}{q_i^{r+1}} Q_r = k_r(\frac{N}{q_i} P)$.
10. Si $r = e_i - 1$ alors c'est terminé sinon, il faut retourner à l'étape(7).

Cette procédure permettant de retrouver tous les k_i , il est donc possible d'écrire :

$$k \equiv k_0 + k_1 q_i + \cdots + k_{e_i-1} q_i^{e_i-1} (\mathrm{mod}\, q_i^{e_i}).$$

Cette méthode est efficace si les nombres premiers diviseurs de N sont petits. Dans le cas où N admet un très grand diviseur premier q, il est alors difficile de calculer l'ensemble T. Il est possible de calculer k_i sans calculer tous les éléments de T, cependant cela revient à calculer le problème du logarithme discret dans le groupe généré par $(N/q)P$ qui est d'ordre q. Ainsi, si q est de « même taille » que N (par exemple $q = N$ ou $q = N/2$) alors cette méthode se révèle inefficace. Une contre-mesure de cette attaque est donc que l'ordre N du groupe ne contienne que de grands facteurs premiers.

En effet, si N contient malgré tout quelques petits premiers diviseurs, il est possible d'utiliser la méthode de Pohlig-Hellman afin de récupérer des informations partielles sur k. Il faut donc choisir avec précaution le groupe utilisé et par conséquent le point générateur.

En résumé les attaques dites génériques agissent directement sur le groupe de points considéré mais il existe différentes contre-mesures pour ces attaques qui sont rappelées dans le tableau 3.1. Cependant, même en mettant en place ces contre-mesures, il est possible d'adopter une autre stratégie et de réduire le problème du logarithme discret à une instance plus facile. Cela peut être réalisé en utilisant les propriétés des couplages.

Tableau 3.1 – Attaques génériques et leurs contre-mesures

Attaque	Coût en opérations	Coût en stockage	Contre-mesure
Pas de bébés - Pas de géants	$O(\sqrt{N})$	$O(\sqrt{N})$	grand N
ρ et λ Pollard	$O(\sqrt{N})$	$O(1)$	grand N
Pohlig-Hellman	$O(\sqrt{\max(q_i)})$	$O(1)$	travailler dans un groupe d'ordre premier ou contenant uniquement de grands facteurs premiers.

3.1.2 Attaques basées sur les couplages de Tate et de Weil

Les attaques basées sur les couplages de Tate et de Weil ont pour but de transférer le calcul du logarithme discret sur une extension de « petit degré » du corps de base. Dans la suite, E est considérée comme étant une courbe elliptique sur $\mathbb{F}_q$ avec q une puissance d'un nombre premier p. P et Q sont deux points de $E(\mathbb{F}_q)$. Soit N l'ordre de P. Il est supposé que $\gcd(N, q) = 1$. Le problème consiste à retrouver k, s'il existe, tel que $Q = kP$. Pour plus de détails sur les couplages de Weil et Tate voir [Was08, sections 3.3 et 3.4].

3.1.2.1 L'attaque MOV

Cette attaque tient son nom de ses auteurs, à savoir Menezes, Okamoto et Vanstone [MOV+93], qui ont utilisé le couplage de Weil afin de transférer le problème du logarithme discret de $E(\mathbb{F}_q)$ à $\mathbb{F}_{q^m}^{\times}$. La propriété principale du couplage de Weil qui est utilisée pour l'attaque est la suivante : Si $\gcd(q, N) = 1$ et si $\{S, T\}$ est une base de $E[N]$, alors le couplage de Weil $e_N(S, T)$ est une racine N^{e} de l'unité [Was08, Corollaire 3.10].

La preuve du lemme suivant donne la base de l'attaque.

Lemme 3.1.1
Il existe k tel que $Q = kP$ si et seulement si $NQ = \mathcal{O}$ et si le couplage de Weil vérifie $e_N(P, Q) = 1$.

Démonstration. Si $Q = kP$ alors $NQ = NkP = \mathcal{O}$. De plus, les propriétés du couplage impliquent également que

$$e_N(P, Q) = e_N(P, kP) = e_N(P, P)^k = 1.$$

Le premier sens de l'implication est ainsi démontré. Si maintenant $NQ = \mathcal{O}$, alors Q est un point de N-torsion. Étant donné que $\gcd(q, N) = 1$, alors ils possible d'écrire $E[N] \simeq \mathbb{Z}/N\mathbb{Z} \oplus \mathbb{Z}/N\mathbb{Z}$. Soit un point R tel que $\{P, R\}$ soit une base de $E[N]$. Il est possible d'écrire Q dans cette base, $Q = aP + bR$ avec a, b des entiers. $e_N(P, R)$ étant une racine N^{e} de l'unité ζ, s'il est supposé que $e_N(P, Q) = 1$, alors il est possible d'écrire

$$1 = e_N(P, Q) = e_N(P, aP + bR) = e_N(P, P)^a e_N(P, R)^b = e_N(P, R)^b = \zeta^b.$$

Par conséquent $b \equiv 0 (\bmod N)$ et donc $bR = \mathcal{O}$. Cela implique que $Q = aP$. □

L'attaque procède comme suit. Tout d'abord, il faut choisir un entier m tel que

$$E[N] \subseteq E(\mathbb{F}_{q^m}).$$

La définition de $E[N]$ implique l'existence d'un tel entier. D'autre part, le groupe μ_N des racines N^{e} de l'unité étant inclus dans $\mathbb{F}_{q^m}$, tous les calculs auront lieu dans $\mathbb{F}_{q^m}$. L'algorithme de l'attaque est le suivant :

1. Choisir aléatoirement un point T dans $E(\mathbb{F}_{q^m})$.
2. Calculer l'ordre M de ce point.
3. Soient $d = \gcd(M, N))$ et $T_1 = (M/d)T$. L'ordre d de T_1 divise donc N, et par conséquent $T_1 \in E[N]$.
4. Calculer $\zeta_1 = e_N(P, T_1)$ et $\zeta_2 = e_N(Q, T_1)$. Il s'en suit que $\zeta_1, \zeta_2 \in \mu_d \subseteq \mathbb{F}_{q^m}^{\times}$.
5. Résoudre le problème du logarithme discret $\zeta_2 = \zeta_1^k$ dans $\mathbb{F}_{q^m}^{\times}$. Cela permet de retrouver $k \pmod d$.
6. Il faut réitérer ce processus avec des points aléatoires T jusqu'à ce que le plus petit commun multiple des différents d soit N. Cela détermine donc $k \pmod N$.

L'efficacité de cette attaque réside dans la taille de l'entier m. Si m est grand alors la résolution du logarithme discret dans le groupe $\mathbb{F}_{q^m}^{\times}$ (d'ordre $q^m - 1$) est aussi difficile que la résolution du problème d'origine dans le plus petit sous-groupe de $E(\mathbb{F}_q)$ (d'ordre environ p). Cependant, on peut trouver des courbes pour lesquelles m est petit.

Dans le cas d'une courbe super-singulière, si la trace t de l'endomorphisme de Frobenius est égal à 0, il est possible de prouver que $m = 2$, c'est-à-dire que $E[N] \subseteq E(\mathbb{F}_{q^2})$ (voir [Was08, proposition 5.3]). Le problème du logarithme discret est ainsi transféré dans $\mathbb{F}_{q^2}^{\times}$. Dans le cas où la courbe est super-singulière mais que $t \neq 0$, l'attaque est encore possible cependant la valeur de m est plus grande ($m = 3, 4, 6$) ([MOV+93]). Ces valeurs permettent malgré tout d'accélérer les calculs du logarithme discret. Une contre-mesure naturelle à cette attaque est donc d'avoir une grande valeur de m. Ce qui sera a priori le cas si la courbe est « générée aléatoirement ».

3.1.2.2 L'attaque de Frey-Rück

Frey et Rück [FR94] ont montré qu'il était possible d'utiliser le couplage de Tate-Lichtenbaum τ_n afin de résoudre le problème du logarithme discret. Le principe général de l'attaque est le même que celui pour l'attaque MOV. Il faut simplement vérifier les conditions pour lesquelles on retrouve la racine primitive de l'unité. Pour ce faire, le lemme suivant est utilisé.

Lemme 3.1.2
Soit ℓ un nombre premier tel que $\ell | (q-1)$, $\ell | \#E(\mathbb{F}_q)$ et $\ell^2 \nmid \#E(\mathbb{F}_q)$. Soit P un générateur de $E(\mathbb{F}_q)[\ell]$. Alors $\tau_\ell(P, P)$ est une racine ℓ^e de l'unité.

En considérant le contexte de ce lemme, si $Q = kP$, alors

$$\tau_\ell(P, Q) = \tau_\ell(P, P)^k = \zeta^k,$$

avec ζ une racine ℓ^e de l'unité. Cela permet donc de calculer $k(\mathrm{mod}\,\ell)$. Le transfert du logarithme discret sur $\mathbb{F}_q^\times$ est donc possible.

Une contre-mesure à cette attaque est donc d'être dans un contexte pour lequel l'ordre ℓ de P est un grand nombre premier qui ne divise pas $q-1$. Cependant, en réalité il faut aussi s'assurer que $q^m \not\equiv 1(\mathrm{mod}\,\ell)$ pour de petites valeurs de m car il serait possible de réaliser une attaque. En effet, même si ℓ ne divise pas $q-1$, il est possible d'étendre le corps de base à $\mathbb{F}_{q^m}$ où $\ell|(q^m-1)$ et ainsi réaliser l'attaque.

En résumé, les attaques basées sur les couplages permettent de transférer les problèmes du logarithme discret sur $E(\mathbb{F}_{q^m})$ à des instances sur $\mathbb{F}_{q^m}^\times$ quand $E[n] \subseteq E(\mathbb{F}_{q^m})$ où n est l'ordre de P. La complexité et les contres-mesures des attaques sont présentées dans le tableau 3.2.

Tableau 3.2 – Attaques basées sur les couplages et leurs contre-mesures

Attaque	Coût	Contre-mesure
MOV	sous-exponentiel	E n'est pas super-singulière
Frey-Rück	sous-exponentiel	$n \nmid (q^m-1)$ pour des « petites » valeurs de m et pas trop de petits premiers diviseurs.

3.1.3 Courbes anormales

Nous rappelons qu'une courbe anormale E vérifie $\#E(\mathbb{F}_q) = q$. Des méthodes concernant ces courbes ont été développées dans [Sem98], [SA98] et [Sma99]. Leur complexité est en temps polynomial. De ce fait, l'usage cryptographique de ces courbes est proscrit.

3.1.4 Relèvement canonique

Définition 3.1.3 ([Deu41])
Le relèvement canonique $\mathcal{E}$ d'une courbe elliptique ordinaire E définie sur $\mathbb{F}_q$, avec q puissance d'un nombre premier p, est une courbe elliptique sur $\mathbb{Q}_p$ qui vérifie :

- *la réduction de $\mathcal{E}$ modulo p est égale à E,*
- *le morphisme d'anneau $End(\mathcal{E}) \to End(E)$ induit par la réduction modulo p est un isomorphisme.*

Deuring [Deu41] a montré que le relèvement canonique existe toujours et qu'il est unique à isomorphisme près.

On peut utiliser ce relèvement canonique afin de calculer le logarithme discret [Gau03].

3.1.5 Influence de l'indice de classe

Les résultats présentés ici sont issus de l'article de Jao, Miller et Venkatesan [JMV09] qui fait le lien entre les graphes d'expansion et le problème du logarithme discret sur les courbes elliptiques.

Ces résultats ont pour base une observation de Galbraith [Gal99] qui montre qu'étant donnée une isogénie calculable efficacement entre des courbes E et E', alors il est possible de calculer le logarithme discret sur E en calculant celui sur E'. Il faut procéder de la manière suivante :
Soient $P, Q \in E$ et $\Phi : E \to E'$ une isogénie.

1. Calculer $\Phi(P)$ et $\Phi(Q)$.

2. Déterminer le logarithme discret x de $\Phi(Q)$ sur E' par rapport au générateur $\Phi(P)$.

3. L'équation $x\Phi(P) = \Phi(Q)$ détermine la solution pour x modulo le noyau de Φ. Si Φ est une isogénie de bas degré alors, elle est calculable efficacement et son noyau est petit.

Si Φ est de petit degré, elle est facilement calculable et possède un petit noyau. De plus, un théorème de Tate [Tat66] établit que deux courbes elliptiques sur $\mathbb{F}_q$ ont le même nombre de points si et seulement si elles sont isogènes. Le théorème garantit également l'existence d'une isogénie définie sur $\mathbb{F}_q$ pour des courbes qui appartiennent à la même classe d'équivalence.

Deux courbes sont dites de même niveau si et seulement si leurs conducteurs d'anneau d'endomorphisme sont égaux. Ils ont montré que pour des courbes de même niveau, une composition d'isogénies de petit degré existe [JMV09, Preuve du théorème 1.6]. Même si le degré de la composition est très élevé, la structure de celle-ci impose une réduction efficace entre les courbes auxquelles elle est liée. Pour des courbes de niveaux différents, ils définissent l'écart de conducteur comme étant le plus grand facteur premier à partir duquel les factorisations de c_E et $c_{E'}$ diffèrent. Cette notion permet de mesurer combien de niveaux séparent deux courbes E et E'. Par conséquent, un grand écart de conducteur pourrait rendre difficile la réduction entre les courbes E et E'.

3.2 La méthodologie Brainpool

La méthodologie Brainpool [LM10] propose des paramètres de courbes elliptiques définies sur des corps premiers qui peuvent être utilisées dans des protocoles cryptographiques. Elle donne également les méthodes qui ont été utilisées pour générer à la fois les corps de base et les courbes. Il permet ainsi de générer des courbes E définies par $y^2 = x^3 + Ax + B$ sur $\mathbb{F}_p$ avec p premier et donne également un générateur G d'ordre q.

3.2.1 Critères vérifiés

Les critères vérifiés sont de deux types : les critères de sécurité et les critères techniques.

Critères de sécurité

Ces critères ont été définis en fonction des attaques connues décrites dans la section précédente et dans l'optique de renforcer la confiance que l'on peut avoir dans les courbes générées.

- Soit $\ell = \min\{t | q \text{ divise } p^t - 1\}$. Vérifier que $(q-1)/\ell < 100$ afin de se prémunir des attaques utilisant les couplages de Weil et Tate. Cela garantit également que la courbe n'est pas super-singulière.
- Vérifier que la courbe n'est pas anormale.
- Vérifier que l'indice de classe de l'ordre maximal du quotient de l'anneau d'endomorphisme de E est plus grand que 10^7.
- L'ordre du groupe $\#E(\mathbb{F}_p)$ doit être premier.
- Les paramètres de la courbe doivent être générés de manière aléatoire en utilisant des graines dont l'utilisation a été justifiée.
- Une preuve de sécurité doit être donnée afin de fournir les informations permettant de vérifier que tous les critères de sécurité ont été respectés.

Critères techniques

Ces critères ont été pris en compte afin de répondre à certaines demandes commerciales

- Le nombre premier p doit vérifier $p \equiv 3 \mod 4$ afin de permettre une compression de point plus efficace.
- La courbe générée doit être isomorphe à une courbe E' avec $A' = -3 \mod p$. Cette propriété permet de bénéficier des avantages arithmétiques de ces courbes [BJ03b].
- Le nombre premier p ne doit pas avoir une forme particulière.
- Dans certains cas, la taille en bits de $\#E(\mathbb{F}_p)$ être plus grande que celle de p. Par conséquent, afin éviter des dépassements dans les implantations, il est demandé que $\#E(\mathbb{F}_p) < p$.
- B ne doit pas être un carré modulo p afin de garantir de bonnes propriétés lors de la compression de points. Ce critère rend également impossible l'attaque décrite dans [Gou03].

Avant de définir les algorithmes de génération, des fonctions auxiliaires doivent êtres introduites :

- *to_integer* qui convertit une chaîne de caractère binaire en un entier,
- *to_string* qui convertit un entier en une chaîne de caractère binaire,
- *update_seed* qui met à jour une graine de 160 bits (algorithme I.6).

Algorithme I.6 : update_seed: Algorithme permettant de mettre à jour une graine de 160 bits.

Données : `Une graine` s `de 160 bits.`
Résultat : `Une graine` s' `de 160 bits.`
1 $s = to\_integer(z)$.
2 $z = (z+1) \mod 2^{160}$.
3 $s' = to\_string(z)$.

3.2.2 Algorithme de génération des nombres premiers

L'algorithme de génération de nombres premiers est similaire aux procédures données dans [FIPS-186-4, Annexe 6.4] et [X9.62, section A.3.2]. C'est une version modifiée de la méthode de « recherche incrémentale » définie dans [ISO-18032, section 8.2.2]. Tout d'abord, il est nécessaire de définir une fonction de génération de nombre d'au plus L bits.

L'algorithme de génération de nombre premier est le suivant :

Algorithme I.7 : find_integer: Algorithme permettant de générer un entier de L bits à partir d'une graine de 160 bits.

Données : Une graine s de 160 bits.
Résultat : Un entier L non nul et un entier x d'au plus L bits.

1 $v = floor((L-1)/160)$ et $w = L - 160 * v$
2 $h = \text{SHA-1}(s)$
3 Soit h_0 la chaîne de caractère binaire obtenue en prenant les w bits les plus à droite de h.
4 $z = to\_integer(s)$
5 **pour** $i = 1$ à v **faire**
6 $\quad z_i = (z + i) \mod 2^{160}$
7 $\quad s_i = to\_string(z_i)$
8 $\quad h_i = \text{SHA-1}(s_i)$
9 Soit h la chaîne de caractère obtenue en concaténant $h_0, \ldots, h_v$ de gauche à droite.
10 $x = to\_integer(h)$

Algorithme I.8 : find_prime: Algorithme permettant de générer un nombre premier de L bits à partir d'une graine de 160 bits.

Données : Un entier L non nul et une graine s de 160 bits.
Résultat : Un nombre premier p de L bits.

1 $c = find\_integer(s)$.
2 Soit p le plus petit nombre premier tel que $p \geqslant c$ et $p = 3 \mod 4$.
3 $z = to\_integer(s)$.
4 Si $2^{(L-1)} \leqslant p \leqslant 2^L - 1$ alors stop.
5 $s = update\_seed(s)$ et retourner à l'étape 1.

Dans le cadre des courbes qui apparaissent dans la méthodologie Brainpool, les graines utilisées ont été obtenues en prenant les 7 sous-chaînes de caractères de 160 bits extraites de $R = floor(\pi 2^{1120})$. La constante $1120 = 160 * 7$ permet de garantir la possibilité d'extraire un nombre suffisant de graines.

3.2.3 Algorithme de génération de la courbe

L'algorithme de génération des paramètres de la courbe est similaire aux procédures données dans [FIPS-186-4, Annexe 6.4] et [X9.62, section A.3.2]. Il suppose que le nombre premier p a été préalablement généré en utilisant la méthode décrite précédemment. Il est rappelé dans l'algorithme I.9.

Algorithme I.9 : Génération de courbes selon le standard Brainpool

Données : L un entier non nul, une graine s de 160 bits et un nombre premier p de L bits.

Résultat : Les paramètres A, B de la courbe ainsi qu'un générateur G.

1 $A = find\_integer2(s)$.
2 Si $-3 = AZ^4 \mod p$ n'a pas de solutions alors $s = update\_seed(s)$ et retourner à l'étape 1.
3 Calculer une solution Z de $-3 = AZ^4 \mod p$.
4 $s = update\_seed(s)$.
5 $B = find\_integer2(s)$.
6 Si B est un carré modulo p alors $s = update\_seed(s)$ et retourner à l'étape 5.
7 Si $4A^3 + 27B^2 = 0 \mod p$ alors $s = update\_seed(s)$ et retourner à l'étape 5.
8 Vérifier que la courbe elliptique obtenue remplit tous les critères définis précédemment. Si ce n'est pas le cas, alors $s = update\_seed(s)$ et retourner à l'étape 1.
9 $s = update\_seed(s)$. $k = find\_integer2(s)$.
10 Trouver les points Q et $-Q$ ayant les plus petites coordonnées x dans $E(\mathbb{F}_p)$. Choisir aléatoirement l'un de ces points comme P.
11 Calculer le générateur $G = kP$.

La fonction de génération de nombre utilisée, $find\_integer2$ (voir algorithme I.10), est semblable à celle définie précédemment mais ne permet de générer que des entiers d'au plus $L - 1$ bits. Cela permet d'éviter un test de comparaison avec p qui lui à L bits.

Les deux algorithmes de génération de nombres premiers ou non, utilisent SHA-1 comme fonction de hachage. Cependant, cette fonction est connue pour être sujette aux collisions et de nombreuses attaques ont été publiées. De plus, les certificats ont également initié une transition vers SHA-2 (ou SHA-256). Par exemple, on peut noter que depuis le 1er janvier 2016, les certificats de signature de code utilisant SHA-1 ne sont plus reconnus par Microsoft et au 1er janvier 2017, il est prévu que les certificats SSL utilisant SHA-1 ne soient plus reconnus également.

Remarque 3.2.1
Le point P n'est pas utilisé directement comme générateur afin prévenir les possibles vulnérabilités que cela pourrait entraîner face aux attaques par canaux cachés.

Dans le cadre des courbes qui apparaissent dans la méthodologie Brainpool, les graines utilisées ont été obtenues en prenant les 7 sous-chaînes de caractères de 160 bits extraites de $R = floor(e2^{1120})$ où e représente la constante d'Euler.

Algorithme I.10 : find_integer2: Algorithme permettant de générer un entier de L bits à partir d'une graine de 160 bits.

Données : Un entier L non nul et une graine s de 160 bits.
Résultat : Un entier x d'au plus $L-1$ bits.

1 $v = floor((L-1)/160)$ et $w = L - 160 * v - 1$
2 $h = \text{SHA-1}(s)$
3 Soit h_0 la chaîne de caractère binaire obtenue en prenant les w bits les plus à droite de h.
4 $z = to\_integer(s)$
5 **pour** $i = 1$ à v **faire**
6 $\quad z_i = (z + i) \bmod 2^{160}$
7 $\quad s_i = to\_string(z_i)$
8 $\quad h_i = \text{SHA-1}(s_i)$
9 Soit h la chaîne de caractère obtenue en concaténant $h_0,\ldots,h_v$ de gauche à droite.
10 $x = to\_integer(h)$

La même fonction de recherche incrémentale est utilisée pour :

- la recherche de a,
- passer de a à b,
- générer chaque bloc de 160 bits de a et b, concaténation de hashs successifs.

Cette redondance ainsi que le choix des graines dans cette méthodologie ont été discutés car l'on ne sait pas exactement pourquoi choisir les constantes e et π et pas d'autres, et cette fonction peut induire de la redondance dans certains bits de a et b [Ber+15].

3.2.4 Preuve de sécurité

Les données de validation doivent contenir :

- Les trois graines utilisées pour la génération de p, A et B.
- La factorisation de d.
- La forme quadratique qui permet de vérifier l'indice de classe.
- La factorisation de $q-1$.
- la valeur de $(q-1)/(\text{ordre de } p \mod q)$.
- Les coordonnées du point P utilisé lors de la construction.
- La graine utilisée pour générer k.

3.2.5 Adaptation à d'autres systèmes de représentation de courbes

Il est possible d'adapter l'algorithme de génération de courbe à d'autres systèmes de représentation de courbes elliptiques. Dans le contexte de la thèse, une adaptation aux courbes quartiques de Jacobi étendues a été réalisée.

La première distinction qui apparaît concerne les critères sur $\#E(\mathbb{F}_p)$. Dans le cas d'une courbe sous forme quartique de Jacobi, il est impossible d'avoir $\#E(\mathbb{F}_p)$ égale à un nombre premier puisqu'il existe des points de 2-torsion sur la courbe. Plus précisément, on peut au mieux obtenir $\#E(\mathbb{F}_p) = 4q'$ où q' est un nombre premier si $d = 1$, et on a un cofacteur de 2 dans le cas du modèle étendu ($d \neq 1$). De ce fait le groupe que l'on retiendra pour les protocoles cryptographiques sera d'ordre q'. Ainsi les critères qui concernent $\#E(\mathbb{F}_p)$ doivent être transposés à q'.

Il n'est pas obligatoire de générer la courbe tordue isomorphe à E telle que $A' = -3 \mod p$ car les courbes quartiques de Jacobi étendues présentent déjà des avantages arithmétiques. La conversion sous forme Weierstrass pourra malgré tout être fournie par souci de compatibilité entre les protocoles cryptographiques.

La méthode de génération adaptée est donnée par l'algorithme I.11. Il est possible d'en faire de même pour adapter l'algorithme de génération pour générer des courbes d'Edwards.

Algorithme I.11 : Génération de courbes quartiques de Jacobi selon le standard Brainpool adapté

Données : L un entier non nul, une graine s de 160 bits et un nombre premier p de L bits.

Résultat : Le paramètre A de la courbe ainsi qu'un générateur G.

1 $A = find\_integer2(s)$.
2 Si $A^2 = 1 \mod p$ alors $s = update\_seed(s)$ et retourner à l'étape 1.
3 $E' == to\_Weierstrass(A, p)$.
4 Si $-3 = A'Z^4 \mod p$ n'a pas de solutions alors $s = update\_seed(s)$ et retourner à l'étape 1.
5 Calculer une solution Z de $-3 = A'Z^4 \mod p$.
6 Si B' est un carré modulo p alors $s = update\_seed(s)$ et retourner à l'étape 1.
7 Vérifier si le cardinal de la courbe est égal à 4 fois un nombre premier. Si ce n'est pas le cas, alors $s = update\_seed(s)$ et retourner à l'étape 1.
8 Vérifier résistance aux attaques liées aux couplages Tate et Weil. Si ce n'est pas le cas, alors $s = update\_seed(s)$ et retourner à l'étape 1.
9 Vérifier la borne de l'indice de classe. Si ce n'est pas le cas, alors $s = update\_seed(s)$ et retourner à l'étape 1.
10 $s = update\_seed(s)$. $k = find\_integer2(s)$.
11 Trouver les points Q et $-Q$ ayant les plus petites coordonnées x dans $E'(\mathbb{F}_p)$. Choisir aléatoirement l'un de ces points comme P.
12 Calculer le générateur $Gw = kP$.
13 $G = to\_jq\_point(Gw)$.

3.3 Implantation logicielle

Nous avons implanté la méthode de Brainpool ainsi que ses adaptations en `C`, en utilisant la bibliothèque `PARI` [PARI]. Les tests réalisés lors de la génération d'une courbe sont les mêmes que ceux décrits précédemment, cependant le choix d'activation n'est pas imposé. Par exemple, le critère sur l'indice de classe peut ne pas être vérifié car il nécessite des factorisations et des recherches de plus grand diviseur sans facteur carré. On peut ainsi améliorer le temps d'exécution de l'algorithme tout sachant qu'il n'y a encore aucune attaque connue liée à ce critère.

Il est possible d'utiliser une interface ou de programmer directement la génération d'une courbe en spécifiant les options suivantes :

- la forme de la courbe,

- la forme du cardinal,
- la taille en bits,
- les graines à utiliser pour la génération,
- la fonction de hachage à utiliser,
- s'il faut faire le test de résistance aux attaques liées aux couplages Tate et Weil (oui ou non),
- s'il faut faire le test de borne l'indice de classe (oui ou non),
- le nom du fichier dans lequel on va écrire les données de la courbe,
- le nom du fichier dans lequel on va écrire les données de la courbe,
- le nom de la courbe.

```
mac@linux:~/generation$ obj/test_ecc_interface
Generation of elliptic curves
Form of curve :
0 : Weierstrass
1 : Jacobi Quartic
2 : Jacobi Quartic extended
3 : Edwards
4 : Twisted Edwards
1
Size in bits :
256
Hash Function :
0 : SHA-1
1 : SHA-256
1
Please enter the 160 bit string for the seed used to generate p (hexadecimal)
3243f6a8885a308d313198a2e03707344a409382
Please enter the 160 bit string for the seed used to random integer (hexadecimal)
2b7e151628aed2a6abf7158809cf4f3c762e7160
Test immunity to weil- and tate-pairing attacks? (y/n)y
Test class number bound? (y/n)y
File to write curve data (max 100 char)
curve_jq256.txt
File to write data validation (max 100 char)
curve_jq256_valid.txt
ID curve
curve_jq256
```

Dans l'annexe C, nous avons mis l'exemple d'une courbe quartique de Jacobi de 256 bits que nous avons générée avec notre implantation.

Les performances de la génération sont influencées par les tests à effectuer sur la courbe. La vérification de la borne de l'indice de classe nécessite une recherche de partie non carrée d'un entier. Pour de grandes tailles de courbes, ce calcul devient prépondérant dans le temps d'exécution de notre implantation. La probabilité que ce critère de sécurité ne soit pas respecté étant faible, on peut fournir les paramètres de la courbe avant la vérification complète et continuer le calcul par la suite.

Le tableau 3.3 présente les coûts de la génération de courbes quartiques de Jacobi pour différentes tailles. Nous avons activé tous les tests de vérification. Les tests ont été réalisés avec la version 2.8 de la bibliothèque `PARI/GP`, sur une machine munie d'un processeur Intel i7-4790, 3.4 GHz. Le compilateur utilisé est `gcc-5.1`.

bits	temps moyen
192	115 s
256	511 s
384	plusieurs heures
512	plusieurs jours

Tableau 3.3 – Performance de l'algorithme de génération de courbes quartiques de Jacobi

3.4 Conclusion

Dans ce chapitre, nous avons étudié les problématiques liées à la génération de courbes elliptiques pouvant être utilisées dans des protocoles cryptographiques. Malgré l'existence de courbes définies par différents standards, certains s'interrogent sur la méthode ou les graines utilisées lors de la génération. Nous avons tout d'abord rappelé différentes méthodes de résolution sur le logarithme discret sur les courbes elliptiques ainsi que les contre-mesures associées.

Notre implantation s'appuie sur la méthodologie Brainpool. Nous avons modifié les procédures afin d'apporter le support des formes quartiques de Jacobi étendues et Edwards. En prenant en compte les remarques concernant la génération aléatoire des nombres, nous avons décidé de proposer différentes alternatives qui permettent de customiser l'algorithme. D'autres fonctions de hachage devraient être également implantées afin d'offrir un maximum de possibilité à l'utilisateur. Le choix des graines à utiliser est spécifié au moment de la génération. Certaines vérifications des critères de sécurité sont optionnelles et doivent être activées par l'utilisateur.

L'évolution de notre implantation pourra se faire à différents niveaux :

- Amélioration des performances liées à la factorisation,
- Rajout du fonction de génération de graines,
- Utilisation d'autres algorithmes de génération,
- Rajout de vérifications sur l'indice de classe ou la forme de $q - 1$.

Chapitre 4

Arithmétique modulaire

Sommaire

Les opérations sur les courbes elliptiques définies sur les corps finis reposent sur l'arithmétique modulaire du corps de base. Selon la représentation choisie pour la courbe, l'opération la plus coûteuse est soit la multiplication, soit la division. Néanmoins, l'écart de coût entre addition et multiplication peut varier selon les cibles. Par exemple, pour un processeur Intel Skylake, le rapport de cycles d'horloge moyens entre la multiplication et l'addition est de 4 pour des opérandes de 64 bits [Fog16]. De plus, les performances sont largement influencées par la manière dont les éléments du corps de base sont représentés.

Mise à part les opérations classiques que sont l'addition, le doublement et la multiplication scalaire, les couplages font également intervenir des calculs modulaires. Cependant, ils ne sont pas réalisés sur le corps de base mais sur des extensions de celui-ci. De plus, pour la conversion de forme entre Weierstrass et quartique de Jacobi étendue, si il on ne trouve pas de points de 2-torsion sur $\mathbb{F}_p$, il est nécessaire de travailler sur l'extension cubique.

Nous allons donc dans ce chapitre, analyser les différentes implantations possibles pour l'arithmétique de $\mathbb{F}_p$ en fonction du contexte des courbes elliptiques. Nous verrons par la suite la construction des tours extensions. Nous présenterons également les opérations de la multiplication et d'élévation au carré, ainsi que leurs versions parallélisées, sur les extensions quadratiques et cubiques. Pour finir, nous donnerons l'exemple de l'extension $\mathbb{F}_{p^6}$ comme application de la méthode de construction de tour.

4.1 Arithmétique modulaire sur $\mathbb{F}_p$

Afin de représenter les éléments de $\mathbb{F}_p$, l'isomorphisme entre $\mathbb{F}_p$ et $\mathbb{Z}/p\mathbb{Z}$ est utilisé. Une classe d'équivalence est par conséquent associée à un unique entier dans $[0, p-1]$ appartenant à celle-ci. Il est possible d'utiliser d'autres représentations en changeant d'intervalle.

Les opérations modulaires vont faire intervenir des opérations sur des entiers de grande taille dans le contexte des nombres p utilisés en cryptographie. Différentes approches peuvent être choisies afin d'accélérer ces calculs. Il est possible d'utiliser un système de représentation par les restes (RNS [1])[BDK01] qui consiste à utiliser une base de petits nombres premiers qui vont servir à représenter un élément de $\mathbb{Z}/p\mathbb{Z}$ par ses résidus dans celle-ci. Cette méthode permet de paralléliser de manière efficace les opérations modulaires cependant les conversions entre les différentes représentations peuvent être coûteuses [Eyn15].

Nous allons ici considérer une représentation classique en travaillant directement sur

1. Residue Number System

les grands entiers. Avant d'étudier l'arithmétique modulaire, nous allons présenter différents algorithmes permettant de réaliser une arithmétique multi-précision efficace sur des cibles logicielles et matérielles.

4.1.1 Arithmétique multi-précision

L'arithmétique multi-précision prend en charge les opérations sur des entiers dont la taille dépasse celle du mot de base de la cible sur laquelle on travaille. Par exemple, les mots de base d'une architecture classique `x86-64` ont une taille maximale de 64 bits mais on peut aller jusqu'à 128 bits sur des processeurs plus récents.

Les entiers sont décrits par rapport à une taille de référence, appelée la base, et qui sera notée b dans la suite. On désignera par b-mot un mot dans celle-ci. Afin de représenter un entier de taille quelconque, il est donc découpé en b-mots. En manipulant les b-mots, il faut faire attention à la représentation qui est utilisée sur la cible. En effet, plusieurs choix sont possibles quant à l'ordre des octets dans un b-mot :

- **Little endian** : L'octet de poids le plus faible est stocké en premier.
- **Big endian** : L'octet de poids le plus fort est stocké en premier.
- **Bi endian** : Les deux représentations Little endian et Big endian sont possibles.
- **Midlle endian** : Représentation complexe qui n'apparaît que sur de rares architectures.

Le tableau 4.1 présente les représentations adoptées par certaines familles de processeurs.

Famille	Ordre des octets
PowerPC (PPC)	Big Endian
Intel x86	Little Endian
Intel x86_64	Little Endian
MIPS	Bi (Big/Little) Endian
ARM	(Big/Little) Endian
IA-64	Bi (Big/Little) Endian

Tableau 4.1 – Ordre des octets pour certaines familles de processeurs

Exemple 4.1.1
L'entier $2^{16} + 2^{11} + 89$ est représenté par :

- 01011001 00001000 00000001 00000000 en Little endian,

- 00000000 00000001 00001000 01011001 en Big endian.

De même on peut ordonner les b-mots d'un entier de différentes manières. Dans la suite on représentera un entier u par $(u_0, \ldots, u_{n-2}, u_{n-1})_b$ avec u_{n-1} le mot de poids le plus fort et u_0 le mot de poids le plus faible. On a $u = \sum_{i=0}^{n-1} u_i b^i$. Cette représentation est généralement utilisée dans l'arithmétique multi-précision car lors de l'implantation, on peut facilement ajouter des zéros au niveau des poids forts afin d'atteindre la taille souhaitée.

Les entiers négatifs peuvent être représentés de différentes manières

Signe-magnitude : Un autre entier est utilisé pour coder le signe de l'entier. u est représenté par $(s, (u_0, \ldots, u_{n-2}, u_{n-1})_b)$ où s indique le signe de u. Par exemple on peut choisir $s = 0$ pour les entiers positifs et $s = 1$ pour les entiers négatifs. L'inconvénient de cette méthode est que l'entier 0 peut être codé de deux manières différentes $(0, (0, \ldots, 0)_b)$ et $(1, (0, \ldots, 0)_b)$. Cependant, elle permet d'avoir rapidement l'opposé de u puisqu'il faut juste changer s.

Complément à 2 : Le bit de poids fort de l'entier encode son signe. S'il vaut 0, l'entier est positif sinon il est négatif. Avec cette méthode, 0 a une unique représentation mais le calcul de l'opposé nécessite plus d'opérations.

Dans la suite, la méthode signe-magnitude sera utilisée et la base b sera considérée comme étant une puissance de 2.

Nous allons maintenant présenter les algorithmes des opérations de base que sont l'addition, la soustraction et la multiplication. Nous renvoyons à [Coh+12, chapitre 10] pour plus de détails et pour la présentation d'autres opérations comme la division.

Addition et Soustraction Ces deux opérations sont réalisées mot par mot. Le point le plus délicat est la gestion de la retenue qui est générée lors de celles-ci. De manière générale, elles peuvent êtres décrites par les algorithmes I.12 et I.13 lorsque les opérandes sont positives et de même taille.

Algorithme I.12 : Addition multi-précision d'entiers positifs de même taille

Données : `Deux entiers de` n b-`mots` $(u_0, \ldots, u_{n-2}, u_{n-1})_b$ `et` $(v_0, \ldots, v_{n-2}, v_{n-1})_b$.

Résultat : `L'entier de` n b-`mots` $(w_0, \ldots, w_{n-2}, w_{n-1})_b$ `ainsi qu'une retenue` c `vérifiant` $c||w = u + v$.

1 $c = 0$

2 **pour** $i = 0$ `à` $n - 1$ **faire** $c||w_i = u_i + v_i + c$

Afin d'améliorer les performances, plusieurs stratégies d'implantation sont possibles :

- Il est préférable d'implanter la troisième étape en une seule opération. Par exemple, il est possible d'utiliser une instruction assembleur prenant en charge la gestion des retenues. L'instruction `add` met à jour directement le registre `CF` qui indique la retenue, et si l'on souhaite prendre en compte celle-ci il suffit d'utiliser l'instruction `adc`.
- Si la taille des entiers est fixe, par exemple 256 bits pour une base 2^{64}, il vaut mieux dérouler la boucle.
- Quand u et v n'ont pas la même taille, il faut examiner cas par cas. Si v est un entier d'un mot, les opérations peuvent être réalisées plus rapidement car, après l'étape initiale de la boucle, il ne reste qu'à gérer des instructions avec la retenue. De ce fait, on peut s'arrêter dès que la retenue est nulle. Quand la taille de u est plus grande que celle de v, ce dernier est complété par des 0 au niveau des bits de poids fort afin d'atteindre la taille désirée. Si cette complétion est implicite, il faut comme précédemment implanter des instructions qui ont comme particularité d'avoir une opérande nulle.

Algorithme I.13 : Soustraction multi-précision d'entiers positifs de même taille

Données : Deux entiers de n b-mots $(u_0,\ldots,u_{n-2},u_{n-1})_b$ et $(v_0,\ldots,v_{n-2},v_{n-1})_b$ tels que $u \geqslant v$.
Résultat : L'entier de n b-mots $(w_0,\ldots,w_{n-2},w_{n-1})_b$ vérifiant $w = u - v$.

1 $c = 0$
2 **pour** $i = 0$ à $n-1$ **faire** $c||w_i = u_i - v_i + c$

Remarque 4.1.2

1. Concernant la soustraction, il n'est pas obligatoire de tester la comparaison $u \geqslant v$ avant de réaliser l'opération. En effet, une implantation peut être réalisée en utilisant une retenue comme indicateur de signe. Par exemple, si $u < v$, alors la retenue va valoir 1 à la fin et donc on sait que le w doit être négatif. En utilisant la méthode de représentation signe-magnitude, cette correction est sans coût.
2. Les algorithmes présentés prennent comme opérandes des entiers positifs, cependant il est est nécessaire de tenir compte du signe lorsque des entiers négatifs apparaissent.

   Par exemple :

   - si u et v sont de même signe s, la somme est obtenue avec une addition des valeurs absolues et sera de signe s. Pour la différence, une soustraction est réalisée mais le signe dépendra de la retenue résultante.

- si u et v sont de signe différent s_1 et s_2, la différence est obtenue avec une addition des valeurs absolues et sera de signe s_1. Pour la somme, une soustraction est réalisée mais le signe dépendra de la retenue résultante.

Le coût de l'addition et de la soustraction sera notée $C_n(A)$ dans la suite et vaut $O(n)$.

4.1.1.0.1 Multiplication La multiplication est une opération qui est souvent utilisée comme base de complexité dans de nombreuses applications comme nous le verrons dans la suite. Parmi toutes les méthodes existantes, nous allons présenter ici les plus communes, à savoir la version naïve et celle de Karatsuba. D'autres versions comme celles de Toom-Cook ou la FFT sont également utilisées pour des entiers avec un nombre important de mots ([Knu97] et tableau 4.2).

La version naïve est l'algorithme le plus simple à implanter car elle suit l'écriture naturelle de la multiplication (voir algorithme I.14). Pour u et v deux entiers de respectivement m et n b-mots, le produit w est de taille $m+n$ et s'écrit comme

$$\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} u_i v_j b^{i+j}.$$

L'instruction majeure est une multiplication suivit d'une addition sur les b-mots.

Algorithme I.14 : Multiplication multi-précision naïve d'entiers positifs

Données : Un entier u de m b-mots $(u_0,\ldots,u_{m-2},u_{m-1})_b$ et un entier v de n b-mots $(v_0,\ldots,v_{n-2},v_{n-1})_b$.
Résultat : L'entier de $m+n$ b-mots $(w_0,\ldots,w_{m+n-2},w_{m+n-1})_b$ vérifiant $w = uv$.

```
1  pour i = 0 à n − 1 faire w_i = 0
2  pour i = 0 à n − 1 faire
3      c = 0
4      si v_i = 0 alors w_{m+i} = 0
5      sinon
6          pour j = 0 to m − 1 faire
7              t = v_i u_j + w_{i+j} + c
8              w_{i+j} = t mod b
9              c = ⌊t/b⌋
10         w_{m+i} = c
```

Afin que le calcul du signe du produit soit plus aisé, nous avons décidé de changer de convention pour la représentation signe-magnitude. Désormais s vaut 1 pour les entiers positifs et -1 pour les entiers négatifs. Ainsi, le signe du produit est le produit des signes.

Pour améliorer les performances, il est possible d'implanter la septième étape en utilisant une unique opération multiplication-addition.

Pour cette méthode, il y a $n \times m$ multiplications élémentaires. Par conséquent le coût $C_n(M)$ de la multiplication est en $O(n^2)$. Pour l'élévation au carré, il est possible d'avoir un gain d'environ 20% en remarquant que

$$\left(\sum_{i=0}^{n-1} u_i b^i\right)^2 = \sum_{i=0}^{n-1} u_i^2 b^{2i} + 2\sum_{i<j} u_i u_j b^{i+j}.$$

La méthode de Karatsuba (algorithme I.15 et [Kar95]) permet elle d'atteindre un coût en $O(n^{\ln 3})$. C'est un algorithme récursif qui se base sur l'observation suivante. Supposons que u et v soient des entiers de deux b-mots, donc $u = u_0 + bu_1$ et $v = v_0 + bv_1$. Le produit uv vérifie

$$uv = u_1v_1b^2 + ((u_0 + u_1)(v_0 + v_1) - u_1v_1 - u_0v_0)b + u_0v_0.$$

Avec cette écriture, au lieu d'utiliser 4 multiplications, seulement 3 sont nécessaires au prix de 3 additions supplémentaires. Pour des entiers de taille > 2, une approche récursive permet de se ramener au cas précédent comme le montre l'algorithme. Cependant, on ne se ramène exactement au cas précédent, on s'arrête des que la multiplication naïve devient plus rapide que cette méthode. En effet, si le coût d'une multiplication de base est inférieure à trois fois le coût d'une addition de base, la version naïve est plus efficace. En pratique il faut déterminer une borne d_0 qui indique quand il faut s'arrêter. Le point clé de cette méthode est de déterminer les valeurs optimales de d_0. Dans [GMP], il est possible de trouver les estimations suivantes pour les processeurs Intel Sandy/Ivy bridge, Haswell et Skylake (voir tableau 4.2).

Algorithme I.15 : Multiplication multi-précision Karatsuba d'entiers positifs

Données : Un entier de m b-mots $(u_{m-1}, u_{m-2}, \ldots, u_0)_b$, un entier de n b-mots $(v_{n-1}, v_{n-2}, \ldots, v_0)_b$, la taille $d = \max\{m, n\}$ et une borne d_0.

Résultat : L'entier de $m+n$ b-mots $(w_{m+n-1}, w_{m+n-2}, \ldots, w_0)_b$ vérifiant $w = uv$.

1 **si** $d \leqslant d_0$ **alors retourner** uv en utilisant l'algorithme I.14
2 $n_1 = \lfloor d/2 \rfloor$ et $n_2 = \lceil d/2 \rceil$
3 $U_0 = (u_{q-1}, \ldots, u_0)_b$ et $V_0 = (v_{q-1}, \ldots, v_0)_b$
4 $U_1 = (u_{p+q-1}, \ldots, u_q)_b$ et $V_0 = (v_{p+q-1}, \ldots, v_q)_b$
5 $U_s = U_0 + U_1$ et $Vs = V_0 + V_1$
6 Calculer récursivement U_0V_0, U_1V_1 et U_sV_s
7 $w = U_1V_1b^{2q} + (U_sV_s - U_1V_1 - U_0V_0)\, b^q + U_0V_0$

Tableau 4.2 – Borne des algorithmes de multiplication multi-précision pour `GMP` sur certains processeurs Intel.

Algorithme	Sandy/Ivy bridge	Haswell	Skylake
Karatsuba	20	22	26
Toom-3	65	74	73
Toom-4	166	195	208
Toom-6.5	256	298	366
Toom-8.5	333	406	430
FFT	4736	4224	4736

On remarque qu'avec les progrès des nouveaux processeurs, la borne est de plus en plus en faveur de la multiplication naïve car le coût d'une multiplication de base est de moins en moins élevé par rapport à l'addition. Dans le contexte des courbes elliptiques, il paraît logique de penser que la borne d'utilisation de Karatsuba ne sera pas atteinte.

En conclusion, après analyse des différentes implantations possibles pour l'arithmétique multi-précision, nous avons décidé d'adopter les stratégies suivantes :

- La représentation signe-magnitude sera utilisée pour coder les entiers. Le signe s vaudra 1 pour les entiers positifs et -1 sinon. 0 sera représenté par $(1, (0, \ldots, 0))_b$.
- La base b utilisée sera dépendante de la cible et sera discutée dans les chapitres concernant les implantations logicielles et matérielles. Cependant ce sera toujours une puissance de 2.
- Seule la multiplication naïve sera implantée au vu des tailles utilisées dans notre contexte.

Dans la suite, les coûts suivants seront utilisés :

$$C_n(A) = O(n), C_n(M) = O(n^2).$$

4.1.2 Représentation des éléments de $\mathbb{Z}/p\mathbb{Z}$

La méthode classique pour représenter les éléments de $\mathbb{Z}/p\mathbb{Z}$ est d'utiliser les entiers multi-précision ainsi que l'arithmétique associée. Cette dernière est modifiée afin que les résultats des calculs restent dans l'ensemble de départ. Ainsi, la réduction modulaire devient l'opération majeure dans les algorithmes modulaires.

Si p est un nombre premier ayant une forme particulière (par exemple si p est un nombre premier de Mersenne $p = 2^k - 1$) alors le coût de la réduction modulaire peut être faible. Dans le cas des nombres premiers de Mersenne, cela revient à une addition

modulaire. D'autres formes, comme celles des nombres premiers du NIST [FIPS-186-4] permettent également d'obtenir une réduction modulaire efficace.

Quand p n'a pas de forme particulière, il est possible d'utiliser une représentation spécifique des entiers afin de réduire les coûts des algorithmes. Le tableau 4.3 compare les complexités de trois méthodes : la réduction classique (utilisation de la division euclidienne), celle de Barrett et celle de Montgomery [BGV94].

Algorithme	Classique	Barrett	Montgomery
Multiplications	$n(n+2.5)$	$n(n+4)$	$n(n+1)$
Divisions	n	-	-
Pré-calculs	Normalisation de p	$\lfloor b^{2n}/p \rfloor$	Réduction
Restrictions	-	$u < b^{2n}$	$u < pb^n$

Tableau 4.3 – Comparaison des algorithmes de réduction modulaire pour des entiers de n bits.

Les tests présentés dans l'article laissent à penser que la méthode de Montgomery est plus rapide que celle de Barrett, tout comme la discussion dans [CP01]. Nous ne traiterons ici que du cas de la représentation de Montgomery. Pour les détails concernant la méthode de Barrett, voir [Coh+12, section 10.4]. Une comparaison d'algorithmes de multiplication modulaire est également donnée dans [Dhe98].

Dans [Mon85], Montgomery introduit une nouvelle manière de représenter les éléments de $\mathbb{Z}/p\mathbb{Z}$ qui permet de rendre les opérations plus rapides. Elle peut être vue comme une généralisation de la division de Hensel pour calculer les inverses des nombres 2-adique [Hen08].

Définition 4.1.3
Soient $p > 3$ un nombre premier et R un entier supérieur à p et premier avec lui. La représentation de Montgomery de $x \in [0, p-1]$ est $[x] = (xR) \bmod p$. La réduction de Montgomery de $u \in [0, Rp-1]$ est $REDC(u) = (uR^{-1}) \bmod p$.

Remarque 4.1.4
La définition d'origine ne nécessite pas que p soit premier.

La réduction de Montgomery permet également de passer d'une représentation à une autre. En effet, $[x]$ peut être obtenu avec le calcul de $REDC((xR^2) \bmod p)$.

$$REDC((xR^2) \bmod p) = xR^2R^{-1} \bmod p = xR \bmod p.$$

D'autre part, pour tout $x \in [0, p-1]$:

$$REDC([x]) = xRR^{-1} \bmod p = x \bmod p = x.$$

Il est possible de pré-calculer la valeur $R^2 \bmod p$ pour accélérer le calcul de transfert de représentation.

Un algorithme efficace peut être obtenu quand R est une puissance de la base b. Dans la suite on considérera $R = b^n$.

Algorithme I.16 : Algorithme de réduction de Montgomery pour des entiers multi-précision

Données : `Un nombre premier` $p > 3, R = b^n$, $p' = (-p^{-1}) \bmod R$ `et un entier de` $2n$ `mots` $u < Rp$

Résultat : `L'entier de` n `mots` t `tel que` $t = REDC(u) = (uR^{-1}) \bmod p$.

1 $t = u$
2 **pour** $i = 0$ à $n-1$ **faire**
3 $\quad k_i = (t_i p') \mod b$
4 $\quad t = t + k_i p b^i$
5 $t = t/R$
6 **si** $t \geqslant p$ **alors** $t = t - p$

Analysons pourquoi cet algorithme calcule bien $REDC(u)$. Tout d'abord, à la fin de la boucle, k et t vérifient les égalités suivantes :

$$k \equiv up' (\bmod R) \text{ et } t = u + kp.$$

t étant un multiple de R, la division de la cinquième étape est exacte. Comme p et R sont premiers entre eux, la nouvelle valeur de t après la division vérifie $t \equiv uR^{-1} (\bmod p)$. L'algorithme suppose que $u \leqslant Rp$, donc il est possible de prouver que $0 \leqslant t < 2p$. Par conséquent t ou $t - p$ vaut $REDC(u)$.

Le coût de cet algorithme est en $O(n^2 + n)$ si on ne prend pas en compte les pré-calculs nécessaires.

Nous allons maintenant présenter les principaux algorithmes modulaires en utilisant la représentation de Montgomery. L'inversion modulaire n'étant pas une opération critique dans les algorithmes considérés pour l'arithmétique des courbes, nous ne la détaillerons pas et renvoyons à [Coh+12, section 11.1.3].

4.1.3 Addition modulaire

L'addition et la soustraction modulaire ne sont pas dépendantes de la représentation utilisée pour les éléments de $\mathbb{Z}/p\mathbb{Z}$ car elles s'appuient sur les algorithmes correspondants pour les entiers multi-précision (voir algorithmes I.12 et I.13).

En effet, si u et v sont des entiers dans l'intervalle $[0, p-1]$, alors la somme sera strictement inférieure à $2p$. Donc, l'addition modulaire est soit $u+v$, soit $u+v-p$. De même, la différence est soit $u-v$ si $u \geqslant v$, sinon en remplaçant u par $u+p$ on se ramène au cas précédent. Il est possible d'unifier les deux algorithmes en utilisant une variable indiquant l'opération en cours. Ainsi, l'algorithme I.17 réalise les opérations désirées.

Algorithme I.17 : Algorithme d'addition modulaire

Données : `Un nombre premeier` $p > 3$ `et` $0 \leq u, v < p$. `Une variable` f `indiquant si on réalise une addition ou une soustraction (`$f = 0$ `pour une addition et 1 sinon).`

Résultat : $C = u + (-1)^f v \pmod p$ `tel que` $0 \leq C < p$.

1 $(C_0, Cf_0) = u + (-1)^f v$
2 $(C_1, Cf_1) = C_0 + (-1)^{(1-f)} p$
3 **retourner** $C_{|1-Cf_f|}$

Vérifions la correction de l'algorithme :

- Si $f = 0$, $C_0 = u + v$ et $C_1 = C_0 - p$.
  - Si le calcul de C_0 produit une retenue alors Cf_0 vaut 1 et il faut donc retourner C_1,
  - sinon, il faut regarder si le calcul de C_1 produit une retenue. Si c'est la cas, alors, $C_0 < p$ donc il faut retourner C_0 sinon C_1.
- Si $f = 1$, $C_0 = u - v$ et $C_1 = C_0 + p$.
  - Si le calcul de C_1 produit une retenue alors Cf_1 vaut 1 et il faut donc retourner C_0,
  - sinon, il faut retourner C_0.

Le coût de l'addition modulaire, noté $C_n(A, p)$ est en $O(n)$.

4.1.4 Multiplication modulaire

En utilisant la représentation de Montgomery, la multiplication modulaire se réduit à l'utilisation de l'algorithme de réduction. Cette propriété de $REDC$ vient de l'égalité suivante :

$$\Big((xR \bmod p)(yR \bmod p)R^{-1} \bmod p\Big) = (xyR) \bmod p.$$

Il s'ensuit que $REDC([x][y]) = [xy]$. Une adaptation de $REDC$ afin de prendre en charge deux opérandes permet donc de réaliser une multiplication modulaire (voir algorithme I.18).

Algorithme I.18 : Algorithme de multiplication modulaire de Montgomery

Données : Un nombre premier $p > 3, R = b^n$, $p' = (-p^{-1}) \bmod R$ et deux entiers u, v de n mots tels que $0 \leqslant u, v < p$

Résultat : L'entier de n mots t tel que $t = REDC(uv) = (uvR^{-1}) \bmod p$.

```
1 t = 0
2 pour i = 0 à n − 1 faire
3     m_i = ((t_0 + u_i v_0) p') mod b
4     t = (t + u_i v + m_i p)/b
5 si t ⩾ p alors t = t − p
```

4.2 Arithmétique sur certaines extensions de $\mathbb{F}_p$

Dans cette section, nous étudierons la représentation des éléments de l'extension de corps $\mathbb{F}_{p^k}$ pour $k > 1$. En fonction des caractéristiques de k et de p, nous verrons que différentes méthodes peuvent implantées en utilisant des paramètres optimaux. Ensuite, les algorithmes de multiplication et d'élévation au carré sur les extensions cubiques et quadratiques seront explicités ainsi que les possibilités de parallélisation. Enfin, nous discuterons des différentes implantations que l'on peut obtenir pour $\mathbb{F}_{p^6}$ en utilisant les méthodes précédemment rappelées.

4.2.1 Tour d'extension de corps

Une représentation naturelle des éléments de l'extension $\mathbb{F}_{p^k}$ consiste à utiliser des polynômes de degré $k-1$ à coefficients dans $\mathbb{F}_p$. On peut donc écrire $\mathbb{F}_{p^k}$ de la manière suivante $\mathbb{F}_{p^k} = \mathbb{F}_p[X]/(P)$ où P est une polynôme irréductible dans $\mathbb{F}_p[X]$ de degré k. La multiplication des éléments de $\mathbb{F}_{p^k}$ se réduit ainsi à une multiplication polynomiale suivie d'une réduction modulo le polynôme P. Le choix de ce dernier influence donc le coût des opérations et il est préférable de le prendre avec un nombre minimal de termes et de « petits » coefficients.

Dans le contexte des couplages sur courbes elliptiques, nous sommes amenés à travailler avec des degrés d'extension qui peuvent être très grands devant 2. Les courbes présentées dans [FST10] vont jusqu'à $k = 50$. Pour de telles extensions, la représentation naturelle implique une arithmétique trop complexe et il faut donc envisager un autre type de construction. Dans le cas où k est friable, l'idée d'utiliser une tour d'extension devient pertinente. Elle consiste à construire chaque couche de l'extension à partir de la précédente. Cette construction a été suggérée par Baktir et Sunar dans [BS04]. Ils ont remarqué qu'en utilisant des tours d'extension, l'implantation des OEFs était facilitée et

plus précisément le coût de l'inversion modulaire était amélioré. De plus, le standard IEEE « P1363.3 : Standard for Identity-Based Cryptographic techniques using Pairings » préconise également cette construction pour les extensions de nombres premiers impairs [IEEE-P1363.3].

Afin d'améliorer performances, il est également possible d'utiliser des réductions partielles. Le principe a été introduit par Lim et Hwang dans [LH00], mais le terme « Lazy reduction » a été inventé par Avanzi dans [Ava04]. Cela consiste à supprimer certaines réductions modulaires lors d'un calcul au prix d'un coût mémoire plus important et d'additions supplémentaires. Son utilisation dans le contexte des calculs de couplages a été discutée dans [Sco07] afin de réduire le nombre de réductions pour les multiplications sur les extensions quadratiques. Dans [AKLGL11], les auteurs ont également travaillé sur une adaptation à la réduction de Montgomery.

Les problématiques soulevées par l'utilisation des tours d'extension sont les suivantes :

1. Quelle méthode utilisée en fonction du p considéré ?
2. Quelle est la meilleure tour à utiliser pour une valeur de k donnée ?

Concernant la première problématique, différentes solutions se trouvent dans la littérature. Dans [KM05], Koblitz et Menezes préconisent l'utilisation des corps « adaptés aux couplages » (pairing-friendly fields).

Définition 4.2.1
L'extension $\mathbb{F}_{p^k}$ est un « corps adapté aux couplages » si $p = 1 \bmod 12$ et si k est de la forme $2^i 3^j$. Pour $j = 0$, la condition sur p est réduite à $p = 1 \bmod 4$.

La construction repose sur le théorème suivant :

Théorème 4.2.2 ([LN97, Théorème 3.75])
Soient $q = p^n$, $k \geqslant 2$ un entier et $\omega \in \mathbb{F}_q^\times$. Alors le binôme $X^k - \omega$ est irréductible dans $\mathbb{F}_q[X]$ si et seulement si les deux conditions suivantes sont satisfaites :

1. *Chaque facteur premier de k divise l'ordre e de ω dans $\mathbb{F}_q^\times$, mais ne divise pas $\frac{q-1}{e}$;*
2. *$q = 1 \bmod 4$ si $k = 0 \bmod 4$.*

Dans le cas des corps « adaptés aux couplages », k est de la forme $2^i 3^j$. La première condition peut donc se résumer au fait que ω ne soit pas résidu quadratique ou cubique. De plus pour $j = 0$, il n'est pas nécessaire de vérifier que ω ne soit pas un résidu cubique. L'extension se construit par ajout des racines cubiques et quadratiques successives de ω. Afin de justifier la possibilité d'une telle construction on peut utiliser la proposition suivante qui caractérise de telles racines.

Proposition 4.2.3
Soit ω un élément de $\mathbb{F}_{p^n}$ qui n'est pas un résidu quadratique ou cubique. Soient ω_ℓ une de ses

racines ℓ-ièmes dans $\mathbb{F}_{p^{n\ell}}$ pour $\ell = 2, 3$. Alors ω_ℓ n'est pas un résidu quadratique ou cubique dans $\mathbb{F}_{p^{n\ell}}$.

Démonstration. On utilise l'ordre des éléments pour démontrer ce résultat. Soient e l'ordre de ω et e_ℓ l'ordre de ω_ℓ. On a la relation suivante qui est vérifiée $e_\ell = \ell \times e$. Étant donné que ω n'est pas un résidu quadratique ou cubique, son ordre vérifie

$$e = 0 \bmod 2, 3 \text{ et } \frac{p^n - 1}{e} \neq 0 \bmod 2, 3.$$

Il est immédiat que e_ℓ vérifie $e_\ell = 0 \bmod 2, 3$. Supposons que $\frac{p^{n\ell}-1}{e_l} = 0 \bmod 2, 3$. Alors,

$$\frac{p^{2n} - 1}{2e} = \frac{(p^n - 1)(p^n + 1)}{2e} = 0 \bmod 2, 3,$$

et

$$\frac{p^{3n} - 1}{3e} = \frac{(p^n - 1)(p^{2n} + p^n 1)}{3e} = 0 \bmod 2, 3.$$

On en déduit que

$$\frac{p^n + 1}{2} = 0 \bmod 2 \text{ et } \frac{p^{2n} + p^n + 1}{3} = 0 \bmod 3.$$

Or $p = 1 \bmod 12$, donc cela n'est pas possible. Par conséquent, ω_ℓ n'est pas un résidu quadratique ou cubique dans $\mathbb{F}_{p^{n\ell}}$. □

Exemple 4.2.4
Soient $k = 6$ et $p = 2^{256} - 2^{168} + 1$. Nous sommes bien dans le cas d'un corps « adapté aux couplages ». On peut vérifier que 11 n'est pas un résidu quadratique ou cubique, donc posons $\omega = 11$. On construit tout d'abord la première couche d'extension $\mathbb{F}_{p^2}$ (on aurait pu commencer par $\mathbb{F}_{p^3}$; on discutera de l'ordre des couches dans la suite) en utilisant le binôme $X^2 - 11$. On peut donc écrire $\mathbb{F}_{p^2} = \mathbb{F}_p[X]/(X^2 - 11)$. Maintenant que l'on est dans $\mathbb{F}_{p^2}$, on va utiliser une extension cubique pour aller jusqu'à $\mathbb{F}_{p^6}$ avec le polynôme $Y^3 - \sqrt{11}$ qui est irréductible dans $\mathbb{F}_{p^2}[Y]$.

Cette définition impose une condition sur p qui n'est pas respectée dans certaines courbes « adaptée aux couplages ». La difficulté à trouver de telles courbes amène à se demander s'il ne faudrait pas retirer cette condition sur p. Cette observation de Benger et Scott dans [BS10], les a poussés à proposer une nouvelle caractérisation des extensions optimales qui restreint moins les nombres premiers utilisés. Ils ont donc introduit la notion de corps « adaptés aux tours d'extension ».

Définition 4.2.5
Un corps « adapté aux tours d'extension » est un corps de la forme $\mathbb{F}_{q^m}$, où $q = p^n$ est une puissance d'un nombre premier, pour lequel tous les diviseurs premiers de m divisent aussi $q - 1$.

Remarque 4.2.6
Les corps « adaptés aux couplages » de Koblitz et Menezes [KM05] rentrent dans cette définition.

La méthode de construction est semblable à la précédente. Afin de construire $\mathbb{F}_{p^{nm}}$ sur $\mathbb{F}_p$, on utilise un binôme $X^m - \omega$ qui est irréductible sur $\mathbb{F}_{p^n}[X]$ et on ajoute successivement les racines de la dernière racine ajoutée jusqu'à ce que la tour soit terminée. Le théorème 4.2.2 impose que ω ne soit une puissance l-ième dans $\mathbb{F}_{p^n}$ pour aucun des diviseurs premiers ℓ de m. Si l'on examine les autres contraintes, on remarque que cette méthode fonctionne pour tout m, $m \neq 0 \bmod 4$. Cependant, lorsque m est un multiple de 4 et que $p^n \neq 1 \bmod 4$, elle n'est pas utilisable. Benger et Scott ont donc proposé une alternative qui passe par la construction d'une base.

Dans le cas où $p^n = 3 \bmod 4$, ils proposent de prendre comme nouvelle base de la tour l'extension quadratique $\mathbb{F}_{p^{2n}}$. En effet, étant donné que $p^{2n} = 1 \bmod 4$, on peut se ramener dans le cadre du théorème 4.2.2 pour construire la tour sur $\mathbb{F}_{p^{2n}}$ en utilisant le binôme $X^{m/2} - \omega$ avec $\omega \in \mathbb{F}_{p^{2n}}$. Dans le cas où $n = 1$, on peut rajouter la racine carrée de -1 pour construire la base. Benger et Scott ont également remarqué que l'on pouvait utiliser cette idée de base lorsque m ne respecte pas les contraintes fixées (voir [BS10, remarque 3]).

Il est donc possible de construire la tour d'extension avec des contraintes plus faibles sur p. Cependant, il nous reste une problématique à examiner : Quelle est la meilleure tour à utiliser pour une valeur de k donnée ? En particulier, comment choisir de manière optimale la valeur ω et dans quel ordre faut-il faire les extensions ?

Tout d'abord, avant de déterminer la valeur optimale, il faut vérifier si elle respecte les contraintes de la construction. En utilisant [BS10, Théorème 4], il est possible de tester l'irréductibilité du binôme $X^m - \omega$ sur $\mathbb{F}_{p^n}[X]$, où ω est un élément de $\mathbb{F}_{p^n}$ en utilisant la norme de $\mathbb{F}_{p^n}$ sur $\mathbb{F}_p$ de ω. Cette technique est moins onéreuse que le calcul d'ordre qui apparaît dans le théorème 4.2.2.

Si plusieurs valeurs sont possibles pour ω, notre choix peut être influencé par le nombre d'additions que requiert l'utilisation de telles ou telles valeurs. De plus, dans le contexte des couplages, les torsions de courbes sont très importantes. Par conséquent, le choix de ω ne doit pas engendrer un sur-coût dans l'implantation de l'isomorphisme de la torsion (voir [BS10, section 7]).

Concernant l'ordre dans lequel la tour doit être construite, il n'y a pas d'obligation. Certains préconisent de commencer ou de terminer par une extension quadratique quand cela est possible. Par exemple, [BS10, Tableau 1] regroupe les suggestions de Benger et Scott pour la construction des tours.

En conclusion, les corps utilisés dans les couplages sur les courbes elliptiques sont de manière générale des corps « adaptés aux tours d'extension ». Il est possible de construire ces tours en utilisant une succession d'extensions quadratiques et cubiques en utilisant le théorème 4.2.2. Dans le cas où la seconde condition du théorème n'est pas vérifiée, la tour peut être construite en utilisant l'extension quadratique du corps comme base. Le choix du polynôme irréductible $X^k - \omega$ doit prendre en compte le nombre d'addition que génère l'utilisation de ω mais également les torsions utilisées lors du couplage. Enfin, l'ordre de construction de la tour n'est pas immuable, différentes stratégies peuvent être opérées.

Dans la suite, nous verrons comment obtenir une implantation efficace des extensions quadratiques et cubiques en reprenant l'analyse de [DOSD06]. L'addition peut être réalisée composante par composante. Si l'on peut paralléliser l'implantation, on peut obtenir le même coût que pour une addition sur $\mathbb{F}_p$. La multiplication et l'élévation au carré présentent les coûts les plus critiques car ce sont les opérations qui interviennent majoritairement dans les calculs impliquant les extensions. Nous verrons donc en détail les différents algorithmes qu'il est possible d'utiliser. Nous examinerons également l'influence de l'ordre de construction de la courbe dans le cas $k = 6$.

4.2.2 Extension quadratique de $\mathbb{F}_p$

Nous allons construire l'extension quadratique de $\mathbb{F}_p$ en utilisant le binôme $X^2 - \omega$ où $\omega \in \mathbb{F}_p$ n'est pas un résidu quadratique dans $\mathbb{F}_p$. Un élément $a \in \mathbb{F}_{p^2}$ est donc représenté par $a_0 + a_1 X$ avec $a_i \in \mathbb{F}_p$.

Afin de réaliser le produit $c = ab$ dans $\mathbb{F}_{p^2}$, deux algorithmes peuvent être utilisés :

- la multiplication naïve,
- la multiplication de Karatsuba.

Ils sont inspirés de ceux utilisés pour l'arithmétique multi-précision.

Algorithme I.19 : Algorithme de multiplication naïve sur $\mathbb{F}_{p^2}$

Données : $a = [a_0, a_1]$, $b = [b_0, b_1]$ avec $a_i, b_i \in \mathbb{F}_p$, ω un non résidu quadratique de $\mathbb{F}_p$.
Résultat : $c = [c_0, c_1] = ab$ avec $c_i \in \mathbb{F}_p$.

1 $c_0 = a_0 b_0 + \omega a_1 b_1 \pmod p$
2 $c_1 = a_0 b_1 + a_1 b_0 \pmod p$

L'avantage de la multiplication naïve est qu'elle bien adaptée à la parallélisation car

dès le départ, on peut lancer plusieurs opérations en parallèle. La figure 4.1 présente différentes architectures que l'on peut obtenir en fonction du nombre de multiplications et d'additions que l'on peut mettre en parallèle. Les architectures utilisées ont les caractéristiques suivantes :

- Architecture 1 : trois additionneurs et 3 multiplieurs sur $\mathbb{F}_p$,
- Architecture 2 : trois additionneurs et 6 multiplieurs sur $\mathbb{F}_p$,
- Architecture 3 : nombre illimité d'additionneurs et de multiplieurs.

Les mêmes architectures seront utilisées dans la suite pour décrire la parallélisation des autres algorithmes.

L'analyse de l'algorithme I.19 permet de constater que les quatre multiplications suivantes peuvent être lancées en même temps : a_0b_0, a_1b_1, a_0b_1 et a_1b_0. Ensuite, il faudrait une addition pour terminer le calcul de c_1 et une multiplication par ω suivie d'une addition pour celui de c_0. En général, ω prend des valeurs assez petites et on peut considérer que le coût de la multiplication associée est négligeable et que par conséquent ces deux dernières opérations peuvent se terminer en même temps dans certains cas. Par exemple, dans les cas où ω vaut -1, 2 ou -2, cela est possible. Cette supposition sur ω sera utilisée dans la suite également.

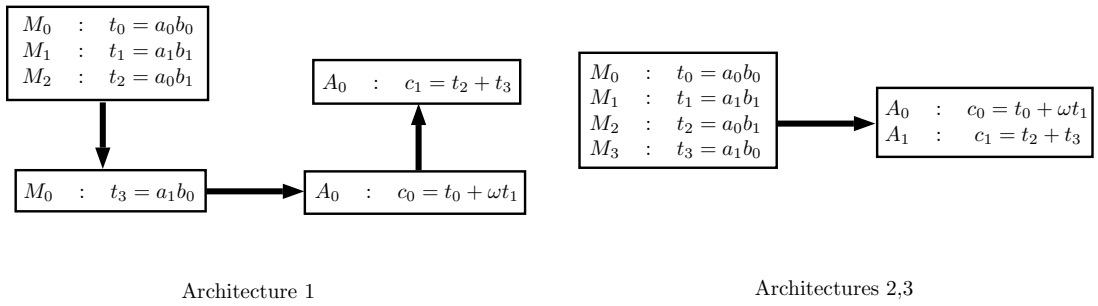


Figure. 4.1 – Parallélisation de la multiplication naïve sur $\mathbb{F}_{p^2}$

Pour la multiplication de Karatsuba (algorithme I.20), on ne peut pas lancer autant de multiplications en parallèle dès le début car les opérandes de la multiplications qui intervient dans le calcul de c_1 nécessitent des additions préalables.

La figure 4.2 montre le schéma de parallélisation que l'on peut obtenir. On constate que les trois architectures présentent les la même possibilité d'implantation car le nombre de multiplications parallélisables est plus faible.

Algorithme I.20 : Algorithme de multiplication Karatsuba sur $\mathbb{F}_{p^2}$

Données : $a = [a_0, a_1]$, $b = [b_0, b_1]$ avec $a_i, b_i \in \mathbb{F}_p$, ω un non résidu quadratique de $\mathbb{F}_p$.

Résultat : $c = [c_0, c_1] = ab$ avec $c_i \in \mathbb{F}_p$.

1 $v_0 = a_0 b_0 \pmod p$

2 $v_1 = a_1 b_1 \pmod p$

3 $c_0 = v_0 + \omega v_1 \pmod p$

4 $c_1 = (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 \pmod p$

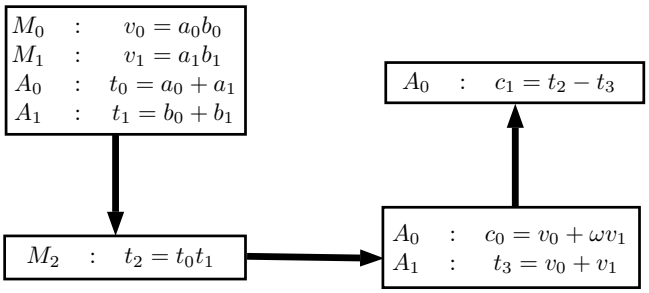


Figure. 4.2 – Parallélisation de la multiplication Karatsuba sur $\mathbb{F}_{p^2}$

S'il on résume les différents coûts d'implantation que l'on peut obtenir pour la multiplication sur $\mathbb{F}_{p^2}$, on obtient le tableau 4.4. Nous n'avons mis que les architectures 1 et 2 car la dernière présente les mêmes performances que la deuxième. Nous avons ajouté le cas des algorithmes non parallélisés afin d'avoir une formule lorsqu'on devra vérifier les coûts dans les tours d'extension. En effet, s'il est relativement aisé d'avoir plusieurs multiplieurs sur $\mathbb{F}_p$ à notre disposition, il se peut que l'on ait qu'un multiplieur pour $\mathbb{F}_{p^2}$. Par conséquent s'il on construit une extension par dessus, on devra utiliser une version non parallélisée. D'autre part, il faut noter qu'en fonction des opérateurs disponibles, l'algorithme le plus efficace peut changer. Nous avons mis en vert les meilleurs coûts obtenus pour chaque architecture.

Algorithme	Non parallélisée	Architecture 1	Architecture 2
Naïve	4M + 2A	2M + 1A	**1M + 1A**
Karatsuba	**3M + 5A**	**1M + 2A**	1M + 2A

Tableau 4.4 – Tableau de comparaison des coûts d'une multiplication sur $\mathbb{F}_{p^2}$

Afin de réaliser le carré $c = a^2$ dans $\mathbb{F}_{p^2}$, trois algorithmes peuvent être utilisés :

- la version naïve,
- la version Karatsuba,
- la version complexe.

Les versions naïve et Karatsuba (algorithmes I.21 at I.22) sont une adaptation des algorithmes précédents. Quasiment toutes les multiplications sont remplacées par des carrés. De plus, dans la version naïve, on gagne une multiplication dans le calcul de c_1.

Algorithme I.21 : Algorithme d'élévation au carré naïve sur $\mathbb{F}_{p^2}$

Données : $a = [a_0, a_1]$ avec $a_i \in \mathbb{F}_p$, ω un non résidu quadratique de $\mathbb{F}_p$.
Résultat : $c = [c_0, c_1] = A^2$ avec $c_i \in \mathbb{F}_p$.

1 $c_0 = a_0^2 + \omega a_1^2 \pmod p$
2 $c_1 = 2a_0a_1 \pmod p$

En analysant les possibilités de parallélisation, on constate que la version naïve est plus adaptée comme dans le cas de la multiplication. De plus, elle ne nécessite que trois multiplieurs pour être à son maximum de performance. Globalement les versions naïve et Karatsuba présentent un schéma identique pour toutes les architectures étudiées (voir figure 4.3).

Algorithme I.22 : Algorithme d'élévation au carré Karatsuba sur $\mathbb{F}_{p^2}$

Données : $a = [a_0, a_1]$ avec $a_i \in \mathbb{F}_p$, ω un non résidu quadratique de $\mathbb{F}_p$.
Résultat : $c = [c_0, c_1] = A^2$ avec $c_i \in \mathbb{F}_p$.

1 $v_0 = a_0^2 \pmod p$
2 $v_1 = a_1^2 \pmod p$
3 $c_0 = v_0 + \omega v_1 \pmod p$
4 $c_1 = (a_0 + a_1)^2 - v_0 - v_1 \pmod p$

La version complexe (algorithme I.23) vient d'un algorithme utilisé dans les opérations arithmétiques sur des nombres complexes. En effet, le carré $c_0 + c_1 i = (a_0 + a_1 i)^2$ est calculé comme suit :

$$c_0 = (a_0 + a_1)(a_0 - a_1),$$
$$c_1 = 2a_0a_1.$$

Algorithme I.23 : Algorithme d'élévation au carré complexe sur $\mathbb{F}_{p^2}$

Données : $a = [a_0, a_1]$ avec $a_i \in \mathbb{F}_p$, ω un non résidu quadratique de $\mathbb{F}_p$.
Résultat : $c = [c_0, c_1] = a^2$ avec $c_i \in \mathbb{F}_p$.

1 $v_0 = a_0a_1 \pmod p$
2 $c_0 = (a_0 + a_1)(a_0 + \omega a_1) - v_0 - \omega v_0 \pmod p$
3 $c_1 = 2v_0 \pmod p$

L'analyse de cet algorithme montre qu'il requiert une multiplication de moins que les autres versions. Cependant, il faut faire une addition de plus que dans la version

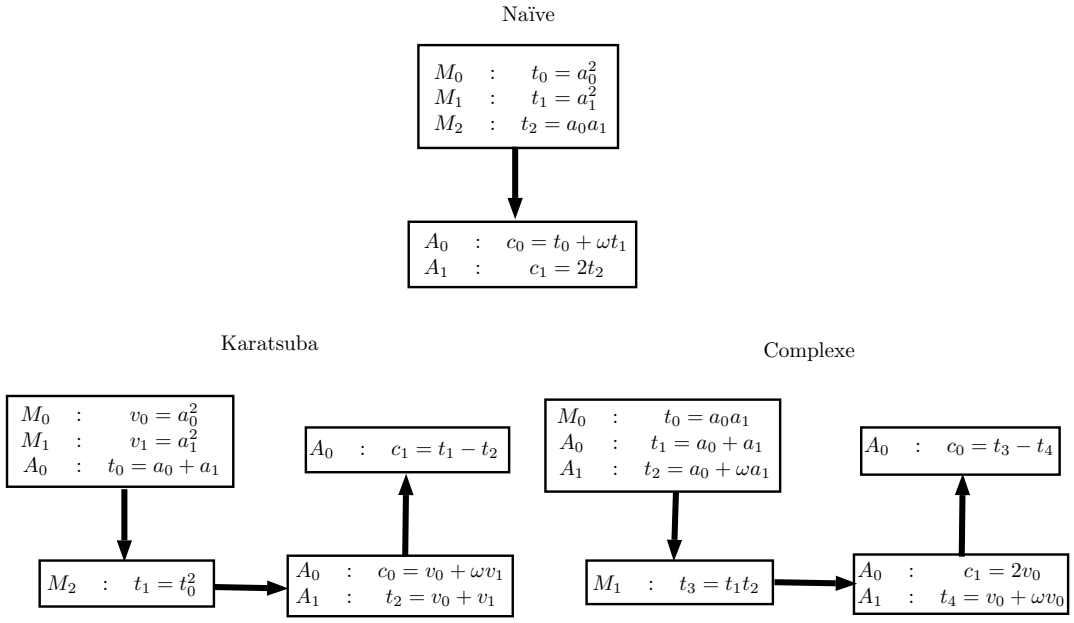


Figure. 4.3 – Parallélisation de l'élévation au carré sur $\mathbb{F}_{p^2}$

Karatsuba et deux plus que dans la version naïve. En implantation non parallélisée, il n'est efficace que si le coût d'une multiplication est supérieure à celui de deux additions. En version parallélisée, on a une performance similaire à celui de Karatsuba.

Le tableau 4.5 résume les coûts de l'élévation au carré pour chacune des architectures. Il faut retenir que l'algorithme naïf est le plus efficace en parallèle et ne nécessite que 3 multiplieurs. En version non parallèle, une discussion peut intervenir car le meilleur algorithme dépend fortement du rapport de coût entre l'addition et la multiplication. C'est donc pour cela que les versions naïve et complexe apparaissent en vert dans le tableau.

Algorithme	Non parallélisée	Architecture 1	Architecture 2
Naïve	**3M + 2A**	**1M + 1A**	**1M + 1A**
Karatsuba	3M + 3A	1M + 2A	1M + 2A
Complexe	**2M + 4A**	1M + 2A	1M + 2A

Tableau 4.5 – Tableau de comparaison des coûts d'une élévation au carré sur $\mathbb{F}_{p^2}$

4.2.3 Extension cubique de $\mathbb{F}_p$

Nous allons construire l'extension cubique de $\mathbb{F}_p$ en utilisant le binôme $X^3 - \omega$ où $\omega \in \mathbb{F}_p$ n'est pas un résidu cubique dans $\mathbb{F}_p$. Un élément $a \in \mathbb{F}_{p^3}$ est donc représenté par

$a_0 + a_1X + a_2X^2$ avec $a_i \in \mathbb{F}_p$.

Nous ne présenterons ici que deux algorithmes (les plus adaptés aux tailles considérées) permettant de réaliser le produit $c = ab$ dans $\mathbb{F}_{p^3}$: la multiplication naïve et la multiplication de Karatsuba.

Algorithme I.24 : Algorithme de multiplication naïve sur $\mathbb{F}_{p^3}$

Données : $a = [a_0, a_1, a_2]$, $b = [b_0, b_1, b_2]$ avec $a_i, b_i \in \mathbb{F}_p$, ω un non résidu cubique de $\mathbb{F}_p$.

Résultat : $c = [c_0, c_1, c_2] = ab$ avec $c_i \in \mathbb{F}_p$.

1 $c_0 = a_0b_0 + \omega(a_1b_2 + a_2b_1) \pmod p$

2 $c_1 = a_0b_1 + a_1b_0 + \omega a_2b_2 \pmod p$

3 $c_2 = a_0b_2 + a_1b_1 + a_2b_0 \pmod p$

La multiplication naïve présente 9 multiplications indépendantes. On peut donc facilement les paralléliser en fonction de l'architecture choisie. La figure 4.4 montre les différentes implantations possibles en fonction du nombre de multiplieurs que l'on a à notre disposition.

L'algorithme de Karatsuba possède moins de multiplications que précédemment (6 au lieu de 9). Cependant, comme pour l'extension cubique, cela se fait au détriment du nombre d'additions et de plus, certaines multiplications nécessitent des additions préalables.

Algorithme I.25 : Algorithme de multiplication Karatsuba sur $\mathbb{F}_{p^3}$

Données : $a = [a_0, a_1, a_2]$, $b = [b_0, b_1, b_2]$ avec $a_i, b_i \in \mathbb{F}_p$, ω un non résidu cubique de $\mathbb{F}_p$.

Résultat : $c = [c_0, c_1, c_2] = ab$ avec $c_i \in \mathbb{F}_p$.

1 $v_0 = a_0b_0 \pmod p$; $v_1 = a_1b_1 \pmod p$; $v_2 = a_2b_2 \pmod p$

2 $c_0 = v_0 + \omega((a_1 + a_2)(b_1 + b_2) - v_1 - v_2) \pmod p$

3 $c_1 = (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 + \omega v_2 \pmod p$

4 $c_2 = (a_0 + a_2)(b_0 + b_2) - v_0 + v_1 - v_2 \pmod p$

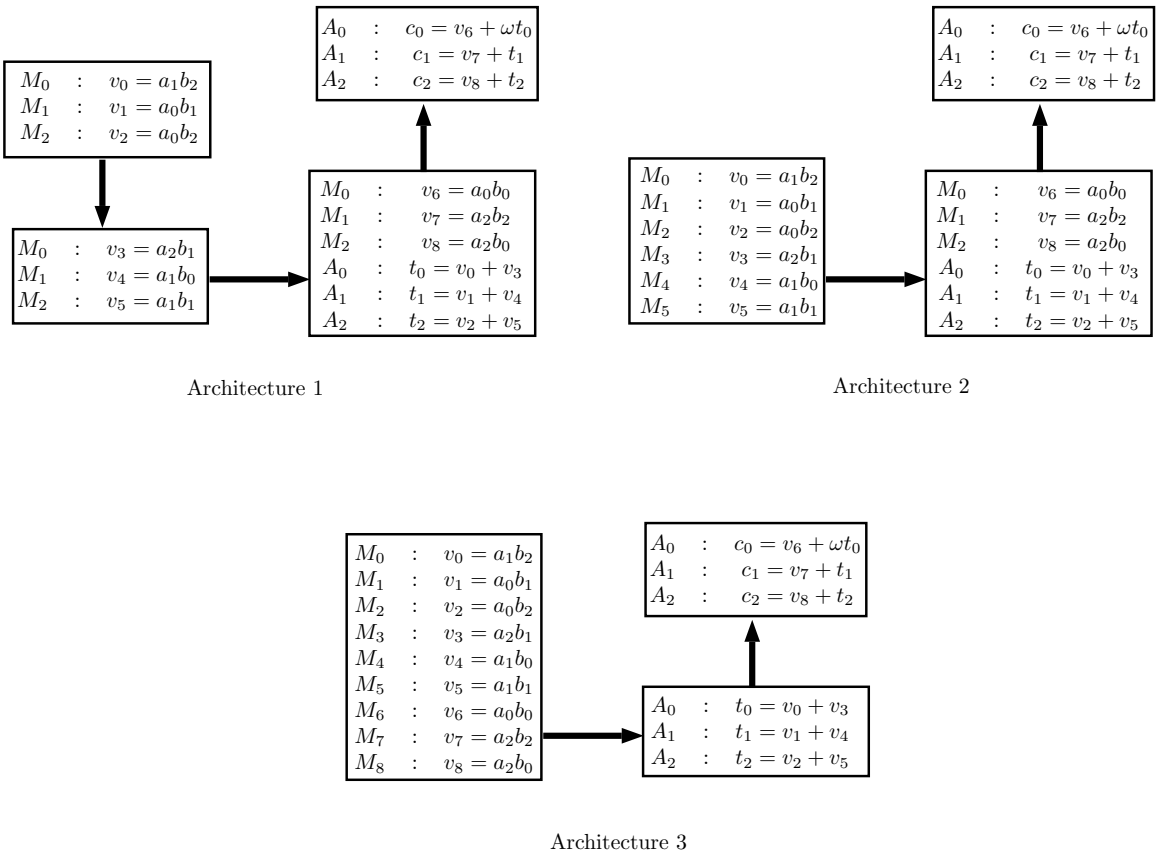
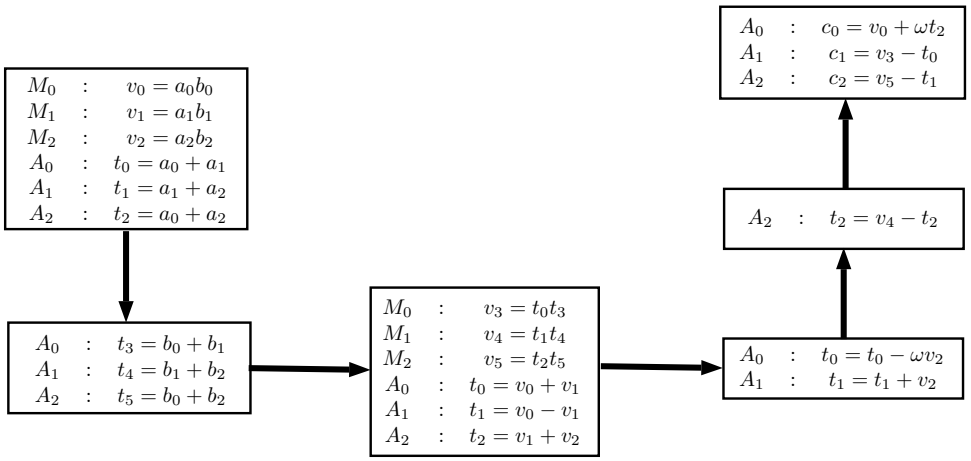


Figure. 4.4 – Parallélisation de la multiplication naïve sur $\mathbb{F}_{p^3}$

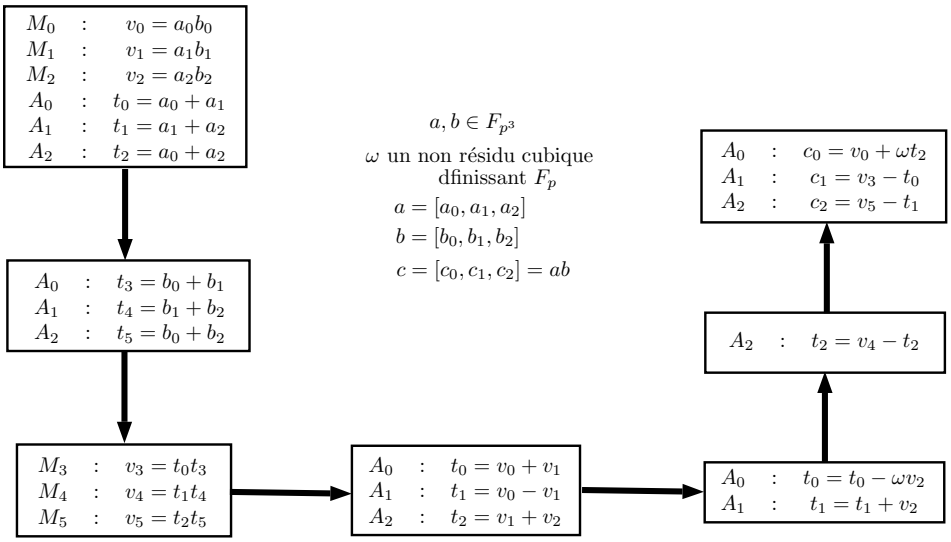
La figure 4.5 regroupe les résultats de parallélisation pour les trois architectures. La différence majeure que l'on observe entre les architectures 2 et 3 réside dans le nombre d'additionneurs disponibles. Cependant, l'analyse des coûts présentée dans le tableau 4.6, montre que le gain entre ces deux architecture est faible. Par conséquent, un compromis est envisageable pour une implantation matérielle si l'on considère également le coût en surface.

Algorithme	Non parallélisée	Architecture 1	Architecture 2	Architecture 3
Naïve	9M + 6A	3M + 1A	2M + 1A	**1M + 2A**
Karatsuba	**6M + 13A**	2M + 2A	1M + 4A	**1M + 3A**

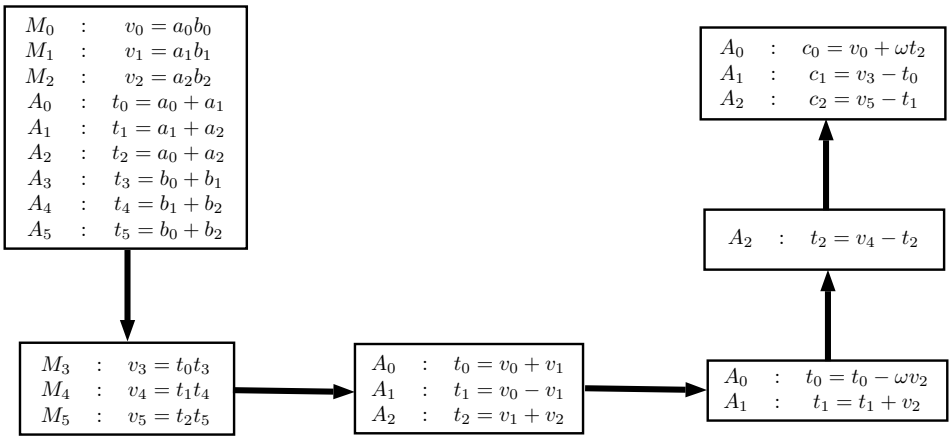
Tableau 4.6 – Tableau de comparaison des coûts d'une multiplication sur $\mathbb{F}_{p^3}$



Architecture 1



Architecture 2



Architecture 3

Figure. 4.5 – Parallélisation de la multiplication Karatsuba sur $\mathbb{F}_{p^3}$

Nous allons maintenant analyser les algorithmes pour $c = a^2$ dans $\mathbb{F}_{p^3}$.

Algorithme I.26 : Algorithme d'élévation au carré naïve sur $\mathbb{F}_{p^3}$

Données : $a = [a_0, a_1, a_2]$ avec $a_i, b_i \in \mathbb{F}_p$, ω un non résidu cubique de $\mathbb{F}_p$.
Résultat : $c = [c_0, c_1, c_2] = a^2$ avec $c_i \in \mathbb{F}_p$.

1 $c_0 = a_0^2 + 2\omega a_1 a_2 (\mathrm{mod}\, p)$
2 $c_1 = 2a_0 a_1 + \omega a_2^2 (\mathrm{mod}\, p)$
3 $c_2 = a_1^2 + 2a_0 a_2 (\mathrm{mod}\, p)$

La version naïve présente trois multiplications de moins que pour la multiplication. La figure 4.6 résume les parallélisations possibles pour les différentes architectures. On peut remarquer que l'architecture 2 est la plus optimale.

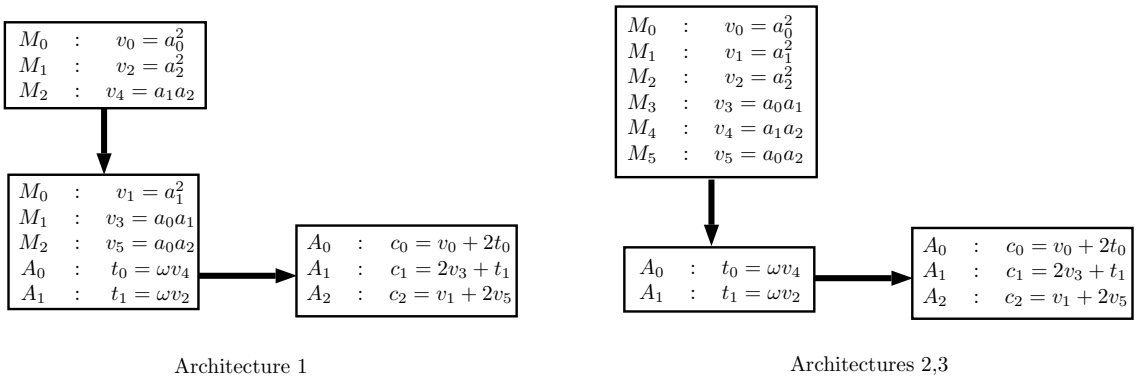


Figure. 4.6 – Parallélisation de la l'élévation au carré naïve sur $\mathbb{F}_{p^3}$

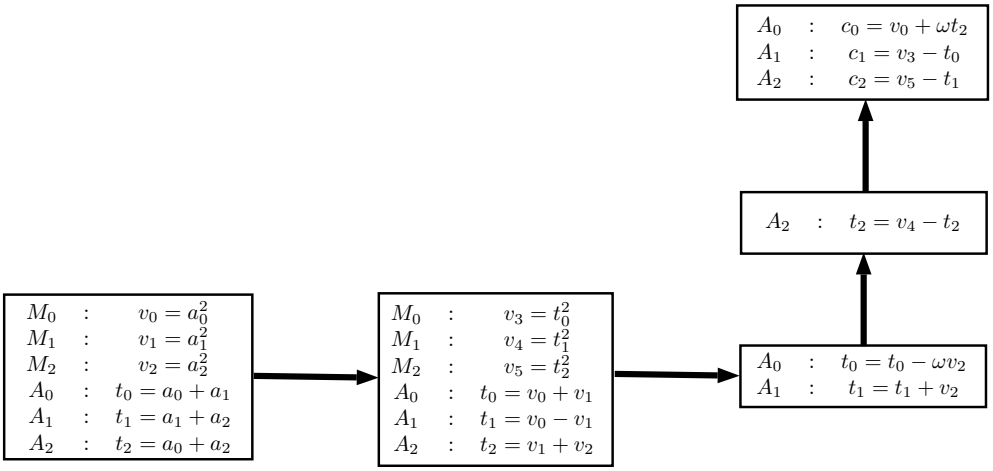
Pour l'algorithme de Karatsuba, comme pour la multiplication, il y a un surcoût en additions qui ne disparaît pas.

Algorithme I.27 : Algorithme d'élévation au carré Karatsuba sur $\mathbb{F}_{p^3}$

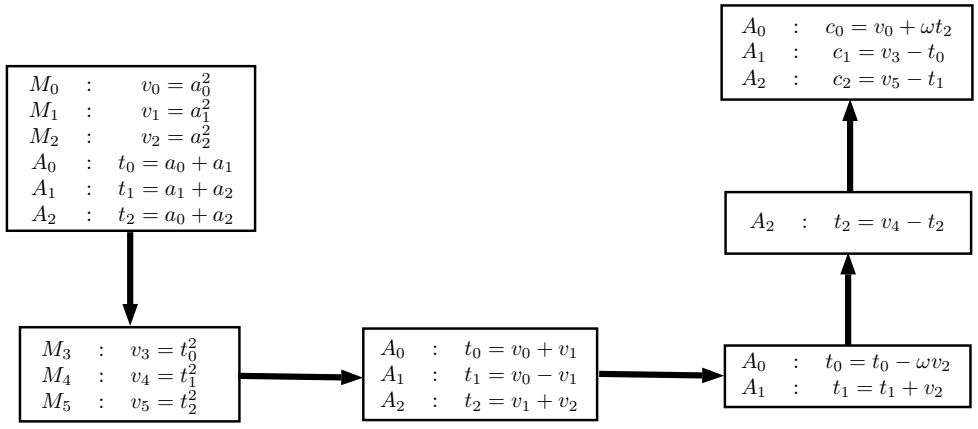
Données : $a = [a_0, a_1, a_2]$ avec $a_i \in \mathbb{F}_p$, ω un non résidu cubique de $\mathbb{F}_p$.
Résultat : $c = [c_0, c_1, c_2] = a^2$ avec $c_i \in \mathbb{F}_p$.

1 $v_0 = a_0^2 (\mathrm{mod}\, p)$
2 $v_1 = a_1^2 (\mathrm{mod}\, p)$
3 $v_2 = a_2^2 (\mathrm{mod}\, p)$
4 $c_0 = v_0 + \omega((a_1 + a_2)^2 - v_1 - v_2)(\mathrm{mod}\, p)$
5 $c_1 = (a_0 + a_1)^2 - v_0 - v_1 + \omega v_2 (\mathrm{mod}\, p)$
6 $c_2 = (a_0 + a_2)^2 - v_0 + v_1 - v_2 (\mathrm{mod}\, p)$

La figure 4.7 montre les résultats de parallélisation. L'architecture 2 est de nouveau optimale pour cet algorithme.



Architecture 1



Architectures 2,3

Figure. 4.7 – Parallélisation de la l'élévation au carré Karatsuba sur $\mathbb{F}_{p^3}$

Le tableau 4.7 présente les différents coûts observés pour chaque architecture et pour chaque algorithme. Pour une version non parallélisée, l'algorithme naïf est le plus rapide et il en est de même pour les autres architectures.

Algorithme	Non parallélisée	Architecture 1	Architecture 2
Naïve	6M + 6A	2M + 1A	**1M + 2A**
Karatsuba	**6M + 13A**	2M + 3A	**1M + 4A**

Tableau 4.7 – Tableau de comparaison des coûts d'une élévation au carré sur $\mathbb{F}_{p^3}$

4.2.4 Impact de l'ordre de création de la tour

Nous allons voir maintenant les différentes possibilités que nous avons pour construire $\mathbb{F}_{p^6}$ en utilisant la méthode de tour d'extensions. Nous verrons les résultats pour la multiplication, mais il est possible d'appliquer la même technique pour l'élévation au carré. Tout d'abord, les écritures que l'on peut avoir pour 6 sont :

- $6 = 2 \times 3$,
- $6 = 3 \times 2$,

En utilisant les résultats des tableaux 4.4 et 4.6, on peut aisément retrouver les coûts de la multiplication sur $\mathbb{F}_{p^6}$. Prenons par exemple le cas de la construction comme une extension quadratique de $\mathbb{F}_{p^3}$. Si on utilise pour les deux extensions les algorithmes de multiplication naïve, on obtient :

$$C(M, \mathbb{F}_{p^6}) = 4C(M, \mathbb{F}_{p^3}) + 2A = 4(9M + 6A) + 2A = 36M + 24A.$$

Les tableaux de l'annexe B résument les différents coûts obtenus pour toutes les combinaisons possibles. On peut donc choisir en fonction des possibilités de parallélisation, la version qui convient le mieux à la cible.

4.3 Conclusion

Nous avons analysé dans ce chapitre différentes implantations possibles pour l'arithmétique de $\mathbb{F}_p$ mais également pour certaines de ses extensions. Dans le contexte des courbes elliptiques, nous avons décidé de certains algorithmes qui sont plus adaptés aux tailles que nous manipulons. Nous utiliserons la représentation de Montgomery avec une arithmétique multi-précision avec les critères suivants :

- La représentation signe-magnitude sera utilisée pour coder les entiers. Le signe s vaudra 1 pour les entiers positifs et -1 sinon. 0 sera représenté par $(1, (0, \ldots, 0))_b$.
- La base b utilisée sera dépendante de la cible et sera discutée dans les chapitres concernant les implantations logicielles et matérielles. Cependant ce sera toujours une puissance de 2.

- Seule la multiplication naïve sera implantée car la borne d'utilisation de l'algorithme de Karatsuba est plus grande que les tailles utilisées dans notre contexte.

Dans le cadre des extensions, et plus précisément les extensions quadratiques et cubiques, nous avons décidé d'implanter les versions non parallélisées pour notre version logicielle. Pour l'architecture matérielle, nous avons utilisé les versions les plus optimales pour les architectures 1 et 2. Cependant, les coûts les plus rapides peuvent être différents selon les cibles d'implantation. En effet, nous avons vu que, dans certains cas, la différence de coûts entre deux algorithmes pouvait se résumer à une multiplication remplacée par plusieurs additions. Or, comme nous l'avons expliqué, selon l'implantation, le rapport entre une multiplication et une addition peut varier et influencer le choix de l'algorithme optimal, comme dans le cas de l'utilisation de la version Karatsuba de la multiplication-précision.

Chapitre 5

Implantation logicielle

Sommaire

Concernant l'arithmétique multi-précision, l'une des bibliothèques les plus utilisées est `GMP` [GMP]. C'est une bibliothèque implantée en `C` qui gère l'arithmétique multi-précision d'entiers signés ou non, de nombres rationnels et flottants. L'un des principaux domaines d'application concerne la cryptographie. Les performances sont réalisées en utilisant des algorithmes rapides mais également du code assembleur optimisé pour différentes architectures. Cependant, il n'y a que peu de fonctions qui traitent des opérations modulaires.

Parmi les bibliothèques gérant les courbes elliptiques, on retrouve en autre [MAGMA ; SAGE ; PARI]. Nous allons présenter en particulier `PARI/GP`. C'est un système de calcul formel initialement développé par Henri Cohen et ses collaborateurs mais qui compte aujourd'hui un grand nombre de contributeurs. Il implante notamment des algorithmes pour l'arithmétique des courbes elliptiques sous forme Weierstrass. Il peut être utilisé sous trois formes :

- une bibliothèque `libpari` implantée en `C` ;
- un calculateur programmable `gp` avec un langage de programme assez haut niveau ;
- un compilateur `gp2c` qui permet de traduire du code de `gp` en `C` et de le charger dans l'interpréteur de `gp`. Les performances sont en général meilleures que celles d'un script qu'on utiliserait directement dans `gp`.

Dans ce chapitre, nous allons présenter notre contribution à la bibliothèque logicielle `MPHELL` qui prend en charge l'arithmétique des courbes elliptiques. Nous détaillerons les fonctionnalités implantées ainsi que les extensions qui sont possibles. Nous comparerons notre implantation par rapport aux deux bibliothèques présentées précédemment, `GMP` et `PARI/GP`. Nous rappelons que les choix d'implantation ont été discutés dans les chapitres 2 et 4.

5.1 Architecture

`MPHELL` [1], Multi-Precision for (Hyper) ELLiptic curves est une bibliothèque développée à l'Institut Fourier depuis 2009. Elle est actuellement implantée en `C` et ne gère que le cas des courbes elliptiques. L'objectif est d'avoir une implantation performante, modulable mais également extensible avec le support de plusieurs systèmes de coordonnées et de formes pour l'arithmétique des courbes.

1. financée par le pôle MSTIC de l'Université Joseph Fourier, le projet FUI SHIVA et le LabEx PERSYVAL-Lab

Avec la nouvelle architecture, les fonctionnalités peuvent être divisées en quatre catégories :

- La génération de nombres aléatoires,
- L'arithmétique multi-précision,
- L'arithmétique des corps,
- L'arithmétique des courbes.

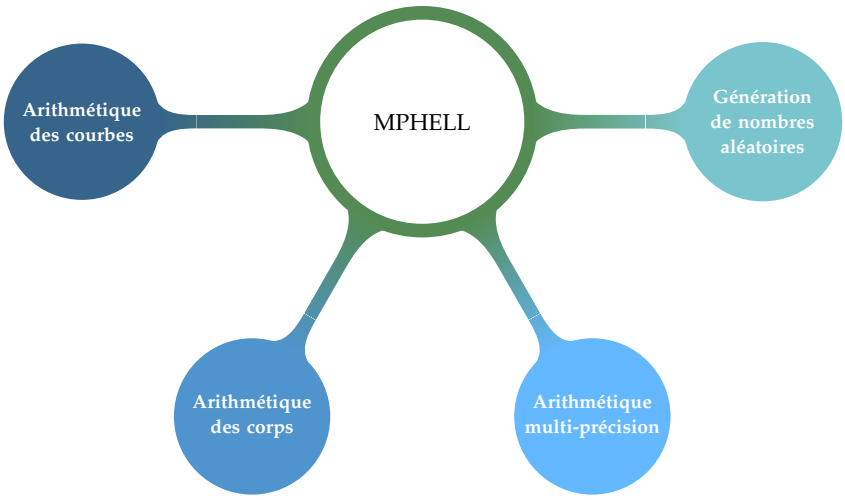
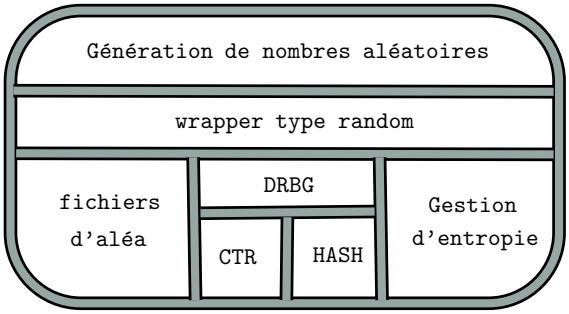


Figure. 5.1 – Architecture de la bibliothèque MPHELL

5.1.1 Génération de nombres aléatoires



Deux types de générateurs sont actuellement implantés :

- les DRBG basés sur les fonctions de hachage,
- les DRBG basés sur les fonctions CTR (en particulier AES [AES]).

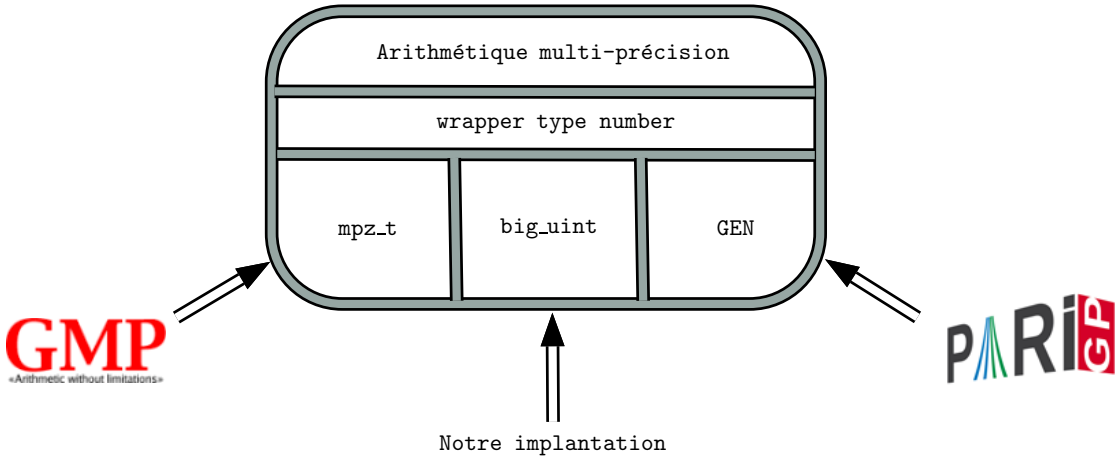
Ces générateurs suivent les recommandations du SP 800-90A [SP800-90A]. L'utilisation de ces générateurs requiert une source entropie. Il est possible d'utiliser :

- le générateur d'INTEL s'il est disponible,
- la lecture d'un fichier d'aléa,
- le fichier `/dev/urandom`,
- le fichier `/dev/hrandom1`.

Les fonctions sont accessibles grâce à un wrapper. Cela permet donc de simplifier l'utilisation par les autres parties de la bibliothèque et de rajouter aisément de nouveaux générateurs.

5.1.2 Arithmétique multi-précision

Au niveau de cette fonctionnalité, différentes options sont disponibles car il est possible de brancher une autre bibliothèque afin par exemple dans le cadre d'un projet où une certaine bibliothèque est requise. Actuellement, l'implantation supporte `GMP` et `PARI/GP`. Au moment de la compilation, les options `USE_GMP` et `USE_PARI` permettent de changer le comportement de la bibliothèque. Par défaut c'est le type `big_uint` qui est utilisé.



`big_uint` est le type que nous utilisons pour représenter l'entier multi-précision. La base de représentation peut être changée au moment de la compilation en spécifiant la taille de bloc que l'on souhaite utiliser. Les entiers sont représentés par une structure avec les champs suivants :

- `size` : le nombre de `block` utilisés pour représenter le nombre,
- `sign` : le signe du nombre, (-1 ou 1),
- `limb` : le tableau de `block` qui représente le nombre.

La majorité des opérations se fait sur le type `block` avec des fonctions en assembleur qui s'adaptent en fonctions des options de compilations `BLOCK=32,64` et `ARCH=ARM,32,64`. Les fonctions suivantes ont été implantées :

- allocation, libération
- initialisation
- conversion avec les chaînes de caractères,
- addition, soustraction,
- multiplication, square,
- division, modulo,
- réduction de Montgomery, inverse modulaire, Legendre

Le type `number` est le wrapper qui permet d'accéder aux fonctions de l'arithmétique multi-précision.

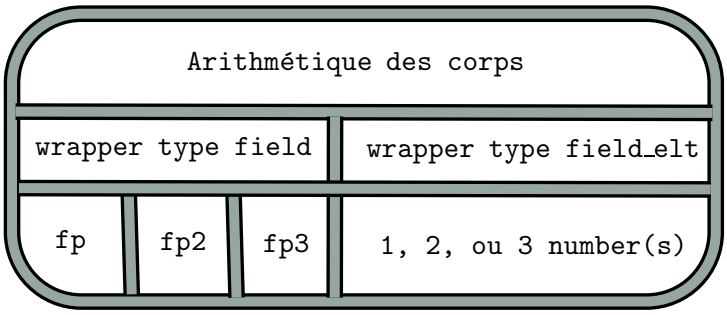
```
#if USE_GMP == 1
#include <gmp.h>
typedef mpz_t number;
typedef __mpz_struct number_t;
#define LIMB(x) x->_mp_d
#define SIZ(x) x->_mp_alloc
#define SIGN(x) x->_mp_size >= 0 ?  1 : -1
#elif USE_BIG_UINT == 1
#include "../big_uint/big_uint.h"
typedef big_uint number;
typedef big_uint_t number_t;
#define LIMB(x) x->limb
#define SIZ(x) x->size
#define SIGN(x) x->sign
#else
#include <pari/pari.h>
typedef GEN number[1];
typedef GEN number_t;
#define LIMB(x) ((*x) + 2)
#define SIZ(x) (lg(*x) - 2)
#define SIGN(x) signe(*x) != 0 ? signe(*x) : 1
#endif
```

5.1.3 Arithmétique des corps

Les corps traités actuellement sont :

- les corps premiers,
- les extensions quadratiques de corps premiers,
- les extensions cubiques de corps premiers.

Les extensions sont définies en utilisant un non-résidu quadratique ou cubique. Le type de base pour représenter les éléments des corps est le type `number`. Pour l'arithmétique des corps premiers, la représentation de Montgomery est utilisée quand la bibliothèque est configurée pour les `big_uint` en utilisant les algorithmes présentés dans le chapitre précédent.



Un wrapper `field` permet de gérer de manière générique les corps, que ce soit $\mathbb{F}_p$, $\mathbb{F}_{p^2}$ et $\mathbb{F}_{p^3}$. Pour ce faire, un corps est une structure générique contenant les champs suivants :

- `id` : une chaîne de caractères qui stocke l'id du corps,
- `type` : indique le type du corps. Les valeurs possibles sont `FP`, `FP2`, `FP3`,
- `size` : indique la taille en nombre de `block` pour stocker les éléments de base du corps,
- `param` : pointe vers une structure contenant les différentes variables nécessaires aux calculs sur le corps. Le type de structure varie en fonction du type du corps.

Cela permet de définir facilement des courbes sur n'importe quel type de corps sans avoir à modifier tout le code.

Les éléments d'un corps sont définis par une structure avec les champs suivants :

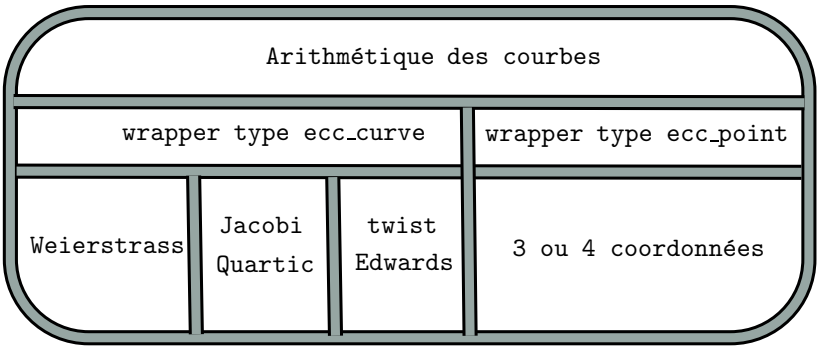
- `type` : indique le type du corps auquel appartient l'élément,
- `v` : un type union qui renvoie vers une structure particulière en fonction du type de corps.
- `k` : pointe vers le corps sur lequel l'élément est défini.

```
void
field_elt_inc (fe_ptr dst, fe_srcptr src)
{
    switch(src->type)
    {
        case FP :
            fp_elt_inc((dst->v).n, (src->v).n, src->k->param);
            break;

        case FP2 :
            fp2_elt_inc((dst->v).n2, (src->v).n2, src->k->param);
            break;

        case FP3 :
            fp3_elt_inc((dst->v).n3, (src->v).n3, src->k->param);
            break;
    }
    dst->k = src->k;
}
```

5.1.4 Arithmétique des courbes

Les courbes traitées actuellement sont :

- les courbes Weierstrass en coordonnées projectives, jacobiennes, co-Z et unifiées
- les courbes tordues d'Edwards,
- les courbes quartiques de Jacobi étendues.



Un wrapper permet de gérer de manière générique les courbes. Pour ce faire, une courbe est une structure générique contenant les champs suivants :

- `id_curve` : une chaîne de caractères qui stocke l'id de la courbe,
- `type` : indique le type de courbe à utilisé.
- `form` : indique le type de coordonnés. Les valeurs possibles sont `PROJECTIVE`, `JACOBIAN`, `CO-Z` et `UNIFIED`
- `k` : le corps sur lequel la courbes est définie (type `field`),
- `a,b` : les coefficients de la courbe qui sont de type `field_elt`,
- `G,q` : un générateur de la courbe et son ordre qui sont renseignés de manière optionnelle. Si un générateur est utilisé, la génération de points aléatoire utilise ce générateur.
- `disc` : le discriminant de la courbe.

```
void
ec_point_add (ec_point P3, const ec_point P1, const ec_point P2,
        const ec_curve E)
{
    switch(E->type)
    {
        case JACOBI_QUARTIC :
            jacobi_quartic_point_add(P3, P1, P2, E);
            break;
        case EDWARDS :
            edwards_point_add(P3, P1, P2, E);
            break;
        case WEIERSTRASS :
            weierstrass_point_add(P3, P1, P2, E);
            break;
    }
}
```

5.2 Tests et performances

Les tests de correction ont été réalisés à l'aide de fichiers de données générées aléatoirement pour lesquelles les résultats des opérations ont été pré-calculées en utilisant la bibliothèque `GMP`, celle de `PARI/GP` ou `MAGMA`. Pour les courbes, nous avons utilisé les

courbes du NIST, celles de Brainpool et des courbes générées aléatoirement. Chaque fonctionnalité a été testée séparément et nous avons également vérifié les wrappers au niveau des branchements avec les autres bibliothèques.

Les tests de performance ont été réalisés par rapport à `GMP` version 6.1.0 et `PARI/GP` version 2.7.5. Nous avons également comparé notre implantation à celle de la bibliothèque `MIRACL`. La machine utilisée est un processeur Intel core i7-4790 à 3.4GHz.

Au niveau de l'arithmétique multi-précision, nous atteignons les mêmes performances que `GMP` mais notre implantation est plus rapide au niveau de la multiplication. Nous avons également vérifié que l'utilisation de notre wrapper sur `GMP` n'avait pas une influence notable sur les performances. Concernant `PARI`, la version utilisée est basée sur `GMP`. Cependant, il y a un surcoût lié à la gestion de la pile et à l'implantation des wrappers qui fait que les performances sont moindres par rapport à `MPHELL` et `GMP`.

Les figures 5.2 et 5.3 présentent nos résultats pour les coordonnées projectives et jacobiennes obtenus sur des courbes de différentes tailles.

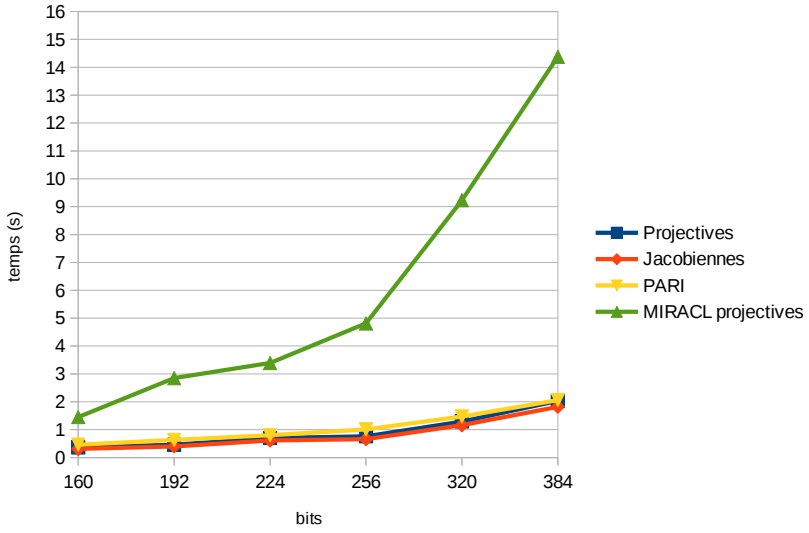


Figure. 5.2 – Comparaison de temps de la multiplication scalaire naïve sous forme Weierstrass (1)

Pour ces résultats, le même algorithme de multiplication scalaire est utilisé, la multiplication scalaire naïve. Nous constatons un grand écart avec la bibliothèque `MIRACL` pouvant aller jusqu'à un facteur proche de 8. Comme les résultats théoriques le laissaient présager, les coordonnées jacobiennes sont plus rapides que les projectives. Si l'on se compare à l'implantation de `PARI`, nous obtenons de meilleurs résultats. Nous sommes en moyenne 30% plus rapides. Cela s'explique par le fait que les opérations sont réalisées en coordonnées affines au sein de `PARI`.

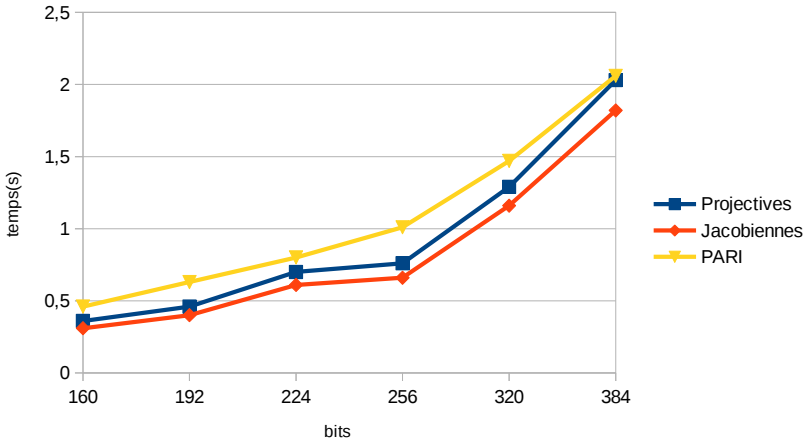


Figure. 5.3 – Comparaison de temps de la multiplication scalaire naïve sous forme Weierstrass(2)

Afin de déterminer un gain minimal que l'on pourrait obtenir en implantant dans `PARI` d'autres systèmes de représentation, il est intéressant de comparer les résultats obtenus en configurant `MPHELL` de manière à utiliser les fonctions de base de `PARI`. Nous avons constaté un gain moyen de 20%. La possibilité de brancher d'autres bibliothèques sur `MPHELL` permet donc d'obtenir une plate-forme de test de formes ou de systèmes de coordonnées.

Nous avons également comparé les temps de multiplication scalaires obtenus en utilisant d'une part les formules unifiées pour Weierstrass et d'autre part celles pour les courbes quartiques de Jacobi étendues. La figure 5.4 présente les résultats obtenus. Nous retrouvons ainsi le bénéfice d'utiliser les courbes quartiques de Jacobi étendues quand cela est possible.

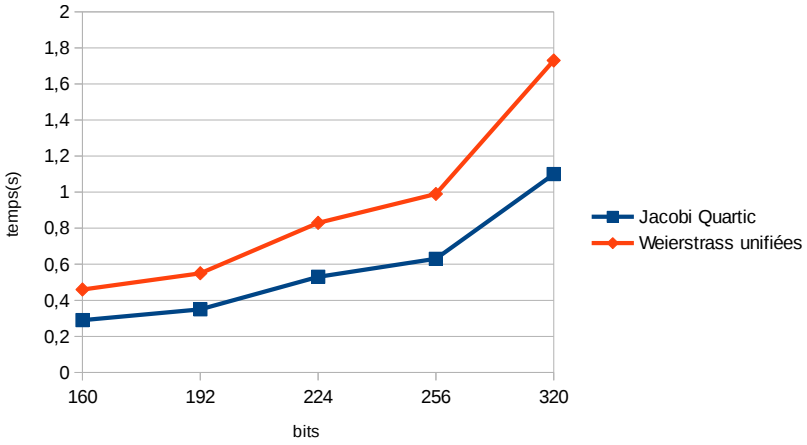


Figure. 5.4 – Comparaison de temps de la multiplication scalaire naïve avec formules unifiées

Chapitre 6

Implantation matérielle

Sommaire

L'implantation de l'arithmétique des courbes elliptiques au niveau matériel doit répondre à des critères de performances mais aussi de résistance aux attaques par canaux cachés. Dans la littérature, il est possible de distinguer les implantations en fonction des courbes visées (NIST ou génériques), des cartes utilisées (grande ou petite capacité), des unités de calculs de base (slices ou DSPs). Le tableau 6.1 regroupe quelques travaux basés sur des cartes du constructeur XILINX ainsi que leurs caractéristiques.

Article	Cible	Courbes	Logique	f (MHz)	Temps (ms)
[OP01]	XCV1000E	NIST-192	5708 LS	40	3
[GP08]	XC4VSX55-12	NIST-224	24452 sl / 468 DSPs	372	0.0265
[GP08]	XC4VSX55-12	NIST-256	24574 sl / 512 DSPs	375	0.04
[GACS09]	Virtex 4	256 bits	13661 sl / 0 DSP	43	9,2
[GACS09]	Virtex 4	256 bits	20123 sl / 0 DSP	43	7,7
[MLPJ13]	Virtex 4	256 bits	4655 sl / 37 DSPs	250	0.44
[MLPJ13]	Virtex 5	256 bits	1725 sl / 37 DSPs	291	0.38

Tableau 6.1 – Comparaison de différents processeurs FPGA pour une multiplication scalaire de courbes sur $\mathbb{F}_p$

On peut remarquer qu'avec le progrès des cartes, les performances ont évolué tant au niveau de l'espace occupé, que du temps d'exécution.

L'une des opérations des plus coûteuses dans l'arithmétique des courbes est la multiplication modulaire. Le tableau 6.2 présentent certaines implantations de l'algorithme de Montgomery ainsi que leurs performances.

Article	Cible	Taille	Logique	f (MHz)	Temps(ms)
[BP01]	XC40250XV	512	3413 CLBs	.	2.93 (max)
[BP01]	XC40250XV	1024	6633 CLBs	.	11.95 (max)
[TTL03]	XC2V3000-6	512	8235 slices, 32 mult.	99.26	0.59 (moy)
[TTL03]	XC2V3000-6	1024	14334 slices, 62 mult.	90.11	2.33 (moy)
[SM11]	XC5VLX30T-1	512	3237 slices, 17 DSPs	450	0.232 (max)
[SM11]	XC5VLX30T-1	1024	3237 slices, 17 DSPs	450	1,52 (max)
[SIN11]	XC6VLX240T-1	512	201 SR, 374 SL, 1 DSP	410.678	1.736 (max)
[SIN11]	XC6VLX240T-1	1024	201 SR, 374 SL, 1 DSP	410.678	11.263 (max)

Tableau 6.2 – Comparaison d'implantations d'exponentiation modulaire sur FPGA

Nous allons présenter dans ce chapitre, nos choix d'implantation pour une arithmétique performante pour les courbes quartiques de Jacobi. Après un rappel des cibles utilisées, nous détaillerons l'algorithme de multiplication modulaire implanté. Ensuite, les différents modules implantés seront introduits ainsi que la parallélisation de l'addition

de point que nous avons réalisée. Enfin nous donnerons les performances que nous arrivons à atteindre ainsi que les perspectives d'évolution.

6.1 Nos cibles

Les cibles que nous avons décidé d'utiliser sont les cartes Virtex 5 et 7 du constructeur XILINX [Xil15b ; Xil15a] car nous avions déjà à notre disposition certains modèles de ses cartes, une `XC5VLX50T-1FFG1136` et une `XC7VX690T-2FFG1761C`. La figure 6.1 présente une architecture simplifiée des cartes.

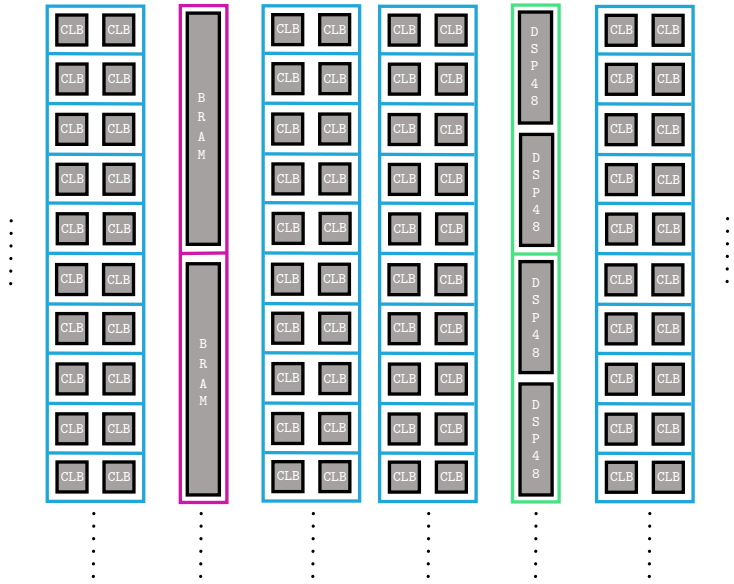


Figure. 6.1 – Architecture simplifiée d'un Virtex 5 ou 7

Notre implantation s'est basée sur l'exploitation des possibilités de calculs liées à l'utilisation des DSP48E(1) [Xil12 ; Xil14a](voir figure 6.2).

Parmi les fonctionnalités disponibles, on retrouve :

- un multiplieur 25×18,
- un additionneur avec une taille d'opérandes jusqu'à 48 bits,
- une fonction de décalage de 17 bits vers la droite,
- un contrôle dynamique des modes utilisés.

L'un des avantages de ces unités de calcul, est qu'il est possible de les mettre en cascade afin d'implanter des fonctions d'arithmétique multi-précision ou des filtres. De plus, la

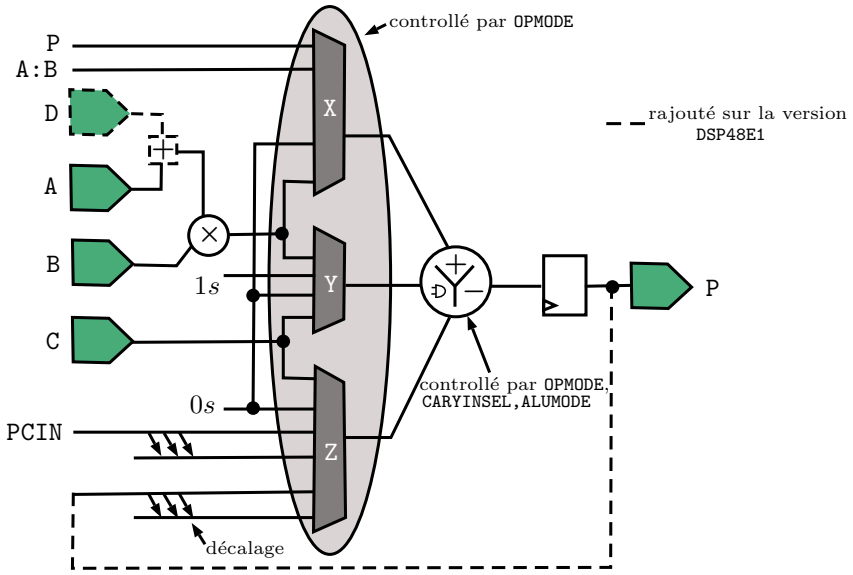


Figure. 6.2 – Architecture simplifiée d'un DSP48E1

programmation du pipeline des données d'entrées et des résultats intermédiaires permet de maintenir le débit de la cascade.

La figure 6.3 présente le diagramme de temps d'un DSP48E(1) lorsque tous les registres sont activés [Xil12]. On peut remarquer que le résultat est disponible sur P, trois cycles après l'arrivée des données sur les entrées A et B.

Afin de tirer un potentiel maximal des DSP48E(1)s, il faut que notre architecture respecte certains critères :

1. Il faut les cadencer à leur fréquence maximale et ne pas laisser des unités inactives. Quand on les utilise pour la multiplication, on doit utiliser tous les registres [Xil14b ; Xil16]. Les fréquences maximales (MHz) sont alors :

	Virtex 5			Virtex 7			
Vitesse	-3	-2	-1	-3	-2/-2L/-2G	-1	-1M
Fréquence	550	500	450	741.84	650.20	547.95	547.95

2. Notre architecture doit pouvoir supporter de grandes tailles d'entiers jusqu'à 640 bits.
3. La taille des bus d'entrée/sortie doit être inférieure à 36 bits de manière à simplifier l'utilisation des BRAMs.

Nous allons maintenant voir comment l'algorithme de multiplication de Montgomery a été adapté pour respecter les critères que nous venons de décrire.

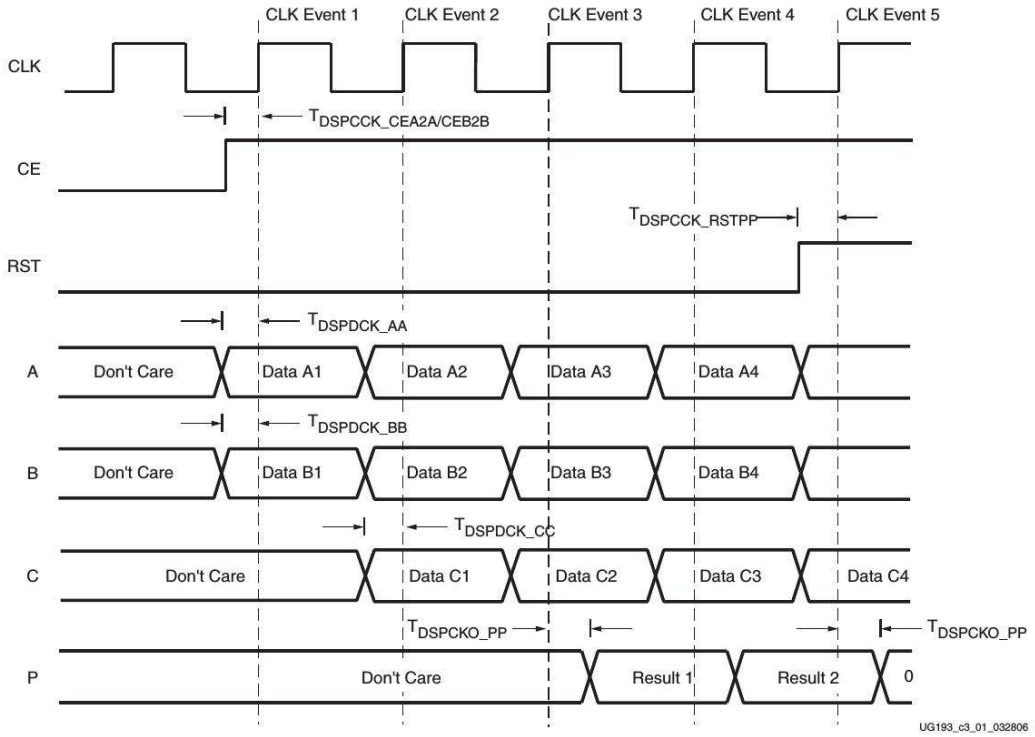


Figure. 6.3 – Diagramme de temps d'un DSP48E(1)

6.2 Multiplication de Montgomery avec quotient du pipeline

Pour permettre aux DSP48E(1)s de tourner à leur fréquence maximale, il faut mettre une certaine latence dans notre design. L'algorithme de multiplication de Montgomery présenté dans 4.1 n'est donc pas adapté. Dans [EW93 ; Oru95], les auteurs ont proposé certaines adaptations de l'algorithme d'origine afin que les opérations puissent être mises en pipeline. Nous allons présenter ici l'algorithme avec pipeline du quotient (algorithme I.28) décrit dans [Oru95].

Deux paramètres gèrent le fonctionnement de l'algorithme : la base utilisée 2^k et la latence introduite notée d. De plus, certaines valeurs doivent pré-calculées. Pour un module $M > 2$ tel que $\gcd(M, 2) = 1$ avec n blocs dans la base 2^k, on définit :

- $R = 2^{kn}$,
- M' tel que $(-MM' \bmod 2^{k(d+1)}) = 1$,
- $\widetilde{M} = (M' \bmod 2^{k(d+1)})M$, $4\widetilde{M} < R$,
- $M'' = (\widetilde{M} + 1)/2^{k(d+1)}$.

Algorithme I.28 : Multiplication de Montgomery avec quotient du pipeline

Données : A `le multiplicande;` B `le multiplicateur;` M `un module tel que` $M > 2$, $\gcd(M,2) = 1$, $0 \leqslant A, B \leqslant 2\widetilde{M}$, $B = \sum_{i=0}^{n+d} (2^k)^i b_i$, $b_i \in \{0, 1, \ldots, 2^k - 1\}$ `et` $b_i = 0$ `pour` $i \geqslant n$.

Résultat : $S_{n+d+2} = ABR^{-1} \bmod M$ `tel que` $0 \leqslant S_{n+d+2} \leqslant 2\widetilde{M}$.

1 $S_0 = 0; q_{-d} = 0; \ldots; q_{-1} = 0$

2 **pour** $i = 0$ à $n + d$ **faire**

3 $\quad q_i = S_i \bmod 2^k$

4 $\quad S_{i+1} = S_i / 2^k + q_{i-d} M'' + b_i A$

5 $S_{n+d+2} = 2^{kd} S_{n+d+1} + \sum_{j=0}^{d-1} q_{n+j+1} 2^{kj}$

6 **retourner** S_{n+d+2}

À la ligne 3, les invariants suivants sont vérifiés :

$$2^{ki} S_i + \sum_{j=i-d}^{i-1} q_j 2^{kj} = 2^k A \sum_{j=0}^{i-1} b_j 2^{kj} + \widetilde{M} \sum_{j=0}^{i-d-1} q_j 2^{kj}$$

et

$$0 \leqslant S_i < 2^k (A + M).$$

On peut donc vérifier qu'à la fin de l'algorithme :

$$S_{n+d+2} = R^{-1} (A \sum_{j=0}^{n-1} b_j 2^{kj} + \widetilde{M} \sum_{j=0}^{n-1} q_{j+1} 2^{kj}) = ABR^{-1} \bmod M.$$

et

$$0 \leqslant S_{n+d+2} \leqslant 2\widetilde{M}.$$

Le résultat de la multiplication est compris entre 0 et $2\widetilde{M}$ par conséquent, il faut faire un traitement supplémentaire s'il on veut récupérer la valeur modulo M. On peut remarquer que si $\widetilde{M} = P$, il suffit de faire une soustraction. Cependant, cela peut donner lieu à une attaque par analyse du temps d'exécution. Dans [HQ00], une contre-mesure est proposée.

Nous avons étudié les nombres premiers de la forme $M = \sum_{i=0}^{n-1} a_i (2^{k(d+1)})^{i+1} - 1$. Cela permet d'obtenir $M' = 1$ et donc $\widetilde{M} = M$. Ces conditions peuvent se résumer à :

- Le premier bloc de $k(d+1)$ bits vaut $2^{k(d+1)} - 1$.
- Pour les blocs suivants, les k bits de poids forts sont à 0.
- Pour une taille s qui n'est pas un multiple de k, on garde la forme jusqu'au dernier bloc de k bits, puis on complète en mettant un bit à 1 à la taille désirée.

6.2.1 Adaptation à la cible

Soient h le nombre de bits de M et h' celui de A et B. Le nombre de blocs n à traiter est donc $n = \lceil h'/k \rceil$. D'après les conditions sur les entrées de l'algorithme, ces deux valeurs sont liées par l'équation suivante :

$$h' = h + k(d+1) + 1.$$

Les DSP48E(1)s possèdent une fonction de décalage de 17 bits. Étant donné que dans l'algorithme il y a une division par 2^k, la base utilisée sera $2^k = 2^{17}$. Pour le paramètre d, afin de pas introduire trop de latence dans notre circuit, il est fixé à 1.

L'algorithme proposé par Suzuki et Matsumoto [SM11] est une combinaison de l'algorithme I.28 et de l'architecture utilisée dans [TK99]. Ils considèrent que si α DSP48E(1)s sont utilisés dans le design, le découpage des données se fera comme suit :

- le nombre de blocs traités par un DSP48E(1) noté r, vérifie $r = 2\lceil(\lceil n/\alpha \rceil)/2\rceil$ afin que r soit pair,
- le nombre de blocs traités par α DSP48E(1)s est donc $\alpha r \geqslant n$; les blocs excédentaires doivent être mis à zéro au début de l'algorithme.

Dans l'article, les auteurs ont fixé la valeur de α à 17 car cela leur permet d'avoir des tailles de modules supérieure ou égale à 512. Dans notre cas, on peut adapter α en fonction du contexte d'utilisation (voir tableau 6.3).

h (nb bits M)	n (nb blocs)	r (nb blocs par DSP48E(1))	α (nb DSP48E(1))
128	10	2	5
192	14	2	7
256	18	2	9
384	25	2	13
512	33	2	17

Tableau 6.3 – Paramètre pour la multiplication de Montgomery sur FPGA

L'algorithme I.29 est celui proposé dans [SM11] pour leur architecture. Dans la base 2^{17}, on peut écrire :

- $A = \sum_{j=0}^{\alpha r-1} (2^{17})^j a_j$,
- $B = \sum_{j=0}^{n+1} (2^{17})^j b_j$,
- $M'' = \sum_{j=0}^{\alpha r-1} (2^{17})^j m_j$,

- $S_i = \sum_{j=0}^{\alpha r-1} (2^{17})^j s_{(i,j)}$,

avec $a_j, b_j, m_j, s_{(i,j)} \in \{0, 1, \ldots, 2^{17} - 1\}, a_j = b_j = 0$ pour $j \geqslant n$ et $m_j = 0$ pour $j \geqslant \lceil h/17 \rceil$.

Les lignes 4-5 sont en charge du calcul $b_i A$. Chaque DSP48E(1)s calcule r multiplications de la boucle, puis transmet la retenue au DSP48E(1) suivant qui poursuit le calcul Il faut mettre en cascade les DSP48E(1)s pour réaliser ce calcul. Après avoir transmis la retenue, il peut enchaîner sur sa partie de calcul de $q_{i-d}M''$ (lignes 6-10) et appliquer le même enchaînement que précédemment.

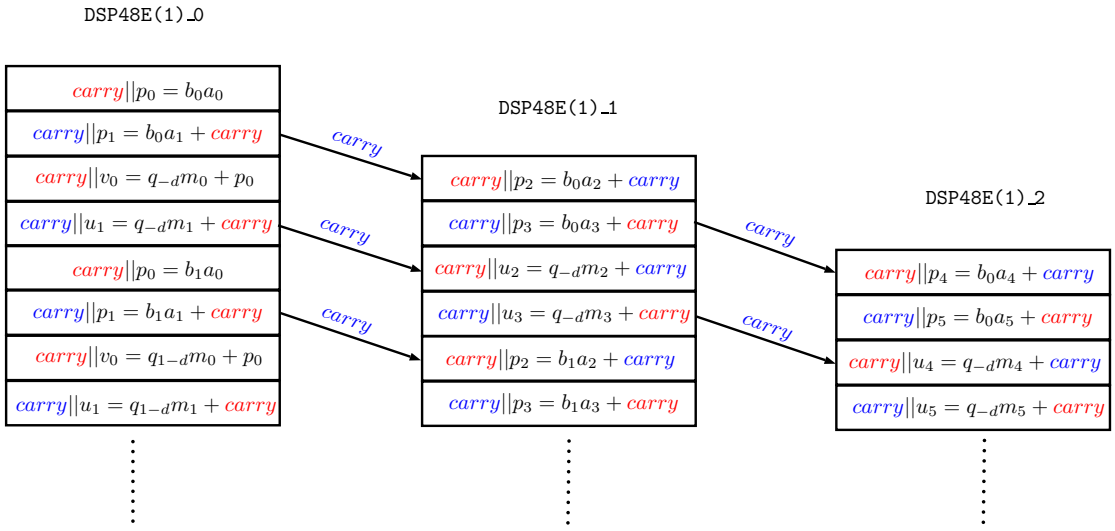


Figure. 6.4 – Pipeline apparaissant dans la multiplication de Montgomery.

À l'issue des multiplications, des additions doivent être réalisées afin d'obtenir les blocs de S_i. Cette fois-ci, ce sont deux blocs qui sont traités à chaque tour de boucle. De plus, ces opérations ne sont pas réalisées avec les DSP48E(1)s mais avec des slices LUTs.

Cet algorithme permet aux DSP48E(1)s de prendre en charge des multiplications sur un bloc à leur fréquence maximale et les additionneurs réalisent des opérations sur deux blocs à une fréquence égale à la moitié de celle des DSP48E(1)s.

Algorithme I.29 : Multiplication de Montgomery pour une cible FPGA

Données : A `le multiplicande;` B `le multiplicateur;` M `un module tel que` $M > 2,\ \gcd(M,2) = 1,\ 0 \leqslant A, B \leqslant 2\widetilde{M}$.

Résultat : $S_{n+3} = ABR^{-1} \bmod M$ `tel que` $0 \leqslant S_{n+3} \leqslant 2\widetilde{M}$.

$S_0 = 0;\ q_{-1} = 0$
pour $i = 0$ `à` $n+1$ **faire**
  $carry = 0^{17}; cv = 0; cs = 0$
  /* Multiplication multi-précision $P = b_i A$ */
  pour $j = 0$ `to` $\alpha r - 1$ **faire**
    $carry|p_j = b_i a_j + carry$
  /* Multiplication multi-précision $U = q_{i-d} M''$ */
  pour $j = 0$ `to` $\alpha r - 1$ **faire**
    si $j = 0$ **alors**
      $carry||v_0 = q_{i-d} m_j + p_0$
    sinon
      $carry||u_j = q_{i-d} m_j + carry$
  /* Calcul de q_{i+1} */
  $q_{i+1} = v_0 + s_{(i,1)}$
  /* Addition multi-précision $V = P + U$ */
  pour $j = 0$ `to` $\alpha r/2 - 1$ **faire**
    si $j = 0$ **alors**
      $c_v||v_1||v_0 = (p_1||0^{17}) + (u_1||u_0)$
    sinon
      $c_v||v_{2j+1}||v_{2j} = (p_{2j+1}||p_{2j}) + (u_{2j+1}, u_{2j}) + cv$
  /* Addition multi-précision $S_{i+1} = S_i/2^{17} + V$ */
  pour $j = 0$ `to` $\alpha r/2 - 1$ **faire**
    $c_s||s_{(i+1,2j+1)}||s_{(i+1,2j)} = (v_{2j+1}||v_{2j}) + (s_{(i,2j+2)}||s_{(i,2j+1)}) + cs$
$S_{n+3} = S_{n+2}||s_{(n+1,0)}$
retourner S_{n+3}

6.3 Module en charge des opérations modulaires

Nous allons décrire dans cette section, les modules qui ont été implantés en s'inspirant des algorithmes que nous avons étudiés précédemment.

6.3.1 Multiplication

L'architecture de notre module de multiplication modulaire est la suivante.

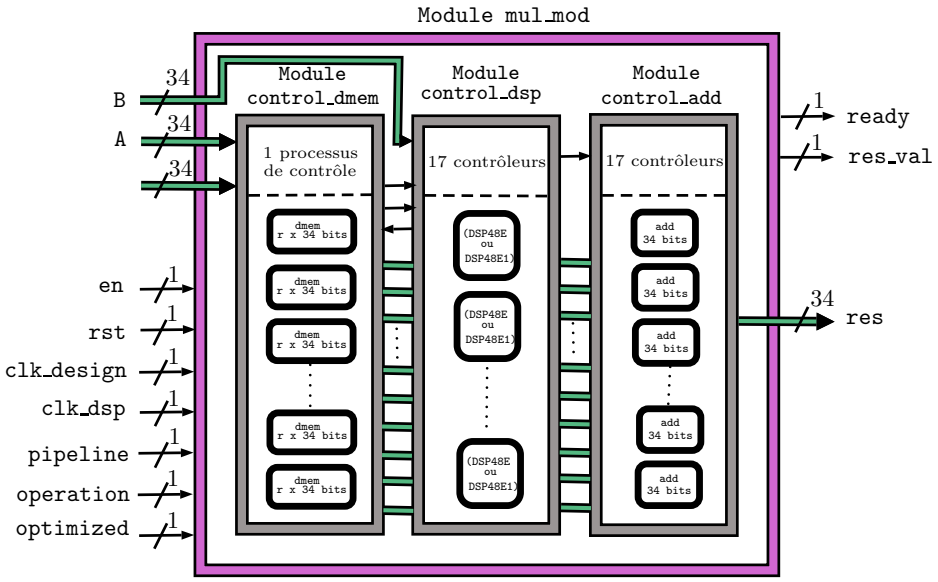


Figure. 6.5 – Architecture du module de multiplication modulaire

Nous avons divisé notre implantation en trois modules :

- un contrôleur de mémoire qui est en charge de gérer les DMEMs qui stockent les données auxquelles chaque DSP48E(1) peut accéder,
- un contrôleur de DSPs qui gère les modes et le chargement de données des DSP48E(1)s,
- un contrôleur d'addition qui prend en charge les additions nécessaires à la multiplication.

Les entrées et sorties A,B, P et R sont prévues pour être branchées sur des BRAMs. De plus deux horloges sont utilisées :

- une qui est dédiée aux DSP48E(1)s `clk_dsp`,
- et une qui cadence le reste du design `clk_design`.

6.3.2 Addition

L'architecture de notre module d'addition modulaire est la suivante.

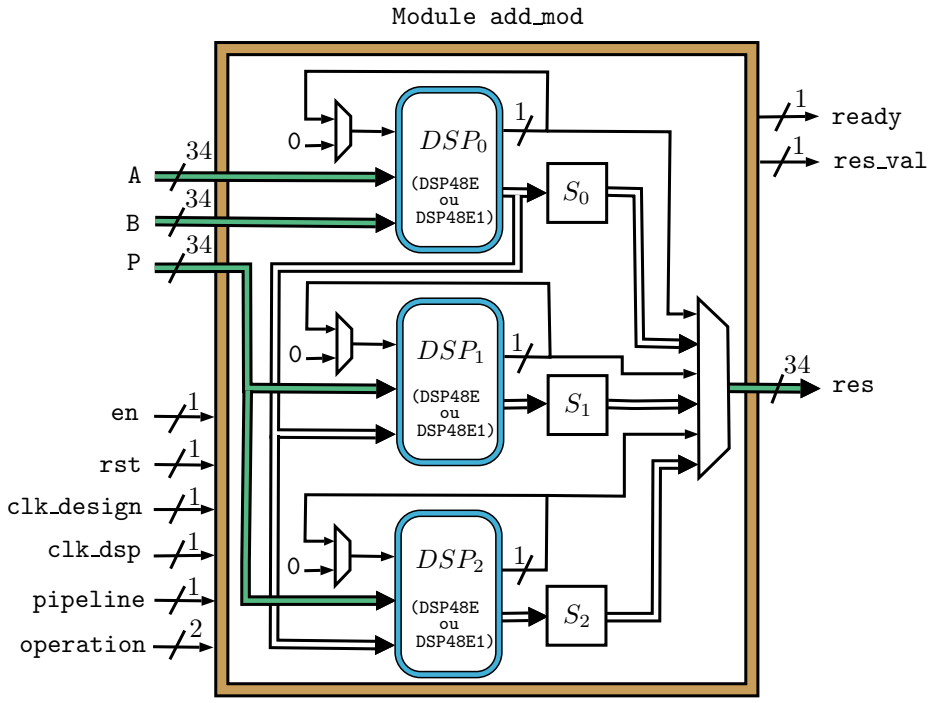


Figure. 6.6 – Architecture du module d'addition modulaire

Les fonctionnalités sont les suivantes :

- $A + B \mod P$
- $A - B \mod P$
- $2A + B \mod P$
- $2A - B \mod P$

Notre module d'addition modulaire est composé de trois DSP48E(1)s. Le premier est en charge de l'opération principal (addition et soustraction) et les deux autres s'occupent des réductions nécessaires (voir figure 6.7).

Les opérations sont réalisés sur des blocs de 34 bits afin de garder la base de représentation utilisée pour la multiplication. Les entrées et sorties A,B, P et R sont prévues pour être branchées sur des BRAMs. Comme pour la multiplication, nous avons deux horloges séparées. Le signal `ready` indique que l'opération en cours sur le premier DSP48E(1) va se terminer au prochain coup d'horloge et que par conséquent, il est possible d'envoyer de nouvelles données même si le résultat n'est pas encore disponible.

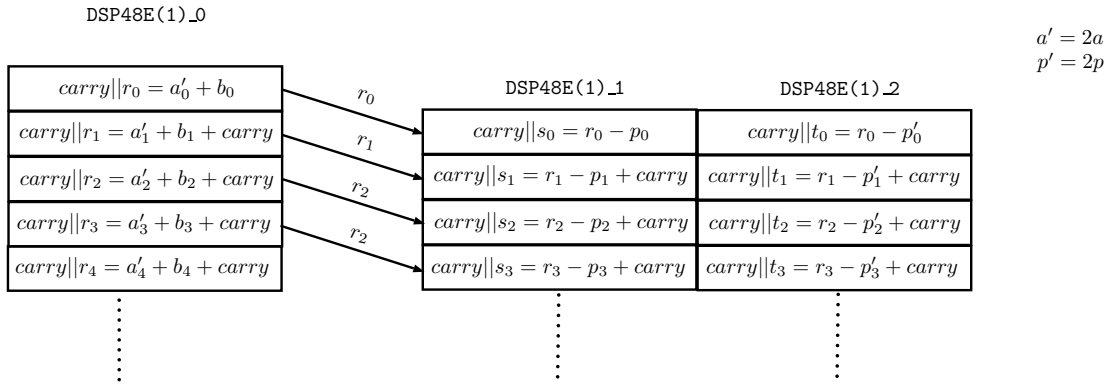


Figure. 6.7 – Pipeline des opérations lors d'une addition modulaire $2A + B$

6.3.3 Module complet

Pour l'architecture du module en charge de l'arithmétique sur $\mathbb{F}_p$ nous avons décidé d'implanter deux solutions :

- trois modules de multiplication + trois modules d'addition (Architecture 1),
- six modules de multiplication + trois modules d'addition (Architecture 2).

Ces choix sont le résultat de l'analyse de la parallélisation des opérations sur les extensions $\mathbb{F}_{p^2}$ et $\mathbb{F}_{p^3}$ que nous avons réalisée dans la section 4.2.

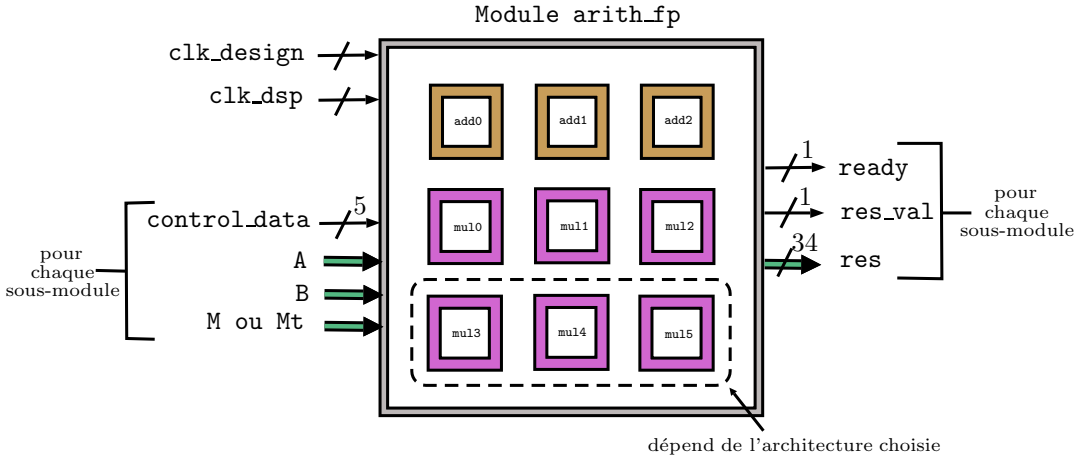


Figure. 6.8 – Architecture du module d'arithmétique sur $\mathbb{F}_p$

6.3.4 Opérations sur les extensions

Les opérations implantées s'appuient sur les stratégies de parallélisation discutées dans 4.2.

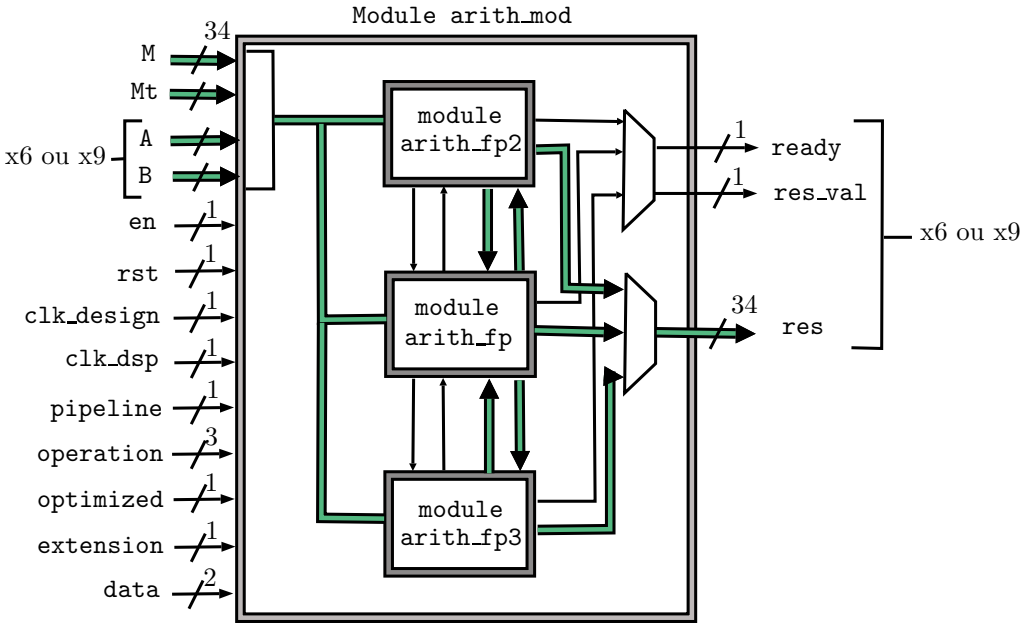


Figure. 6.9 – Architecture d'arithmétique modulaire

6.4 Module en charge de l'addition de point

6.4.1 Forme quartique de Jacobi

Avec les deux architectures à notre disposition pour l'implantation des opérateurs modulaires, nous pouvons définir deux stratégies de parallélisation de l'algorithme d'addition de point sur les courbes quartiques de Jacobi.

Pour rappel, pour $d = 1$, les formules utilisées sont :

$$
\begin{aligned}
X_3 &= (X_1Y_2 + Y_1X_2)(Z_1Z_2 - T_1T_2),\\
Y_3 &= (Y_1Y_2 + 2aX_1X_2)(Z_1Z_2 + T_1T_2) + 2X_1X_2(T_1Z_2 + Z_1T_2),\\
T_3 &= (X_1Y_2 + Y_1X_2)^2,\\
Z_3 &= (Z_1Z_2 - T_1T_2)^2.
\end{aligned}
$$

En travaillant sur l'architecture 2, on peut paralléliser l'addition et obtenir un coût en $3M + 2A$ (voir figure 6.10).

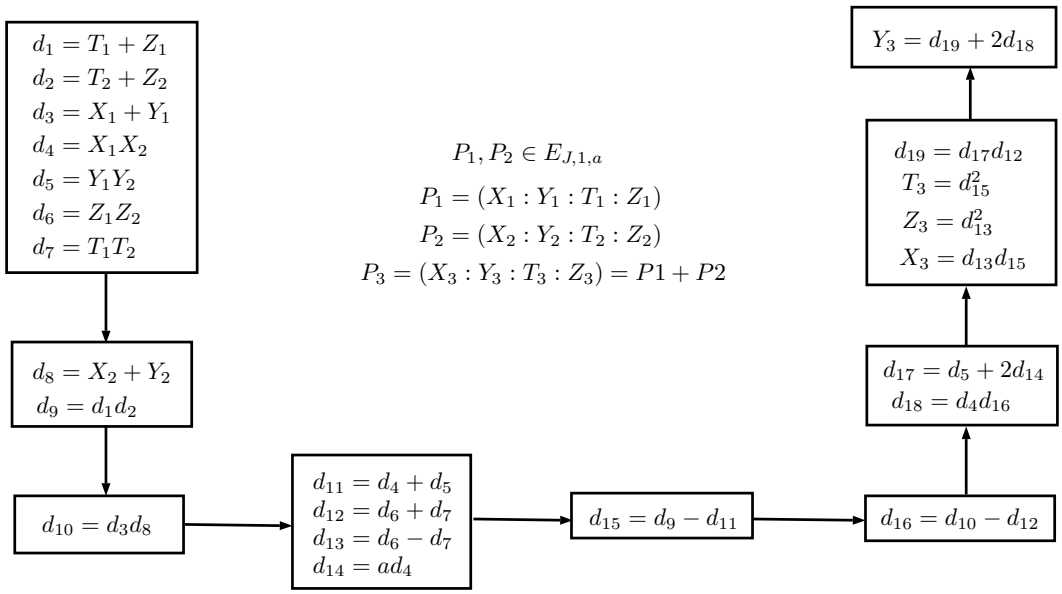


Figure. 6.10 – Parallélisation de l'addition point pour une courbe quartique de Jacobi

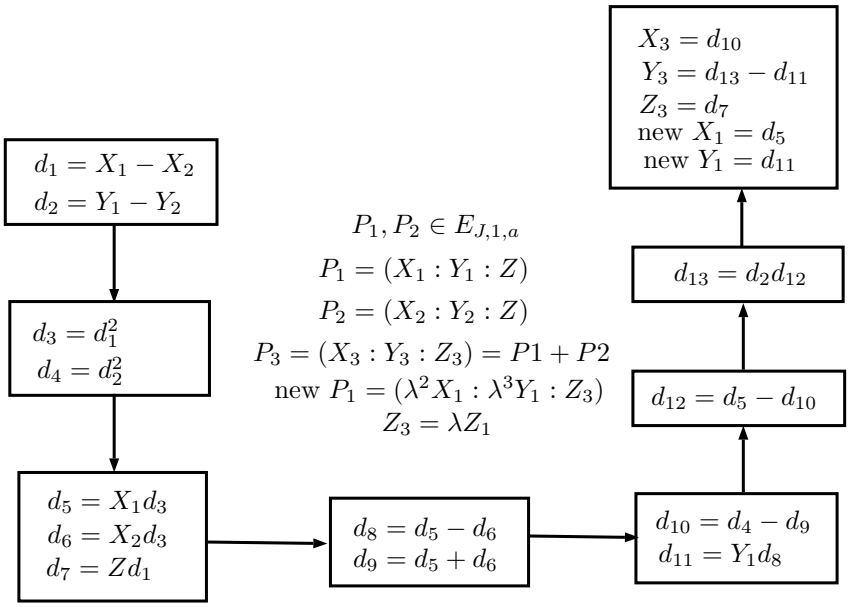
6.4.2 Addition co-Z

Pour les coordonnées co-Z nous avons décider d'implanter l'algorithme suivant pour la multiplication scalaire qui utilise les fonctions ZADDU et ZADDC d'additions de points.

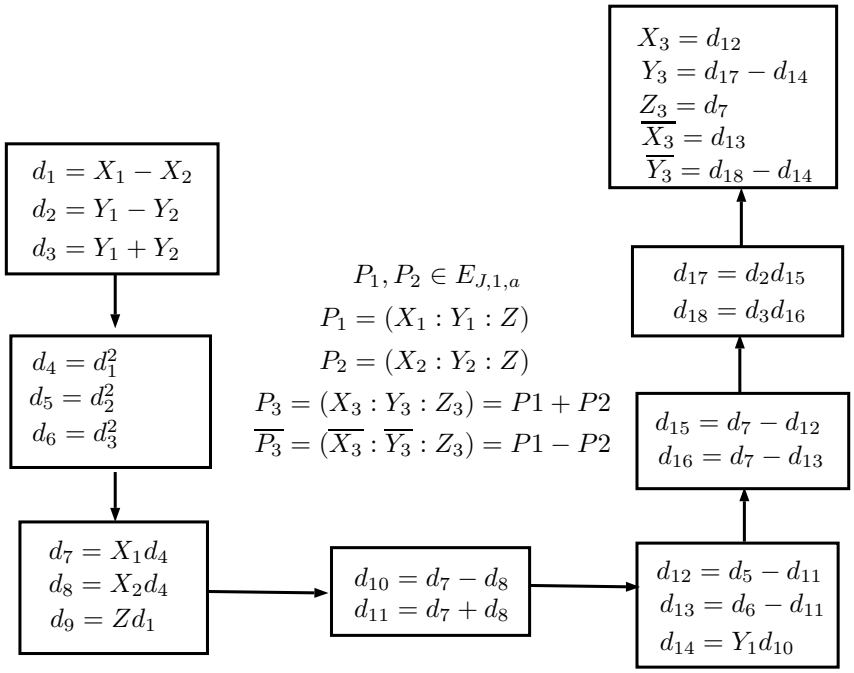
Algorithme 1.30 : Algorithme de multiplication scalaire pour les coordonnées co-Z

1 $b = k_i$, $R_b = P$ et $(R_{1-b}, R_b) = TPLU(R_b)$
2 **pour** $i = 2$ à $n - 1$ **faire**
3 $\quad b = k_i$
4 $\quad (R_b, R_{1-b}) = ZADDU(R_{1-b}, R_b)$
5 $\quad (R_{1-b}, R_b) = ZADDC(R_b, R_{1-b})$
6 **retourner** Q_0

Nous avons donc étudier la parallélisation de ces deux fonctions. Notre implantation permet d'obtenir un coût de 3M + 5A. La figure 6.11 montre le schéma adopté.



ZADDU



ZADDC

Figure. 6.11 – Parallélisation des fonctions ZADDU et ZADDC pour les formules d'addition de points co-Z

6.5 Résultats

Nous obtenons les coûts suivants pour les opérations implantées. Tout d'abord le tableau 6.4 présente les résultats pour les calculs sur $\mathbb{F}_p$. La formule reliant le coût d'une

bits	n	r	α	Addition	Multiplication
128	10	2	5	7	59
192	14	2	7	9	72
256	18	2	9	11	99
384	25	2	13	15	135
512	33	2	17	19	175

Tableau 6.4 – Cycles pour les opérations modulaires sur $\mathbb{F}_p$

multiplication aux données du contexte est

$$C(M) = 2 + 2r(n+2) + (\alpha - 1) * 2 + 1.$$

Le rapport entre le coût d'une multiplication et celui d'une addition sur $\mathbb{F}_p$ est compris entre 8 et 9. Cela nous a permis de faire un choix définitif sur les algorithmes a implanté pour les extensions lorsque la différence entre deux algorithmes reposait essentiellement sur ce rapport.

Pour les additions sur les extensions, le coût d'une addition est égale à celui sur $\mathbb{F}_p$ car dans nos deux architectures, nous avons à notre disposition trois additionneurs. Nous ne présentons donc dans le tableau 6.5 que les coûts pour la multiplication. Nous pouvons

	Multiplication $\mathbb{F}_{p^2}$		Multiplication $\mathbb{F}_{p^3}$	
bits	Architecture 1	Architecture 2	Architecture 1	Architecture 2
128	73	66	132	87
192	90	81	162	108
256	121	110	220	143
384	165	150	300	195
512	213	194	388	251

Tableau 6.5 – Cycles pour une multiplication modulaire sur $\mathbb{F}_{p^2}$ et $\mathbb{F}_{p^3}$

remarquer que nous retrouvons bien les coûts théoriques établis dans la section 4.2. Pour l'architecture 1, le rapport entre une multiplication sur $\mathbb{F}_{p^2}$ et une sur $\mathbb{F}_p$ est de l'ordre de 1,25 et pour $\mathbb{F}_{p^3}$, il est de l'ordre de 2,25. Pour l'architecture 2, le rapport entre une multiplication sur $\mathbb{F}_{p^2}$ et une sur $\mathbb{F}_p$ est de l'ordre de 1,11 et pour $\mathbb{F}_{p^3}$, il est de l'ordre de 1,43.

Pour l'addition de points, nous considérons uniquement l'architecture 2. Le tableau 6.6 résume les coûts pour des courbes sur $\mathbb{F}_p$ en version parallélisée.

bits	Addition quartique de Jacobi sur $\mathbb{F}_p$
128	191
192	234
256	319
384	435
512	563

Tableau 6.6 – Cycles pour une addition unifiée pour une courbe quartique de Jacobi sur $\mathbb{F}_p$

Sur la carte `XC7VX690T-2FFG1761C`, pour une courbe de 256 bits nous obtenons un temps de $0.98\mu s$ pour une addition de point sur $\mathbb{F}_p$.

Afin de supporter les courbes ne pouvant se mettre sous la forme quartique de Jacobi sur $\mathbb{F}_p$, nous avons implanté les formules sur $\mathbb{F}_{p^3}$. Le tableau 6.7 présente une comparaison entre une version non parallélisée, une version parallélisée ainsi qu'une estimation coût pour une addition en coordonnées Jacobiennes. Nous avons décidé d'implanter la version non parallélisée afin de fournir une alternative à la parallélisation car celle-ci nécessite un grand nombre de DSP48E(1)s. Par exemple, pour 256 bits il faut 351 DSP48E(1)s.

bits	Addition quartique de Jacobi sur $\mathbb{F}_{p^3}$ non parallélisée	Addition quartique de Jacobi sur $\mathbb{F}_{p^3}$ parallélisée	Addition Jacobienne parallélisée sur $\mathbb{F}_p$
128	1055	275	383
192	1314	342	477
256	1727	451	627
384	2355	615	855
512	3027	791	1099

Tableau 6.7 – Cycles pour une addition unifiée pour une courbe quartique de Jacobi sur $\mathbb{F}_{p^3}$

Nous pouvons remarquer qu'il y a un facteur de 1,39 entre l'addition de point en version Jacobienne dur $\mathbb{F}_p$ et celle pour les courbes quartiques de Jacobi sur $\mathbb{F}_{p^3}$. Mise à part le coût en surface qui est plus important pour la version quartique de Jacobi, nous pouvons déduire que l'implantation sur $\mathbb{F}_{p^3}$ en version parallélisée est plus rapide. Pour une courbe ne pouvant se mettre directement sous la forme quartique de Jacobi, le surcoût lié au travail sur $\mathbb{F}_{p^3}$ est complètement absorbé par notre parallélisation et les performances

sont meilleures.

Pour le moment, les algorithmes de multiplication scalaire n'ont pas encore été entièrement testés sur nos cartes. Nous pouvons cependant estimer le coûts des implantations en utilisant les précédents tableaux. Nous présentons donc dans le tableau 6.8, des coûts pour des opérandes arbitraires pour :

- une multiplication scalaire naïve avec les courbes quartiques de Jacobi sur $\mathbb{F}_p$,
- une multiplication scalaire naïve avec les courbes quartiques de Jacobi sur $\mathbb{F}_{p^3}$,
- une multiplication scalaire Double/Add Always avec les courbes Weierstrass en coordonnées jacobiennes sur $\mathbb{F}_p$.

Nous avons décidé d'utiliser telles implantations afin de faire des comparaisons avec un niveau de sécurité semblable par rapport aux attaques SPA. On peut constater que,

bits	Multiplication D/A Quartique de Jacobi sur $\mathbb{F}_p$	Multiplication D/A Quartique de Jacobi sur $\mathbb{F}_{p^3}$	Multiplication D/A-Always Jacobienne sur $\mathbb{F}_p$
128	36481	52525	97282
192	67158	98154	182214
256	122177	172733	319770
384	250125	353625	654930
512	431821	606697	1123178

Tableau 6.8 – Comparaison de nombre de cycles pour une multiplication scalaire

comme précédemment il est préférable de travailler avec l'implantation de la forme quartique de Jacobi même dans le cas de $\mathbb{F}_{p^3}$. De plus, sur notre carte Virtex 7, pour une courbe 256 bits avec notre estimation de multiplication, on obtient un temps de 0.38 ms. Notre implantation est donc de performance comparable et parfois meilleure que les travaux présentés dans le tableau 6.1.

Nous avons également comparer notre implantation aux formules d'addition de points co-Z. Tout d'abord concernant le nombre de cycles utilisés (voir figure 6.12), l'implantation des courbes quartiques de Jacobi est plus rapide que celle de l'addition co-Z pour les courbes définies sur $\mathbb{F}_p$. Concernant $\mathbb{F}_{p^3}$, on peut observer un surcoût du au travail sur l'extension cubique qui nous place au dessus de la courbe pour l'addition co-Z.

Le nombre de DSP48E(1)s utilisés est sensiblement le même pour les implantations sur $\mathbb{F}_p$. On peut remarquer une croissance exponentielle pour l'extension cubique car nous avons fait le choix de paralléliser au maximum tous nos algorithmes.

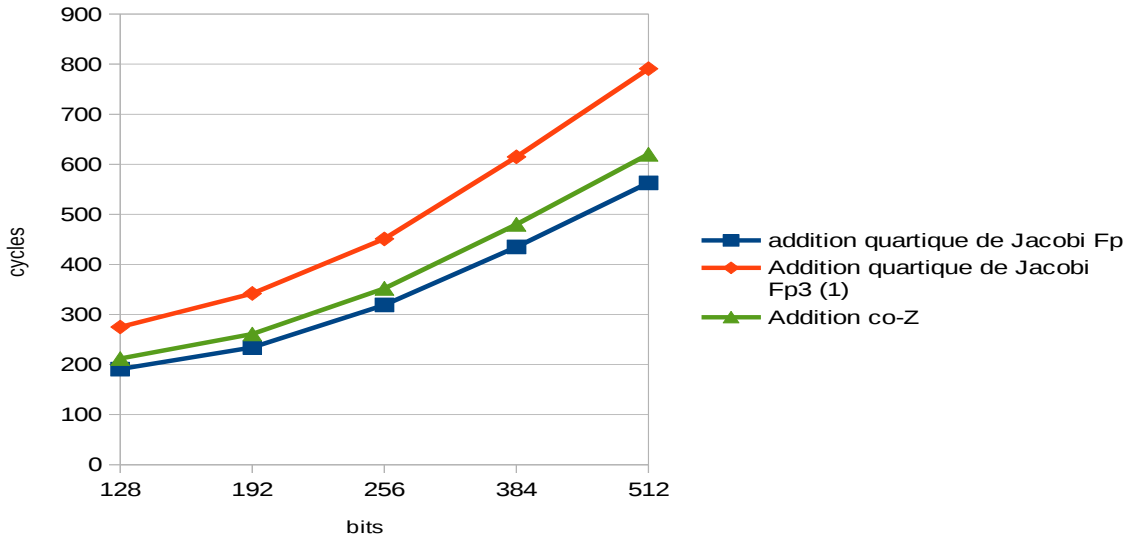


Figure. 6.12 – Comparaison en nombre de cycles des additions de points quartique de Jacobi et co-Z

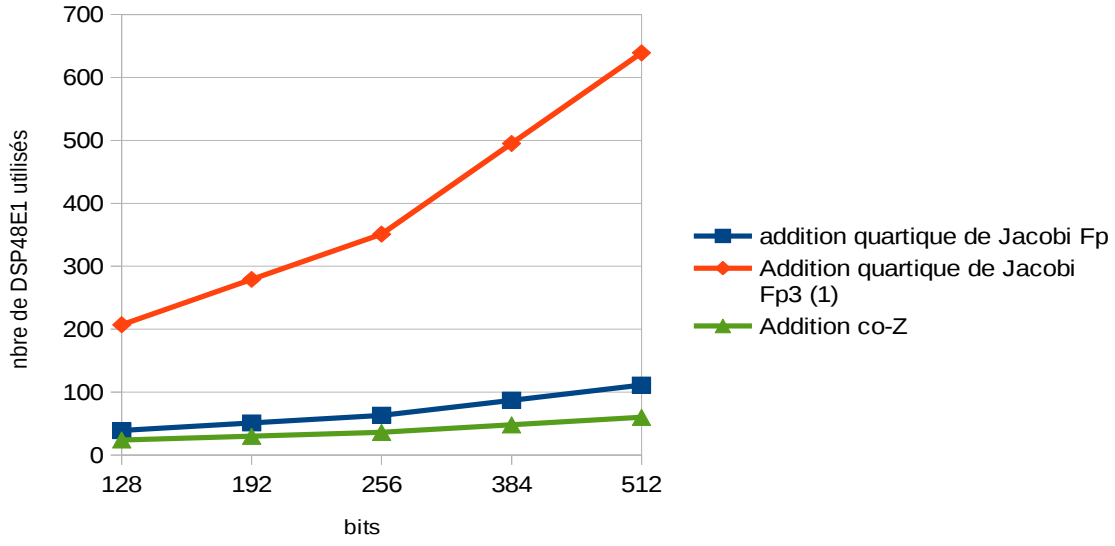


Figure. 6.13 – Comparaison en nombre de DSP48(E)1s des additions de points quartique de Jacobi et co-Z

La figure 6.14 illustre les résultats obtenus pour la multiplication scalaire en terme de nombres de cycles nécessaires pour des opérandes aléatoires. Alors que précédemment l'implantation sur $\mathbb{F}_{p^3}$ était moins efficace, nous pouvons constater qu'au niveau de la multiplication scalaire, les deux algorithmes pour l'addition de points quartique de Jacobi sur $\mathbb{F}_p$ et $\mathbb{F}_{p^3}$ sont plus rapide que la version co-Z sur $\mathbb{F}_p$. Cela s'explique par le fait qu'une seule opération est nécessaire dans le premier cas alors qu'il faut deux pour le second. On peut donc constater l'intérêt des courbes quartiques de Jacobi même dans le cas où le transfert depuis la forme Weierstrass nécessite un travail sur $\mathbb{F}_{p^3}$.

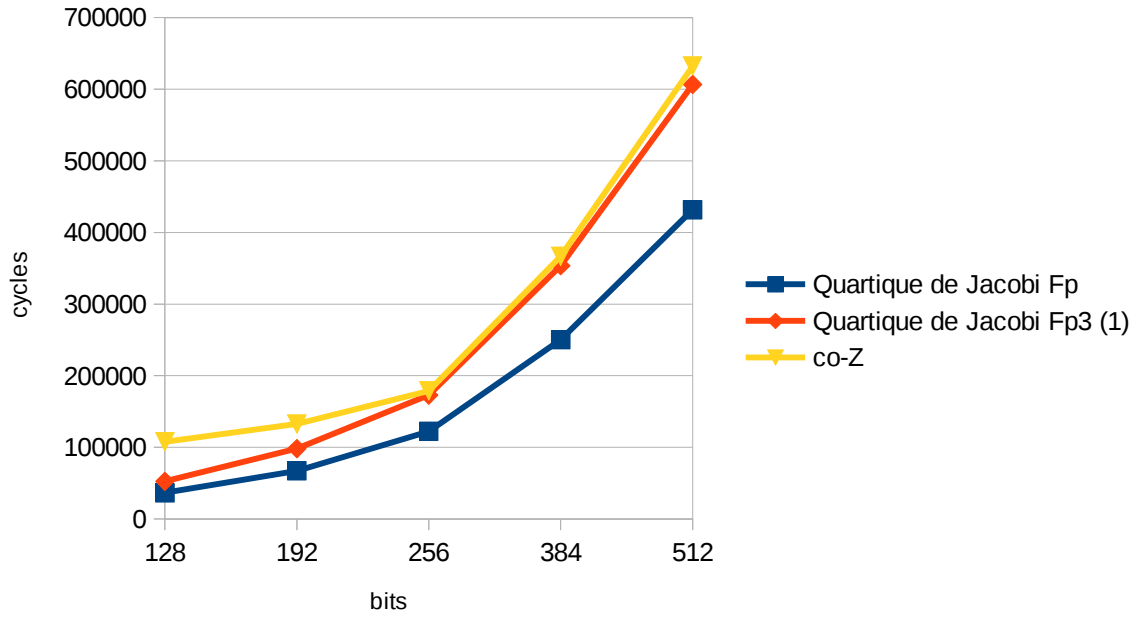


Figure. 6.14 – Comparaison en nombre de cycles des multiplications scalaires pour les quartiques de Jacobi et les co-Z

Des travaux sont en cours afin de valider l'intégralité de notre architecture afin qu'elle soit utilisée par un protocole ECDSA implanté en logiciel. Les travaux de thèse de Simon Pontié permettront également de vérifier la résistance de notre implantation par rapport aux attaques SPA.

Chapitre 7

Conclusion et perspectives

Dans cette partie, nous avons étudié les solutions qui permettent d'avoir une implantation des courbes elliptiques sur des cibles logicielle et matérielle qui soit à la fois performante et sûre. Nous avons en particulier analyser les formes permettant d'obtenir une arithmétique unifiée. Pour l'arithmétique modulaire, nous avons discuté des différentes possibilités d'implantation et de parallélisation possibles.

La génération de courbes elliptiques résistantes et non biaisées est un jeu important dans la sécurité des protocoles. Nous avons analysé la méthode de génération de Brainpool et adapté celle-ci aux courbes quartiques de Jacobi étendues et à celles d'Edwards. Nous avons implanté notre solution et vérifié que nous pouvions re-générer les courbes Brainpool. Notre implantation actuelle nous permet donc de générer des courbes sous forme Weierstrass, quartique de Jacobi étendue et Edwards en permettant à l'utilisateur de désactiver ces critère afin d'accélérer la méthode de génération et de paramétrer l'algorithme de génération aléatoire des paramètres. Nous fournissons également les données assurant la résistance des courbes générées.

Nous avons continué l'implantation de notre bibliothèque `MPHELL` en apportant les modifications qui font suite à notre analyse de l'arithmétique modulaire et de celle des courbes. Nous avons ainsi obtenu des performances comparables à celles de l'état de l'art. Notre implantation est modulable afin de permettre d'ajouter aisément de nouvelles fonctionnalités mais également de brancher d'autres bibliothèques. Il est prévu de finaliser `MPHELL` pour la première distribution à la fin du premier trimestre de cette année. De plus, des tests supplémentaires sont en cours afin d'obtenir des benchs avec d'autres bibliothèques.

Nous avons également implanté une arithmétique modulaire sur $\mathbb{F}_p$, $\mathbb{F}_{p^2}$ et $\mathbb{F}_{p^3}$ sur des cibles FPGA. Après l'étude de l'existant, nous avons décidé de concentrer nos efforts

sur la multiplication de Montgomery en implantant une architecture basée sur l'utilisation de DSPs. Cela nous a permis d'obtenir un module efficace. Nous avons également étudié la parallélisation de l'addition de point pour la forme quartique de Jacobi sur nos cibles. Cela nous a permis d'améliorer les performances par rapport à la version séquentielle. L'implantation réalisé pour $\mathbb{F}_{p^3}$ permet de concurrencer une implantation classique Weierstrass sur$\mathbb{F}_p$ et nous avons également constaté un gain par rapport à une version de l'addition co-Z. Nous avons ainsi le support de courbes ne pouvant pas se mettre directement sous la forme quartique de Jacobi, et ceci avec une arithmétique rapide. La finalisation du module est en cours afin que l'on puisse tester sa résistance par rapport aux attaques SPA. De cette manière, nous aurons un module fonctionnel qui pourra être utilisé par des protocoles logiciels.

Liste des Algorithmes

Liste des Figures

Liste des Tableaux

Bibliographie

Standards

[AES]	FIPS PUB 197. *Advanced Encryption Standard (AES)*. U.S.Department of Commerce/National Institute of Standards and Technology. 2001.	[93]
[FIPS-186-4]	FIPS PUB 186-4. *Digital Signature Standard (DSS)*. U.S. Department of Commerce, National Institute of Standards et Technology, juil. 2013.	[10, 40, 44, 54, 55, 71]
[IEEE-P1363.3]	IEEE P12363.3. *Standard for Identity-Based Cryptographic techniques using Pairings*. Mai 2013.	[10, 75]
[ISO-14888-3]	INFORMATION TECHNOLOGY - SECURITY TECHNIQUES. *Digital signatures with appendix – Part 3 : Discrete logarithm based mechanisms, ISO/IEC 14888-3*. International Organization for Standardization, 2006.	[10]
[ISO-18032]	INFORMATION TECHNOLOGY - SECURITY TECHNIQUES. *Prime Number Generation, ISO/IEC 18032*. International Organization for Standardization, 2005.	[54]
[LM10]	LOCHTER, M. et MERKLE, J. *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*. RFC 5639 (Informational). Internet Engineering Task Force, mar. 2010. URL : http://www.ietf.org/rfc/rfc5639.txt.	[44, 53]
[SEC1]	RESEARCH, C. *SEC 1 : Elliptic Curve Cryptography*. http://www.secg.org/sec1-v2.pdf. 2009.	[40]

[SEC2] RESEARCH, C. *SEC 2 : Recommended Elliptic Curve Domain Parameters*. http://www.secg.org/sec2-v2.pdf. Standards for Efficient Cryptography. 2010.

[SP800-57] BARKER, E. B. *SP 800-57. Recommendation for Key Management, Part 1 rev4*. Rapp. tech. Gaithersburg, MD, United States, 2016. [10, 127]

[SP800-90A] BARKER, E. B. et KELSEY, J. M. *SP 800-90A Rev.1. Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Rapp. tech. Gaithersburg, MD, United States, 2015. [93]

[X9.62] ANSI X9.62. *Public Key Cryptography For The Financial Services Industry : The Elliptic Curve Digital Signature Algorithm (ECDSA)*. American National Standards Institute, X9-Financial Services. 2005. [10, 44, 54, 55]

Bibliothèques logicielles

[GMP] GRANLUND, T. et THE GMP DEVELOPMENT TEAM. *GNU MP : The GNU Multiple Precision Arithmetic Library*. 6.1.0. http://gmplib.org/. 2015. [69, 92]

[MAGMA] BOSMA, W., CANNON, J. et PLAYOUST, C. « The Magma algebra system. I. The user language ». In : *J. Symbolic Comput.* 24.3-4 (1997). Computational algebra and number theory (London, 1993), p. 235–265. ISSN : 0747-7171. DOI : 10.1006/jsco.1996.0125. URL : http://dx.doi.org/10.1006/jsco.1996.0125. [92]

[PARI] PARI/GP. *version* 2.7.5. available from http://pari.math.u-bordeaux.fr/. The PARI Group. Bordeaux, 2015. [59, 92]

[SAGE] DEVELOPERS, T. S. *Sage Mathematics Software (Version x.y.z)*. http://www.sagemath.org. 2016. [92]

Articles et livres

[AKLGL11] ARANHA, D. F., KARABINA, K., LONGA, P., GEBOTYS, C. H. et LÓPEZ, J. « Faster Explicit Formulas for Computing Pairings over Ordinary Curves. » In : *EUROCRYPT*. Sous la dir. de PATERSON, K. G. T. 6632. Lecture Notes in Computer Science. Springer, 2011, p. 48–68. [75]

[Ava04] AVANZI, R. M. « Aspects of Hyperelliptic Curves over Large Prime Fields in Software Implementations. » In : *CHES*. Sous la dir. de JOYE, M. et QUISQUATER, J.-J. T. 3156. Lecture Notes in Computer Science. Springer, 2004, p. 148–162. [75]

[BBJLP08] BERNSTEIN, D. J., BIRKNER, P., JOYE, M., LANGE, T. et PETERS, C. « Twisted edwards curves ». In : *Progress in Cryptology–AFRICACRYPT 2008*. Springer, 2008, p. 389–405. [38–40]

[BDFGLR16] BAIGNÈRES, T., DELERABLÉE, C., FINIASZ, M., GOUBIN, L., LEPOINT, T. et RIVAIN, M. « Trap Me If You Can - Million Dollar Curve. » In : *IACR Cryptology ePrint Archive* (2016). URL : http://cryptoexperts.github.io/million-dollar-curve. [44, 45]

[BDK01] BAJARD, J.-C., DIDIER, L.-S. et KORNERUP, P. « Modular Multiplication and Base Extensions in Residue Number Systems. » In : *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 2001, p. 59–65. [64]

[Ber+15] BERNSTEIN, D. J., CHOU, T., CHUENGSATIANSUP, C., HÜLSING, A., LAMBOOIJ, E., LANGE, T., NIEDERHAGEN, R. et VREDENDAAL, C. van. « How to Manipulate Curve Standards : A White Paper for the Black Hat ». In : *SSR*. Sous la dir. de CHEN, L. et MATSUO, S. T. 9497. Lecture Notes in Computer Science. Springer, 2015, p. 109–139. [57]

[BGV94] BOSSELAERS, A., GOVAERTS, R. et VANDEWALLE, J. « Comparison of three modular reduction functions ». In : *Advances in Cryptology—CRYPTO'93*. Springer. 1994, p. 175–186. [71]

[BJ03a] BILLET, O. et JOYE, M. « The Jacobi model of an elliptic curve and side-channel analysis ». In : *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. Springer, 2003, p. 34–42. [30, 31, 35, 36]

[BJ03b] BRIER, E. et JOYE, M. « Fast Point Multiplication on Elliptic Curves through Isogenies. » In : *AAECC*. Sous la dir. de FOSSORIER, M. P. C., HØHOLDT, T. et POLI, A. T. 2643. Lecture Notes in Computer Science. Springer, 2003, p. 43–50. [54]

[BL] BERNSTEIN, D. et LANGE, T. *Explicit-formulas database*. URL : http://www.hyperelliptic.org/EFD. [23, 33, 36]

[BL07a] BERNSTEIN, D. J. et LANGE, T. « Faster addition and doubling on elliptic curves ». In : *Advances in cryptology–ASIACRYPT 2007*. Springer, 2007, p. 29–50. [37, 40]

[BL07b] BERNSTEIN, D. J. et LANGE, T. « Inverted edwards coordinates ». In : *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*. Springer, 2007, p. 20–27. [38, 40]

[BLS03] BARRETO, P. S., LYNN, B. et SCOTT, M. « Constructing elliptic curves with prescribed embedding degrees ». In : *Security in Communication Networks*. Springer, 2003, p. 257–267. [44]

[BP01] BLUM, T. et PAAR, C. « High-Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. » In : *IEEE Trans. Computers* 50.7 (2001), p. 759–764. [102]

[BS04] BAKTIR, S. et SUNAR, B. « Optimal tower fields ». In : *Computers, IEEE Transactions on* 53.10 (2004), p. 1231–1243. [74]

[BS10] BENGER, N. et SCOTT, M. « Constructing Tower Extensions of Finite Fields for Implementation of Pairing-Based Cryptography ». In : *Arithmetic of Finite Fields*. Sous la dir. d'HASAN, M. et HELLESETH, T. T. 6087. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, p. 180–195. [76, 77]

[CC86] CHUDNOVSKY, D. V. et CHUDNOVSKY, G. V. « Sequences of numbers generated by addition in formal groups and new primality and factorization tests ». In : *Advances in Applied Mathematics* 7.4 (1986-87), p. 385–434. [25]

[CF96] CASSELS, J. W. S. et FLYNN, E. V. *Prolegomena to a middlebrow arithmetic of curves of genus* 2. T. 230. Cambridge University Press, 1996. [29]

[CMO98]	COHEN, H., MIYAJI, A. et ONO, T. « Efficient elliptic curve exponentiation using mixed coordinates ». In : *Advances in Cryptology—ASIACRYPT'98*. Springer. 1998, p. 51–65.	[25]
[Coh+12]	COHEN, H., FREY, G., AVANZI, R., DOCHE, C., LANGE, T., NGUYEN, K. et VERCAUTEREN, F. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition*. 2nd. Chapman & Hall/CRC, 2012.	[16, 19, 20, 66, 71, 72]
[Cor99]	CORON, J.-S. « Resistance against differential power analysis for elliptic curve cryptosystems ». In : *Cryptographic Hardware and Embedded Systems*. Springer. 1999, p. 292–302.	[27]
[Cox89]	COX, D. A. « Primes of the Form $x^2 + ny^2$ ». In : *Wiley-Intersci. Publ. John Wiley & Sons Inc., New York* (1989).	[22]
[CP01]	CRANDALL, R. et POMERANCE, C. « Prime Numbers : A Computational Perspective ». In : (2001).	[71]
[Deu41]	DEURING, M. « Die typen der multiplikatorenringe elliptischer funktionenkörper ». In : *Abhandlungen aus dem mathematischen Seminar der Universität Hamburg*. T. 14. 1. Springer. 1941, p. 197–272.	[23, 51, 52]
[Dhe98]	DHEM, J.-F. « Design of an efficient public-key cryptographic library for RISC-based smart cards ». Université catholique de Louvain, mai 1998.	[71]
[DOSD06]	DEVEGILI, A. J., O'EIGEARTAIGH, C., SCOTT, M. et DAHAB, R. « Multiplication and Squaring on Pairing-Friendly Fields. » In : *IACR Cryptology ePrint Archive* 2006 (2006), p. 471.	[78, 145]
[Duq07]	DUQUESNE, S. « Improving the arithmetic of elliptic curves in the Jacobi model ». In : *Information Processing Letters* 104.3 (2007), p. 101–105.	[33, 35, 36]
[Edw07]	EDWARDS, H. « A normal form for elliptic curves ». In : *Bulletin of the American Mathematical Society* 44.3 (2007), p. 393–422.	[37]
[EW93]	ELDRIDGE, S. E. et WALTER, C. D. « Hardware Implementation of Montgomery's Modular Multiplication Algorithm. » In : *IEEE Trans. Computers* 42.6 (1993), p. 693–699.	[105]

[Eyn15] EYNARD, J. « Approche arithmétique RNS de la cryptographie asymétrique ». Université Pierre et Marie Curie, mai 2015. [64]

[fai10] FAILOVERFLOW. « Console hacking ». In : 27th Chaos Communication Congress, 2010. URL : http://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html. [12]

[Fog16] FOG, A. *Instruction tables : Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. 2016. URL : http://www.agner.org/optimize/instruction_tables.pdf. [64]

[FPRE15] FLORI, J.-P., PLÛT, J., REINHARD, J.-R. et EKERÅ, M. « Diversity and Transparency for ECC. » In : *IACR Cryptology ePrint Archive* 2015 (2015), p. 659. [44]

[FR94] FREY, G. et RÜCK, H.-G. « A remark concerning m-divisibility and the discrete logarithm in the divisor class group of curves ». In : *Mathematics of computation* 62.206 (1994), p. 865–874. [50]

[FST10] FREEMAN, D., SCOTT, M. et TESKE, E. « A Taxonomy of Pairing-Friendly Elliptic Curves. » In : *J. Cryptology* 23.2 (2010), p. 224–280. [74]

[GACS09] GHOSH, S., ALAM, M., CHOWDHURY, D. R. et SENGUPTA, I. « Parallel crypto-devices for GF(p) elliptic curve multiplication resistant against side channel attacks. » In : 35.2 (2009), p. 329–338. [102]

[Gal99] GALBRAITH, S. D. « Constructing isogenies between elliptic curves over finite fields ». In : *LMS Journal of Computation and Mathematics* 2 (1999), p. 118–138. [52]

[Gau03] GAUDRY, P. « Some remarks on the elliptic curve discrete logarithm ». 2003. [52]

[GJM10] GOUNDAR, R. R., JOYE, M. et MIYAJI, A. « Co-Z addition formulæ and binary ladders on elliptic curves ». In : *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer, 2010, p. 65–79. [27]

[Gou03] GOUBIN, L. « A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. » In : *Public Key Cryptography*. Sous la dir. de DESMEDT, Y. T. 2567. Lecture Notes in Computer Science. Springer, 2003, p. 199–210. [54]

[GP08] GÜNEYSU, T. et PAAR, C. « Ultra High Performance ECC over NIST Primes on Commercial FPGAs. » In : *CHES*. Sous la dir. d'OSWALD, E. et ROHATGI, P. T. 5154. Lecture Notes in Computer Science. Springer, 2008, p. 62–78. [102]

[Has36] HASSE, H. « Zur Theorie der abstrakten elliptischen Funktionenkörper I, II & III. » In : *Journal für die reine und angewandte Mathematik* 175 (1936). [21]

[HCD07] HISIL, H., CARTER, G. et DAWSON, E. « New Formulae for Efficient Elliptic Curve Arithmetic ». In : *Progress in Cryptology – INDOCRYPT 2007*. Sous la dir. de SRINATHAN, K., RANGAN, C. et YUNG, M. T. 4859. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, p. 138–151. [34]

[Hen08] HENSEL, K. *Theorie der algebraischen Zahlen*. T. 1. BG Teubner, 1908. [71]

[HQ00] HACHEZ, G. et QUISQUATER, J.-J. « Montgomery Exponentiation with no Final Subtractions : Improved Results. » In : *CHES*. Sous la dir. de KOÇ, c. K. et PAAR, C. T. 1965. Lecture Notes in Computer Science. Springer, 2000, p. 293–301. [106]

[HWCD08] HISIL, H., WONG, K. K.-H., CARTER, G. et DAWSON, E. « Twisted Edwards curves revisited ». In : *Advances in Cryptology-ASIACRYPT 2008*. Springer, 2008, p. 326–343. [39, 40]

[HWCD09a] HISIL, H., WONG, K., CARTER, G. et DAWSON, E. « Jacobi Quartic Curves Revisited ». In : *Information Security and Privacy*. Springer. 2009, p. 452–468. [34–36]

[HWCD09b] HISIL, H., WONG, K. K.-H., CARTER, G. et DAWSON, E. « Faster group operations on elliptic curves ». In : *Proceedings of the Seventh Australasian Conference on Information Security-Volume 98*. Australian Computer Society, Inc. 2009, p. 7–20. [34, 36]

[Hıs10] HISIL, H. « Elliptic curves, group law, and efficient computation ». phd. Queensland University of Technology, 2010. [40]

[JMV09] JAO, D., MILLER, S. D. et VENKATESAN, R. « Expander graphs based on GRH with an application to elliptic curve cryptography ». In : *Journal of Number Theory* 129.6 (2009), p. 1491–1504. [52]

[Joy07] JOYE, M. « Highly Regular Right-to-Left Algorithms for Scalar Multiplication. » In : *CHES*. Sous la dir. de PAILLIER, P. et VERBAUWHEDE, I. T. 4727. Lecture Notes in Computer Science. Springer, 2007, p. 135–147. [27]

[JQ01a] JOYE, M. et QUISQUATER, J.-J. « Hessian elliptic curves and side-channel attacks ». In : *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer. 2001, p. 402–410. [27]

[JQ01b] JOYE, M. et QUISQUATER, J.-J. « Hessian Elliptic Curves and Side-Channel Attacks. » In : *CHES*. Sous la dir. de KOÇ, c. K., NACCACHE, D. et PAAR, C. T. 2162. Lecture Notes in Computer Science. Springer, 2001, p. 402–410. [40]

[JRWM15] JUNOD, P., RINALDINI, J., WEHRLI, J. et MICHIELIN, J. « Obfuscator-LLVM – Software Protection for the Masses ». In : *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*. Sous la dir. de WYSEUR, B. IEEE, 2015, p. 3–9. DOI : 10.1109/SPRO.2015.10. [226]

[Kar95] KARATSUBA, A. A. « The complexity of computations ». In : *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation* 211 (1995), p. 169–183. [69]

[KM05] KOBLITZ, N. et MENEZES, A. « Pairing-Based Cryptography at High Security Levels ». In : *Cryptography and Coding : 10th IMA International Conference, Cirencester, UK, December 19-21, 2005, Proceedings*. T. 3796. Springer. 2005, p. 13. [75, 77]

[Knu97] KNUTH, D. E. *The Art of Computer Programming, Volume 2 (3rd Ed.) : Seminumerical Algorithms*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN : 0-201-89684-2. [68]

[Kob87] KOBLITZ, N. « Elliptic curve cryptosystems ». In : *Mathematics of computation* 48.177 (1987), p. 203–209. [9]

[Koh96] KOHEL, D. « Endomorphism rings of elliptic curves over finite fields ». Thèse de doct. University of California at Berkeley, 1996. [22]

[LH00] LIM, C. H. et HWANG, H. S. « Fast Implementation of Elliptic Curve Arithmetic in GF(p^n). » In : *Public Key Cryptography*. Sous la dir. d'IMAI, H. et ZHENG, Y. T. 1751. Lecture Notes in Computer Science. Springer, 2000, p. 405–421. [75]

[LN97] LIDL, R. et NIEDERREITER, H. *Finite fields*. T. 20. Encyclopaedia of mathematics and its applications. New York : Cambridge University Press, 1997. [75]

[LS01] LIARDET, P.-Y. et SMART, N. P. « Preventing SPA/DPA in ECC systems using the Jacobi form ». In : *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer. 2001, p. 391–401. [27, 29]

[LW15] LENSTRA, A. K. et WESOLOWSKI, B. « A random zoo : sloth, unicorn, and trx. » In : *IACR Cryptology ePrint Archive* 2015 (2015), p. 366. [45]

[Mel07] MELONI, N. « New point addition formulae for ECC applications ». In : *Arithmetic of Finite Fields*. Springer, 2007, p. 189–201. [26]

[Mil86] MILLER, V. « Use of elliptic curves in cryptography ». In : *Advances in Cryptology-CRYPTO'85 Proceedings*. Springer. 1986, p. 417–426. [9]

[MLPJ13] MA, Y., LIU, Z., PAN, W. et JING, J. « A High-Speed Elliptic Curve Cryptographic Processor for Generic Curves over GF(p). » In : *Selected Areas in Cryptography*. Sous la dir. de LANGE, T., LAUTER, K. E. et LISONEK, P. T. 8282. Lecture Notes in Computer Science. Springer, 2013, p. 421–437. [102]

[Mon85] MONTGOMERY, P. L. « Modular multiplication without trial division ». In : *Mathematics of computation* 44.170 (1985), p. 519–521. [71]

[Mon87] MONTGOMERY, P. L. « Speeding the Pollard and elliptic curve methods of factorization ». In : *Mathematics of computation* 48.177 (1987), p. 243–264. [27]

[MOV+93] MENEZES, A. J., OKAMOTO, T., VANSTONE, S. et al. « Reducing elliptic curve logarithms to logarithms in a finite field ». In : *Information Theory, IEEE Transactions on* 39.5 (1993), p. 1639–1646. [49, 50]

[OP01] ORLANDO, G. et PAAR, C. « A Scalable GF(p) Elliptic Curve Processor Architecture for Programmable Hardware. » In : *CHES*. Sous la dir. de KOÇ, c. K., NACCACHE, D. et PAAR, C. T. 2162. Lecture Notes in Computer Science Generators. Springer, 2001, p. 348–363. [102]

[Oru95] ORUP, H. « Simplifying Quotient Determination in High-Radix Modular Multiplication. » In : *IEEE Symposium on Computer Arithmetic*. IEEE Computer Society, 1995, p. 193–199. [105]

[Plu11] PLUT, J. « On Various Families of Twisted Jacobi Quartics. » In : *Selected Areas in Cryptography*. Sous la dir. de MIRI, A. et VAUDENAY, S. T. 7118. Lecture Notes in Computer Science. Springer, 2011, p. 373–383. [36, 127]

[Pol78] POLLARD, J. M. « Monte Carlo methods for index computation (mod p) ». In : *Mathematics of computation* 32.143 (1978), p. 918–924. [46]

[RCB15] RENES, J., COSTELLO, C. et BATINA, L. « Complete addition formulas for prime order elliptic curves. » In : *IACR Cryptology ePrint Archive* 2015 (2015), p. 1060. [28]

[RSA78] RIVEST, R. L., SHAMIR, A. et ADLEMAN, L. « A method for obtaining digital signatures and public-key cryptosystems ». In : *Commun. ACM* 21.2 (1978), p. 120–126. [10]

[SA98] SATOH, T et ARAKI, K. « Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves ». In : *Commentarii Math. Univ. St. Pauli* 47 (1998), p. 81–92. [51]

[Sco07] SCOTT, M. « Implementing cryptographic pairings ». In : *Lecture Notes in Computer Science* 4575 (2007), p. 177. [75]

[Sem98] SEMAEV, I. « Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p ». In : *Mathematics of Computation of the American Mathematical Society* 67.221 (1998), p. 353–356. [51]

[Sha71] SHANKS, D. « Class number, a theory of factorization, and genera ». In : *1969 Number Theory Institute (Proc. Sympos. Pure Math., Vol. XX, State Univ. New York, Stony Brook, N.Y., 1969)*. Providence, R.I., 1971, p. 415–440. [46]

[SIN11] SONG, B., ITO, Y. et NAKANO, K. « CRT-Based DSP Decryption Using Montgomery Modular Multiplication on the FPGA. » In : *IPDPS Workshops*. IEEE, 2011, p. 532–541. [102]

[SM11] SUZUKI, D. et MATSUMOTO, T. « How to Maximize the Potential of FPGA-Based DSPs for Modular Exponentiation. » In : *IEICE Transactions* 94-A.1 (2011), p. 211–222. [102, 107]

[Sma99] SMART, N. P. « The discrete logarithm problem on elliptic curves of trace one ». In : *Journal of cryptology* 12.3 (1999), p. 193–196. [51]

[SMKLM01] SUNG-MING, Y., KIM, S., LIM, S. et MOON, S. « A countermeasure against one physical cryptanalysis may benefit another attack ». In : *Information Security and Cryptology—ICISC 2001*. Springer, 2001, p. 414–427. [27]

[Tat66] TATE, J. « Endomorphisms of abelian varieties over finite fields ». In : *Inventiones mathematicae* 2.2 (1966), p. 134–144. [52]

[TK99] TENCA, A. F. et KOÇ, c. K. « A Scalable Architecture for Montgomery Multiplication. » In : *CHES*. Sous la dir. de KOÇ, c. K. et PAAR, C. T. 1717. Lecture Notes in Computer Science. Springer, 1999, p. 94–108. [107]

[TTK01] TENCA, A. F., TODOROV, G. et KOÇ, c. K. « High-Radix Design of a Scalable Modular Multiplier. » In : *CHES*. Sous la dir. de KOÇ, c. K., NACCACHE, D. et PAAR, C. T. 2162. Lecture Notes in Computer Science Generators. Springer, 2001, p. 185–201.

[TTL03] TANG, S. H., TSUI, K. S. et LEONG, P. H. W. « Modular exponentiation using parallel multipliers. » In : *FPT*. IEEE, 2003, p. 52–59. [102]

[Van92] VANSTONE, S. « Responses to NIST's Proposal. » In : *Communications of the ACM* 35 (juil. 1992), p. 50–52. [10]

[Was08] WASHINGTON, L. C. *Elliptic Curves : Number Theory and Cryptography, Second Edition*. 2^{e} éd. Chapman & Hall/CRC, 2008. [46, 49, 50]

[Wei49] WEIL, A. « Numbers of solutions of equations in finite fields ». In : *Bull. Amer. Math. Soc* 55.5 (1949), p. 497–508. [21]

[Xil12] XILINX. *Virtex-5 FPGA XtremeDSP Design Considerations Slice User Guide*. Jan. 2012. URL : http://www.xilinx.com/support/documentation/user_guides/ug193.pdf. [103, 104]

[Xil14a] XILINX. *7 Series DSP48E1 Slice User Guide*. Nov. 2014. URL : http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf. [103]

[Xil14b] XILINX. *Virtex-5 FPGA Data Sheet : DC and Switching Characteristics*. Déc. 2014. URL : http://www.xilinx.com/support/documentation/data_sheets/ds202.pdf. [104]

[Xil15a] XILINX. *7 Series FPGAs Overview*. Mai 2015. URL : http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf. [103]

[Xil15b] XILINX. *Virtex-5 Family Overview*. Août 2015. URL : http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf. [103]

[Xil16] XILINX. *Virtex-7 T and XT FPGAs Data Sheet : DC and AC Switching Characteristics*. Fév. 2016. URL : http://www.xilinx.com/support/documentation/data_sheets/ds183_Virtex_7_Data_Sheet.pdf. [104]

[YJ00] YEN, S.-M. et JOYE, M. « Checking before output may not be enough against fault-based cryptanalysis ». In : *IEEE Transactions on Computers* 49.9 (2000), p. 967–970. [27]

Annexe A

Tables des opérations pour addition et doublement

A.1 Weierstrass coordonnées projectives

A.1.1 Addition

$$
\begin{aligned}
Y_1Z_2 &= Y_1 * Z_2 \\
X_1Z_2 &= X_1 * Z_2 \\
Z_1Z_2 &= Z_1 * Z_2 \\
u &= Y_2 * Z_1 - Y_1Z_2 \\
uu &= u^2 \\
v &= X_2 * Z_1 - X_1Z_2 \\
vv &= v^2 \\
vvv &= v * vv \\
R &= vv * X_1Z_2 \\
A &= uu * Z_1Z_2 - vvv - 2 * R \\
X_3 &= v * A \\
Y_3 &= u * (R - A) - vvv * Y_1Z_2 \\
Z_3 &= vvv * Z_1Z_2
\end{aligned}
$$

A.1.2 Doublement

$$
\begin{aligned}
XX &= X1^2 \\
ZZ &= Z1^2 \\
w &= a * ZZ + 3 * XX \\
s &= 2 * Y_1 * Z_1 \\
ss &= s^2 \\
sss &= s * ss \\
R &= Y_1 * s \\
RR &= R^2 \\
B &= (X_1 + R)2 - XX - RR \\
h &= w^2 - 2 * B \\
X_3 &= h * s \\
Y_3 &= w * (B - h) - 2 * RR \\
Z_3 &= sss
\end{aligned}
$$

A.2 Weierstrass coordonnées projectives, a = -3

A.2.1 Doublement

$$
\begin{aligned}
w &= 3 * (X_1 - Z_1) * (X_1 + Z_1) \\
s &= 2 * Y_1 * Z_1 \\
ss &= s^2 \\
sss &= s * ss \\
R &= Y_1 * s \\
RR &= R^2 \\
B &= 2 * X_1 * R \\
h &= w^2 - 2 * B \\
X_3 &= h * s \\
Y_3 &= w * (B - h) - 2 * RR \\
Z_3 &= sss
\end{aligned}
$$

A.3 Weierstrass coordonnées jacobiennes

A.3.1 Addition

$$
\begin{aligned}
Z_1Z_1 &= Z_1^2 \\
Z_2Z_2 &= Z_2^2 \\
U_1 &= X_1 * Z_2Z_2 \\
U_2 &= X_2 * Z_1Z_1 \\
S_1 &= Y_1 * Z_2 * Z_2Z_2 \\
S_2 &= Y_2 * Z_1 * Z_1Z_1 \\
H &= U_2 - U_1 \\
I &= (2 * H)2 \\
J &= H * I \\
r &= 2 * (S_2 - S_1) \\
V &= U_1 * I \\
X_3 &= r2 - J - 2 * V \\
Y_3 &= r * (V - X_3) - 2 * S_1 * J \\
Z_3 &= ((Z_1 + Z_2)^2 - Z_1Z_1 - Z_2Z_2) * H
\end{aligned}
$$

A.3.2 Doublement

$$
\begin{aligned}
XX &= X_1^2 \\
YY &= Y_1^2 \\
YYYY &= YY^2 \\
ZZ &= Z_1^2 \\
S &= 2 * ((X_1 + YY)2 - XX - YYYY) \\
M &= 3 * XX + a * ZZ^2 \\
T &= M^2 - 2 * S \\
X_3 &= T \\
Y_3 &= M * (S - T) - 8 * YYYY \\
Z_3 &= (Y_1 + Z_1)2 - YY - ZZ
\end{aligned}
$$

A.4 Weierstrass coordonnées jacobiennes, a = -3

A.4.1 Doublement

$$
\begin{aligned}
ZZ &= Z_1^2 \\
YY &= Y_1^2 \\
U_1 &= X_1 * YY \\
U_2 &= 3 * (X_1 - ZZ) * (X_1 + ZZ) \\
X_3 &= U_2^2 - 8 * U_1 \\
Z_3 &= (Y_1 + Z_1)^2 - YY - ZZ \\
Y_3 &= U_2 * (4 * U_1 - X_3) - 8 * YY^2
\end{aligned}
$$

Annexe B

Parallélisation de la multiplication sur $\mathbb{F}_{p^6}$

Dans la suite, nous notons :

- N : Multiplication naïve,
- K : Multiplication Karatsuba,
- i : Architecture parallélisée i.

Dans les tableaux de coûts, la symbolique des architectures est modifiée. Par exemple, pour le tableau B.1, l'architecture parallélisée 2 pour $\mathbb{F}_{p^3}$ signifie qu'il faut utiliser six multiplieurs et 3 additionneurs sur $\mathbb{F}_{p^2}$.

$\mathbb{F}_{p^2}$ \ $\mathbb{F}_{p^3}$	N	N 1	N 2	N 3
N	36M + 30A	12M + 8A	8M + 6A	4M + 6A
N 1	18M + 15A	6M + 4A	4M + 4A	2M + 2A
N 2-3	9M + 15A	3M + 4A	2M + 3A	1M + 2A
K	27M + 57A	9M + 17A	6M + 9A	3M + 9A
K 1-2-3	9M + 24A	3M + 7A	2M + 5A	1M + 3A

Tableau B.1 – Coûts d'une multiplication sur $\mathbb{F}_{p^6}$ (6 = 2x3)(1)

On peut remarquer que pour les combinaisons d'architectures non parallélisées, nous retrouvons les coûts donnés dans [DOSD06].

$\mathbb{F}_{p^2}$ \ $\mathbb{F}_{p^3}$	K	K 1	K 2	K 3
N	24M + 38A	8M + 8A	4M + 10A	4M + 8A
N 1	12M + 19A	4M + 4A	2M + 5A	2M + 4A
N 2-3	6M + 19A	2M + 4A	M + 5A	M + 4A
K	18M + 56A	6M + 14A	3M + 13A	3M + 11A
K 1-2-3	6M + 25A	2M + 6A	M + 6A	M + 5A

Tableau B.2 – Coûts d'une multiplication sur $\mathbb{F}_{p^6}$ (6 = 2x3)(2)

$\mathbb{F}_{p^3}$ \ $\mathbb{F}_{p^2}$	N	N 1	N 2-3	K	K 1-2-3
N	36M + 30A	18M + 15A	9M + 9A	27M + 33A	9M + 12A
N 1	16M + 6A	6M + 3A	3M + 2A	9M + 8A	3M + 3A
N 2	8M + 6A	4M + 3A	2M + 2A	6M + 8A	2M + 3A
N 3	4M + 10A	2M + 5A	1M + 3A	3M + 11A	1M + 4A
K	24M + 58A	12M + 29A	6M + 16A	18M + 54A	6M + 19A
K 1	8M + 10A	4M + 5A	2M + 3A	6M + 11A	2M + 4A
K 2	4M + 6A	3M + 9A	1M + 5A	3M + 17A	1M + 6A
K 3	4M + 14A	2M + 6A	1M + 4A	3M + 14A	1M + 5A

Tableau B.3 – Coûts d'une multiplication sur $\mathbb{F}_{p^6}$ (6 = 3x2)

Annexe C

Exemple d'une courbe quartique de Jacobi générée

```
#Curve Data
Curve-ID: jq256_3
p: b09f075797da89f57ec8c0265b014c5ba37bcb84208a4c4d83199a31d17cb1af
a: 1468727718e87fae1d94287a36205fa0e3b70bbaad31a097c1ffa5a5db263683
x(P_0): 7e99fa90a889802939e8704d3cc8abc5c0c879540dea85075eabfb6c73d515c5
y(P_0): 5d5db0901ca66a952304e220fcf6581c55fb7fb7986522096911c09d1e76de9b
z(P_0): 86eccefd9aa9174b1e61505ba174452525f7fd2989304574b298978e79690c20
t(P_0): 77d5d37af2ee21e362a68ec5e277b40538434a020c6d1a3c885b6eb8fec53dd3
q: 2c27c1d5e5f6a27d5fb2300996c0531746f66dec9a2e0bd13eabeafa863611f7
i: 4

#Weierstrass curve
aw: ab649337eb7e4560dab6c2f1cded3c736c3870a50497d4197053e5af6d68aac6
bw: 15701c0682740ee26df431edd6ebb038cc6dd7b50f846f1e29bfc65b5c65f52c
x(Pw_0): 9b9f3ee36651f00c43397ba5802971106770a73ab71d1f96ae6a2d71a9176d37
y(Pw_0): 64d0c6dd87c1876dd09c7409750ece534a6aba4e14ed17f84aa834ca40283fd2

#Twisted curve
z: 4bcac8dc7ee59948a3dc47a81b5115b4228fcd0311f3b22ccebfa36dcfed1e8e
a': b09f075797da89f57ec8c0265b014c5ba37bcb84208a4c4d83199a31d17cb1ac
b': 8e4704de8dda501715d7b34bdcc01e09814eed0ad5eaf6f38034c864844d451c
x(P_0'): 717629de16624d771be59700c042d7b04e4ee526978372043f01f25ffc9bf4b9
y(P_0'): 625a682ad2872c149201e04d14aea32166f01e30d26e4335d2934747258cf2f7
```

```
#Data for curve validation
Curve-ID: jq512_3

Seeds used to generate p
seed0 : 7be5466cf34e90c6cc0ac29b7c97c50dd3f84d5b
seed_p : 7be5466cf34e90c6cc0ac29b7c97c50dd3f84d5c
Seeds used to generate a and b
seed1 : 5f4bf8d8d8c31d763da06c80abb1185eb4f7c7b5
seed_a : 5f4bf8d8d8c31d763da06c80abb1185eb4f7d2be

u: -fffffffffffffffe87a213d1b7d21d088869ee47b8a469d4
v: 0
d: 264a05483df76395612f670cb21af6b836a0c3e2ddd2bd4f15d630ddb0a4cbcb

#Factorisation of d
Number of different prime factors of d: 4
Factord_1 : 7
Factord_2 : 1eb
Factord_3 : 113099
Factord_4 : 2a78df793d8c1c95a85a460e308ea97d1059ea7db694a50876b13f2f
-d mod 4: 1

#Weil-Tate-bound
#Factorisation of q - 1

Number of different prime factors of q-1: 4
Factord_1 : 2
Exponent_1 : 1

Factord_2 : 3
Exponent_2 : 1

Factord_3 : 1bb
Exponent_3 : 1

Factord_4 : 440b2ef0a0355f55862d5d1899b7b001f7da22f0f0aca9fffdf3ee3ed55eb
Exponent_4 : 1

(q-1)/(order of p mod q): 1

#Class number bound
```

```
#quadratic form :
qf_a: d
qf_b: 1
qf_c: bc801a013110af2e2d3822b4a7e734c5f93ed80e1cab0671a694a1cd3e3eff
MinClass: 10000000

#basepoint-Construction
x(P): 1
y(P): 557a4b4307eeea92ef8a918871e1261ea69a26574e00ec3a18fe82d05c17e627
k: 2d31a097c1ffa5a5db263683b2dd03a9c2c3920a2631a4d33e1c1b0570541a83
Seed used to generate k
seed_k : 5f4bf8d8d8c31d763da06c80abb1185eb4f7d2bf
```

Deuxième partie

Obscurcissement de code source et mesure de complexité

Table des matières

Chapitre 8

Introduction

Dans l'introduction générale, nous avons vu que l'obscurcissement était une technique de protection logicielle qui a pour but de complexifier un programme tout en préservant ces fonctionnalités. Elle fait également l'objet d'un concours annuel IOCCC [1] qui récompense les meilleurs codes obscurcis. Voici par exemple l'un des gagnants de 2014.

```
                                                    #/*
                                                    q{G.
                                    */include<stdio.h>        /*G!*/
                                     #define G(g,o)/**/int    g=o;/**/
                                          #define q(b)int o[]={0,0,0,0,\
                                        0};char*p=#b,P[9999],*d=P,*e[]={"('A')"\
                                      ,"(A')","('A)","('A')"},b/*3792/e887094c*/
                                   q(*s=__TIME__;extern void w(){G(g,*o)for(;g--
                                ;);}extern void x(){}extern void y(){}static void z
                             (char*p){for(;(*d=*p++)!=0;d++)*d-=061*(59==*d||*d==81);}int
                               main(){char N[]="<;iQ=/*do<;dQ=ook8?boc;aQ=odr8>oc;_Q<ox8>oc"
                          ";^Q<d|8=o;]Q<dj8<Pk8<c]'?;QQdi8?P'Q)d8<9'8?c;Qdh8=P']Q<'c8>PQ9_8"
                            ">b;og8=''Q<'b8?'QQ9_8?c;dPe8dQa8<P]Q<''8=;'f8>QQ']8?ocQQ_8='_Q'8aQ"
                        "CYOUQWIN!]Q=;of8BQQ)88bc]QA98P'Qo]8>oQQ_8CP;QQ98b'_8@P8(Q']'^Q='']QB)88"
                      "bQQ]'8J;QQ'88bc99bd('8iQ@''QQo^8E''8c;QQ'Q''8?QQ''kQ<o]8B9PQQ'';_Q<'_8<cbQ"
                   "<o_Q<o_QBdo'QQ';'Q=9']8=bobQ_o_Q<o]8=';bQC(88cQ'bcgQ>oP']8<;cQ>98)^Q?''oc'Q>"
                  "oc']Q]8<;cQ>'8bbQA''QQ''Q?88P;dQ=')nQ?*/;;"/*}}=~/./;print"cp\40sinon.c\40r",
                        "un.c\n";for($i=1;$i<21;$i++){for($j=0;$j<2;$j++){print"gcc\x20-O",$j*2,
                        "\40run.c\x20-o run&&./run|tee run.c\n"if!($i%(2+$j));}print"sleep $j\n";
                      } __DATA__ 8M*/;G(Z,60*(60*(10**s+s[1])+s[3]*10+s[4])+s[6]*10+s[7]-1933008
                    )G(1,(Z+86400-*o)%86400)G(M,o[3]  *17+o[4]*11)   G(0,(&x-&w)-(&y-&x)?1:0)G(
                   m,*o)G(n,m)G(i,0)G(j,O)if(M>197)for(   p=N;*p;     )if(*p>92){*N=p[1  ];N[1]
                =0;for(  O=*p++-90;0<O--;z(  N));p++;}else{   s=       p;for(O=  *p++-59;O--;p++)
              p[-1]=*p;p[-1]=0;z(s);}else{    if(M==0){*o=Z;l=0;         }if(o[O+3]&&o[  O+1]+(O?3:5
              )>1)Z=O;else++o[O+3];o[O+1]     =1;  if(1<60)d+=            sprintf(d,"#include<stdio.h>"
"\12#define G(g,o)  int  g=o;\n#d"       "efine  \161(b"          ")int\40o[]={%d,%d,%d,%d,%d};char"
      "*p=#b,P[9999],*d=P,*e[]="         "{\"%s\", \""        "%s"    "\",\"%s\",\"%s\"},   b\nq(%s)\n"
           ,*o,o[01],o[2],o[3],  o[4]    ,e[3],e[1],      e[2],*e,p);z("/*;; ");for(i=0;          i++<18
              ;z(";"))for(j=0;j++<11;n=(n*193+287)      %384 ){m=(97+(67*m    ))%198;for
            (O=0;O++<n%03;z("Q")   );s=d;for(z(e[         (  n/3)&3]);m<M      &&s-d;*s++
          =040);}if(l<60)  {for   (z(";Time:Q")              ;(l++<60);z       ("#"));for
         (z(";Heca"/*  */   "t"   "eQII:" "Q")               ;o[3  ]++      <=24;z("#"));z
        (Z?"":"Q(j"     "a"  "m"   "me"   /*                    @     */"d"   ")");for(z(
       ";GlockQ18"        "C:" /*          *                          */         "Q");o[4]++<
      25;z("#")             );z(               Z-                     1?            ";":"Q(jam"
     "med);")                  /*              */                      ;}               else z(";"
    ";Gam"                      /*                                    */                 "eQover."
   ";;"                          /*                                  (-                      */);for
                                  (O=                                0;O                       <77;O
                                    ++)        z("Q");z(         "*"                            ""
                                       "/"                    );}
                                          puts           (P);;
                                              return+0;})
```

1. International Obfuscated C Code Contest. http ://www.ioccc.org/

Nous allons présenter maintenant deux exemples concrets où l'on peut voir l'intérêt d'une telle approche.

8.1 Cryptographie en boîte blanche

La cryptographie en boîte blanche (figure 8.1) a été introduite de manière théorique en 2002 par Chow et al[CEJVO02]. Parmi les applications industrielles on retrouve par exemple la « carte à puce virtuelle » et la gestion des droits numériques que nous verrons dans le paragraphe suivant.

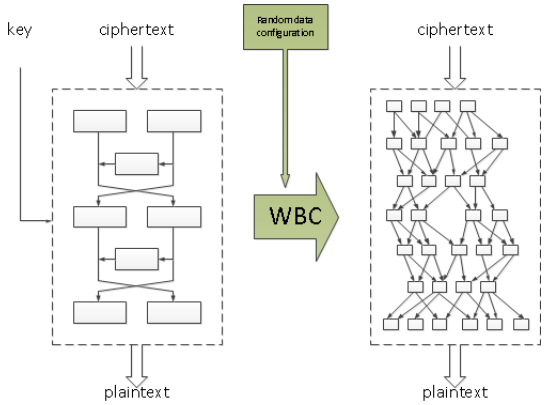


Figure. 8.1 – Cryptographie en boîte blanche : http://www.whiteboxcrypto.com/

Ces méthodes se différencient de la cryptographie classique « en boîte noire », où l'attaquant n'a accès qu'aux entrées/sorties du chiffrement et de celle « en boîte grise » où l'attaquant peut observer et parfois perturber le comportement du chiffrement via l'analyse de canaux auxiliaires. Dans une version « boîte blanche », la clef secrète se retrouve dissimulée dans la structure même du chiffrement et l'attaquant cherche donc à récupérer cette clé et contrôle complètement l'environnement d'exécution. Le problème de ces méthodes est qu'elles sont moins robustes que leur version boîte noire comme l'illustrent les travaux de Billet et al [BGEC04] et Goubin et al [GMQ07].

8.2 Gestion des droits numériques

La gestion des droits numériques ou Digital Rights Managment (DRM) a pour but de contrôler l'utilisation qui est faite des œuvres numériques. On peut l'appliquer à plusieurs supports physiques tels que les disques, les DVDs ou les logiciels ou immatériels comme la télédiffusion. Le tout est est basé sur un système d'accès conditionnel couplé à

un mécanisme de chiffrement qui permet par exemple de restreindre la lecture du média aux utilisateurs détenant une licence, ou à du matériel spécifique.

L'éditeur ou le distributeur ne confie la clé de contrôle d'accès du produit, qu'en échange d'une preuve d'achat ou de souscription à un service comme l'abonnement à une chaîne payante. Avec cette clé, la lecture ou la copie du média est autorisée en fonction des termes du contrat.

Prenons par exemple le cas de télédiffusion sur une plate-forme mobile (smartphone, tablette, ...) (figure 8.2). Une chaîne *TV_cryto* propose un service payant qui permet l'accès au contenu de la chaîne depuis un smartphone. Pour ce faire, l'utilisateur doit télécharger une application qui consiste en un lecteur spécial qui permet de déchiffrer à la volée le contenu de la chaîne. Pour l'obtenir, il doit payer un abonnement, et le lecteur lors de l'installation, récupère les spécificités du smartphone afin d'éviter d'éventuelles copies sur un autre support. Étant donné que le lecteur doit déchiffrer les fichiers du média, il doit embarquer la clé nécessaire. Si le lecteur n'a aucune protection logicielle, un attaquant pourrait récupérer cette clé et s'en servir sans payer l'abonnement requis.

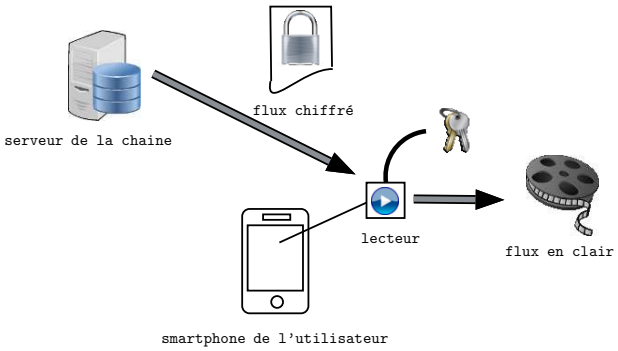


Figure. 8.2 – Télédiffusion sur smartphone

8.3 Contexte et objectifs

Les travaux de cette partie s'inscrivent dans le cadre du logiciel `LOPSI` [2], développé à l'Institut Fourier et ayant bénéficié du financement du pôle MSTIC de l'Université Joseph Fourier, du Labex AMIES et du Labex PERSYVAL. Nous présentons nos contributions à ce logiciel dans la continuité du travail de Cyril Pascal [3]. Cet outil a pour but de permettre d'exécuter des transformations de manière automatique (ou partiellement) sur du code source écrit `C`. Il a été conçu pour s'inscrire dans la chaîne de développement d'un logiciel de manière complètement transparente en n'apparaissant qu'à la fin de celle-ci.

2. Logiciel d'Obscurcissement pour la Protection des Systèmes d'Informations
3. Ingénieur financé dans le cadre du projet PACS du Labex AMIES sous la direction de Ph. Elbaz-Vincent

Notre objectif ici est de continuer l'implantation des techniques d'obscurcissement et de rajouter un module de calcul de complexité qui permet un meilleur de choix dans les techniques que l'on souhaite appliquer au code.

8.4 Plan de cette partie

Le premier chapitre sera consacré aux techniques utilisées dans l'analyse de programme. Nous présenterons les méthodes liées à l'analyse statique et celles qui concernent l'analyse dynamique. Nous verrons également comment il est possible de récupérer les supports de ces analyses à partir d'un binaire. Enfin, nous donnerons des exemples d'outils qui permettent d'analyser les programmes automatiquement ou de manière interactive.

Le deuxième chapitre présentera différentes mesures de complexité de programme. Certaines d'entre-elles sont assez génériques alors que d'autres ont été développées avec comme cible principale les transformations d'obscurcissement. Nous verrons qu'il est possible de les classer en fonction des parties du programme auxquelles elles se réfèrent.

Le troisième chapitre fera tout d'abord un état de l'art des définitions théoriques que l'on trouve pour l'obscurcissement. Nous verrons que malgré l'impossibilité d'avoir un outil d'obscurcissement « global », si l'on se concentre sur certaines propriétés d'un programme, on peut obtenir des résultats théoriques qui permettent de prouver l'existence des outils correspondants. Ensuite, nous détaillerons certaines techniques d'obscurcissement de code source. Nous nous sommes particulièrement intéressés à celles qui permettent d'augmenter la complexité du flot de contrôle du programme. Nous donnerons également un estimation de la complexité apportée par certaines d'entre elles en utilisant les mesures du chapitre précédent.

L'avant-dernier chapitre traitera des implantations réalisées. Nous présenterons l'architecture de notre outil d'obscurcissement ainsi que les fonctionnalités que nous avons implantées. Nous verrons également la partie concernant les mesures de complexité qui permet à l'utilisateur d'avoir une estimation concernant le programme de sortie. Nous discuterons également de l'ordre optimal dans lequel les techniques doivent être implantées.

Enfin, le dernier chapitre fera une conclusion sur les travaux effectués et donnera les perspectives que l'on peut entrevoir.

Chapitre 9

Analyse de programmes

Sommaire

Avant de définir de manière théorique l'obscurcissement de code et de décrire les techniques utilisées, nous allons présenter différents outils de rétro-conception qui peuvent être utilisés à la fois par les développeurs et les attaquants. Les premiers utilisent les résultats des analyses afin de trouver les vulnérabilités de leurs programmes et de les corriger ou d'estimer l'efficacité de leurs techniques de protection. De leur côté, les attaquants utilisent ces techniques afin de récupérer des données secrètes comme des clés ou de rétro-concevoir un algorithme spécifique.

Il existe deux grandes catégories d'analyse de programme :

- **l'analyse statique** : elle consiste à retrouver différentes informations sur un programme sans l'exécuter ;
- **l'analyse dynamique** : elle consiste à récupérer des données pendant l'exécution du programme.

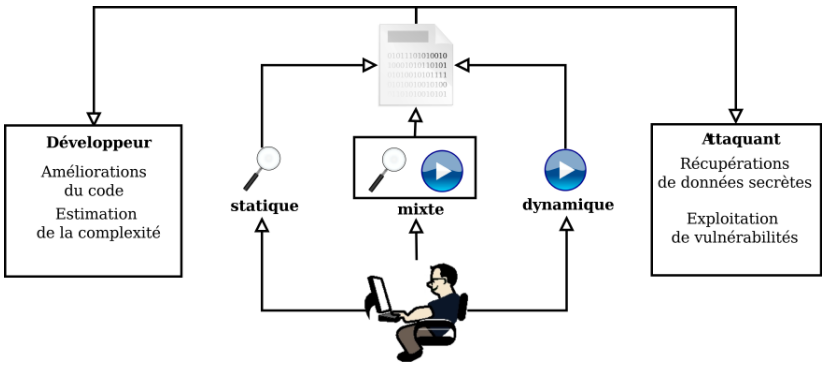


Figure. 9.1 – Analyse de programme

Dans les deux premières sections, nous allons présenter ces deux types d'analyse et déduire quels types d'information il est possible de récupérer. Ensuite, nous allons comparer différents outils de rétro-conception que l'on peut trouver dans la littérature avant de faire une conclusion sur ce chapitre et d'expliquer nos choix d'outils pour nous aider dans l'analyse de nos techniques de protection.

9.1 Analyse statique

Différents algorithmes peuvent être utilisés afin de réaliser l'analyse statique d'un programme selon la précision des résultats que l'on souhaite obtenir. De plus, il faut tenir compte du temps d'exécution qui peut être crucial dans certains contextes et fortement influencer la complexité du programme analysé.

Dans la suite, nous verrons des techniques qui se focalisent sur une fonction en parti-

culier alors que d'autres prennent en compte les interactions entre les fonctions. D'autre part, nous distinguerons l'analyse du flot de contrôle et les analyses liées aux données du programme.

9.1.1 Analyse du flot de contrôle

Afin de mettre en œuvre une analyse de flot de contrôle, il est nécessaire de travailler sur une représentation particulière des fonctions. La plupart des outils utilisent un graphe de flot de contrôle (CFG).

Définition 9.1.1
Un graphe de flot de contrôle est un graphe orienté $G = \langle V, E \rangle$ où V représente l'ensemble des blocs basiques (des sections de code avec une seule entrée et une seule sortie) et E est l'ensemble des transitions entre ces blocs (les sauts de contrôle du programme).

Une transition dans un CFG symbolise une transmission de contrôle possible entre deux blocs. Cependant, cela ne signifie pas qu'à l'exécution du programme nous aurons toujours cette transmission. Une telle représentation est par conséquent conservative car elle représente toutes les exécutions possibles du programme.

Afin de construire un CFG on peut utiliser la procédure suivante :

1. Donner un indice à chaque instruction d'un programme.
2. Marquer chaque instruction qui pourrait commencer un bloc basique (instruction de tête) :
   - La première instruction,
   - Toute cible d'une branche,
   - L'instruction suivant une branche de condition.
3. Un bloc basique correspond à l'ensemble des instructions entre une instruction de tête et la prochaine sans inclure celle-ci.
4. Ajouter un arc $A \to B$ si A termine avec une branche vers B ou si B peut succéder à A.

Exemple 9.1.2
La figure 9.2 donne un exemple de CFG obtenu pour la fonction de multiplication scalaire naïve pour les courbes elliptiques.

À partir du CFG il est possible d'identifier les boucles. Cela peut être utile quand l'on souhaite marquer certaines fonctionnalités d'un programme. Pour ce faire, il faut utiliser le principe de dominance.

```
void
ec_point_mul_naive (ec_point P3, const number n,
    const ec_point P1, const ec_curve E)
{
    field_t *k = (field_t*)E->k;
    uint8_t k_size = k->size;
    ec_point tmp;
    ec_point_tmp_alloc(tmp, k);

    number m;
    number_tmp_alloc(m, k_size);
    number_copy(m, n);

    ec_point_copy(tmp, P1);
    ec_point_set_neutral(P3, E);

    while (!number_iszero(m))
    {
        if (LIMB(m)[0] & (block)1)
    {
            ec_point_add(P3, P3, tmp, E);
        }
        ec_point_dbl(tmp, tmp, E);
        number_rshift(m, m, 1);
    }

    ec_point_tmp_free(tmp);
    number_tmp_free(m);
}
```

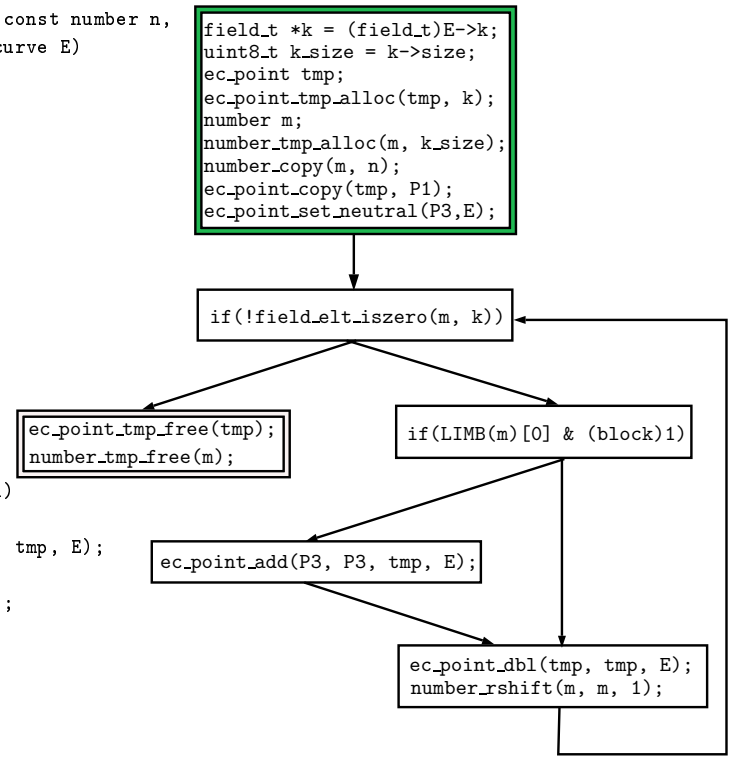


Figure. 9.2 – Graphe de flot de contrôle de la fonction de multiplication scalaire naïve

Définition 9.1.3
Un noeud A d'un CFG domine un autre noeud B si tous les chemins partant de B doivent passer par A. De plus si A domine immédiatement B, l'arc entre les deux noeuds est dit de retour.

En utilisant cette relation, il est possible de construire un arbre de dominance. Cet arbre permet d'identifier les boucles de la manière suivante. D'une manière générale, un noeud n est dans une boucle si :

- il est dominé par le noeud de tête,
- le noeud de tête peut être atteint à partir de lui,
- il y a un seul un arc de retour.

Exemple 9.1.4
Reprenons notre exemple pour illustrer l'identification (voir figure 9.2). Son arbre de dominance est donné dans la figure 9.3. Il n'y a qu'un arc de retour entre B_5 et B_1. Par conséquent B_1 sera le noeud de tête de la boucle. Les noeuds dominés par B_1 et depuis lesquels on peut atteindre B_1 sont B_3, B_4, B_5. On peut donc déduire que les noeuds B_1, B_3, B_4 et B_5 forment une boucle.

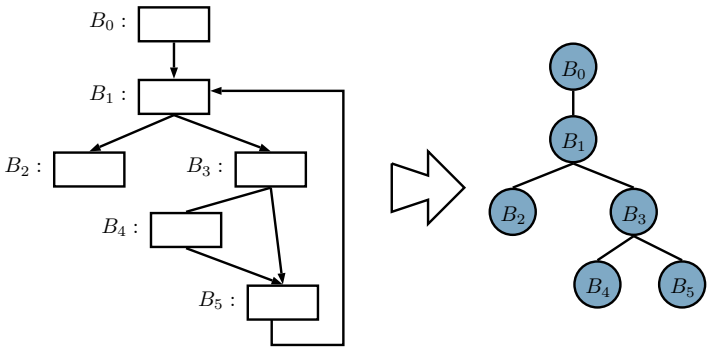


Figure. 9.3 – Arbre de dominance de la fonction de multiplication scalaire naïve

L'analyse du code peut se faire à trois niveaux. Elle peut être

locale : on analyse les blocs basiques séparément,

intra-procédurale : on analyse le flot au niveau d'une fonction,

inter-procédurale : on prend en compte également l'interaction entre les fonctions.

Dans le cas d'une analyse inter-procédurale, en plus du CFG de chaque fonction, il faut construire un graphe d'appel pour le programme.

Définition 9.1.5
Un graphe d'appel possède un noeud pour chaque fonction du programme et il y a un arc $f \to g$ si la fonction f est susceptible d'appeler g.

Un cycle dans le graphe d'appel permet d'identifier la récursion et un noeud qui n'est pas atteignable depuis le point d'entrée indique que la fonction correspondante n'est jamais exécutée.

Au lieu de considérer le CFG et le graphe d'appel séparément, il est possible de créer un graphe mixte en remplaçant les noeuds du graphe d'appel par les CFG des fonctions correspondantes.

En général, il est assez facile de construire de tels graphes. Cependant l'utilisation de pointeurs de fonction peut fausser l'analyse. Prenons par exemple le code suivant et son graphe d'appel associé (voir figure 9.4). Dans le graphe d'appel nous avons un noeud isolé donc on pourrait conclure que la fonction associée n'est jamais appelée ce qui est faux car elle est appelée via un pointeur. En présence d'un tel code, il est donc nécessaire de faire une analyse du flot de données ou de pointeurs afin d'obtenir un graphe correct.

```
void
ec_point_mul_naive (ec_point P3, const number n,
    const ec_point P1, const ec_curve E)
{
...
}

void t()
{
  printf("debut\n");
}

void h()
{
  printf("fin\n");
}

int main()
{
    t();
    void (*p)() = &ecc_point_mul_maive;
    p();
    h();
}
```

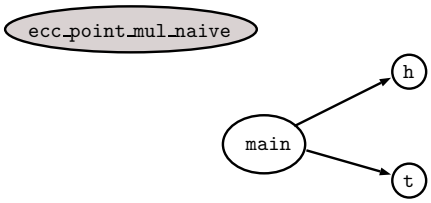


Figure. 9.4 – Graphe d'appel avec pointeurs de fonction

9.1.2 Analyse des données

On va s'intéresser ici à deux types d'analyse, l'analyse du flot de données et celle de la dépendance entre les données.

L'analyse du flot de données permet d'obtenir des informations sur comment les variables d'un programme sont utilisées. Elle s'appuie sur un CFG dans le cas général mais il faut aussi utiliser le graphe d'appel si l'on souhaite réaliser une analyse inter-procédurale. Elle répond à trois problématiques :

- La valeur de la variable x sera-t-elle utilisée après le point p ?
- Dans quelle partie du programme une variable x utilisée au point p a-t-elle été affectée ?
- Est-ce qu'une variable x est une constante à un point p et si tel est le cas, quelle est sa valeur ?

L'analyse de la dépendance entre les données peut être réalisée selon quatre critères. Soient A et B deux instructions. Il y a :

- une dépendance de flot entre A et B si A affecte une variable et que B l'utilise,

$$A : x = 6;$$

$$B : y = x \times 7;$$

- une anti-dépendance si A lit une variable et que B l'affecte,

$$A : y = x \times 7;$$

$$B : x = 6;$$

- une dépendance de sortie si A et B affectent la même variable,

$$A : x = 6 \times y;$$

$$B : x = 7;$$

- une dépendance de contrôle entre A et B si la sortie de A décide si oui ou non B s'exécute.

$$A : if(\dots)$$

$$B : x = 7;$$

9.1.3 Analyse d'alias

Définition 9.1.6
On dit que deux pointeurs sont alias l'un de l'autre s'ils pointent vers la même zone mémoire.

Les alias peuvent apparaître sous différentes formes dans un programme :

- quand on appelle une fonction avec deux paramètres formels qui renvoie à la même zone mémoire,
- en passant en paramètre d'une fonction une variable globale,
- par effet de bord d'une autre fonction,
- en utilisant un tableau.

Le problème d'alias consiste à déterminer si à un point p d'un programme, deux variables peuvent ou doivent pointer vers la même zone mémoire. On peut distinguer deux cas, les problèmes d'alias possible (*may-alias*) et d'alias obligatoire (*must-alias*). Considérons deux références mémoire a et b. À un point p du programme, $\langle a, b\rangle \in \textit{may-alias}(p)$ s'il existe une exécution pour laquelle a et b pointent vers la même zone. Si pour toutes exécutions, cela est le cas alors $\langle a, b\rangle \in \textit{must-alias}(p)$. On peut illustrer cela avec la figure 9.5.

L'analyse d'alias sur un programme peut être très longue si l'on souhaite avoir des résultats précis. Deux approches sont possibles :

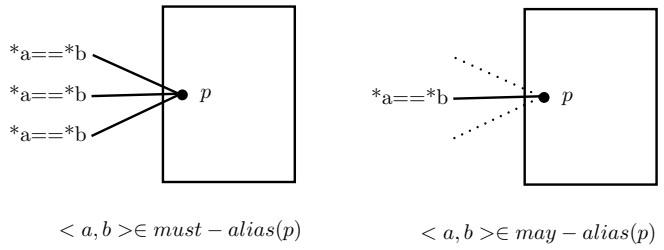


Figure. 9.5 – Alias possible et obligatoire

- On peut mener une analyse sensible au flot de contrôle qui calcule à chaque point du programme les ensembles d'alias. Plus précis mais plus long.
- On peut choisir de ne pas prendre en compte le flot de contrôle. L'analyse est plus rapide mais moins précise.

La différence de précision peut se voir quand on a une affectation conditionnelle de pointeurs. De plus, comme pour les analyses précédentes, on peut prendre en compte ou pas les interactions entre les fonctions.

En général, les algorithmes utilisés en pratique sont à fois insensibles au flot de contrôle et aux appels de fonctions pour une question de rapidité.

9.1.4 Autres types d'analyse

La méthode de découpage ou slicing permet en autre de déterminer pour une variable x à un point p d'un programme, l'ensemble des instructions qui ont contribué à sa valeur (*backwards_slice*). On peut l'utiliser par exemple pour trouver une partie de code qui ne contient pas de données sensibles et que l'on pourrait stocker sur un serveur dans une approche « exécution côté serveur ». D'un autre côté, on peut s'en servir dans l'évaluation des conditions de contrôle.

L'interprétation abstraite consiste à faire une analyse statique du programme en remplaçant les opérations élémentaires par une abstraction de leur interprétation standard. Par exemple, il est possible d'abstraire très facilement un programme qui détermine la parité d'un entier. On définit tout d'abord le domaine d'abstraction

$$Parity = \{\mathbb{Z}, pair, impair, inconnu\}.$$

On peut définir des tables de valeurs d'abstraction pour plusieurs opérateurs :

x	y	$x *_a y$	$x +_a y$
$pair$	$pair$	$pair$	$pair$
$pair$	$impair$	$pair$	$impair$
$impair$	$pair$	$pair$	$impair$
$impair$	$impair$	$impair$	$pair$

Cette technique est utilisée par exemple pour simplifier des expressions booléennes ou des conditions de structure de contrôle [PMBG06].

9.2 Analyse dynamique

Lors d'une analyse dynamique, on observe les chemins d'exécution et les modifications de données. Au contraire de l'analyse statique, l'information récupérée n'est pas conservative, elle ne peut pas être étendue à toutes les exécutions possibles d'un programme. Les techniques utilisées sont les mêmes qu'un développeur qui souhaiterait déboguer son programme.

9.2.1 Débogage

Le débogage est utilisé afin de réaliser des exécutions pas-à-pas d'un programme. Il est possible d'utiliser des points d'arrêt afin d'analyser plus précisément le contexte à travers les valeurs de variable ou de registre. C'est un outil utilisé de manière interactive.

Les points d'arrêt peuvent être de deux natures, logicielle et matérielle. Il est possible de mettre un nombre arbitraire de points d'arrêt logiciels mais cela modifie le programme que l'on analyse. Cette action peut donc être détectée par celui-ci. Les points d'arrêt matériels sont implantés directement par le CPU. Cette méthode est très rapide et ne nécessite pas de modification du programme.

Le débogage relatif permet d'exécuter deux ou plusieurs versions similaires d'un programme sur les mêmes entrées. Il met en lumière les différences dans les flots de contrôle ou les valeurs des variables. Pour un programmeur, c'est une technique utile pour déboguer en comparant la version qui pose problème à une précédente qui était valide. Un attaquant peut également tirer partie de cette fonctionnalité afin de déterminer les parties différentes de deux copies d'un même logiciel. Ces parties pourraient par exemple cacher une clé de chiffrement.

9.2.2 Profilage

Le profilage d'une exécution consiste à récupérer le nombre de fois qu'une partie de code a été exécutée ou le temps passé dans celle-ci. En général, on utilise cette technique afin de corriger des bugs ou des problèmes de performance. De plus, les compilateurs profilent aussi le code afin de déterminer quelle partie du code doit être optimisée.

Pour un attaquant, cette technique peut lui permettre de voir certaines relations entre des parties de programme comme par exemple deux fonctions qui s'exécutent un même nombre de fois.

Afin de profiler un code, il faut rajouter des instructions qui vont permettre de récolter les informations d'exécution. Il est possible de le faire directement dans le code ou on peut utiliser sur certains compilateurs un flag qui leur indique que l'on souhaite profiler le code. Cette étape s'appelle l'instrumentation. Par exemple avec le compilateur `gcc`, le flag est `-pg`. On obtient ainsi un exécutable instrumenté.

Ensuite, il faut lancer plusieurs exécutions de ce programme instrumenté avec différentes entrées afin d'obtenir une meilleure couverture du code. Pour chaque exécution, un fichier de profilage est créé. Il faut donc compiler tous ces fichiers et utiliser un programme d'analyse qui permettra d'avoir un rapport détaillé. On peut par exemple utilisé l'outil `gprof`.

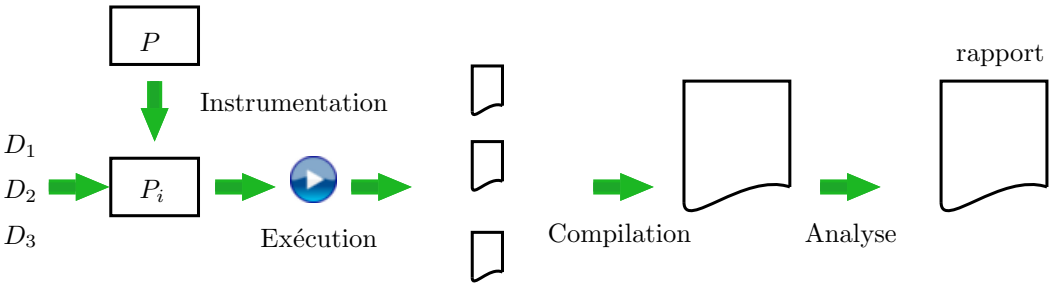


Figure. 9.6 – Profilage de code

Un outil de profilage peut également compter le nombre de fois qu'un bloc basique a été exécuté ou qu'un arc a été utilisé.

9.2.3 Autres techniques

Le traçage de code peut être vu comme une version mixte entre le débogage et le profilage. Il permet de collecter la liste des blocs basiques et des arcs qui sont exécutés (profilage avancé). Il n'est pas interactif comme le débogage car les traces sont généralement

analysées après l'exécution complète du programme. Cette technique permet d'avoir une vue d'ensemble de l'exécution et les traces si elles sont singulières peuvent révéler quel algorithme a été utilisé. Par exemple, on pourrait identifier la trace caractéristique d'un mécanisme cryptographique.

Un émulateur peut être vu comme une implantation logicielle d'une plate-forme matérielle. Le but est de permettre de démarrer n'importe quel système d'exploitation qui ne devait à l'origine s'exécuter que sur une cible matérielle particulière. Il peut être utilisé pour lancer un jeu vidéo sur une autre console. On peut également l'utiliser comme débogueur. Certains programme peuvent se prémunir de cette technique en utilisant des requêtes régulières au dongles de l'ordinateur.

La teinte de données dynamique est une méthode utilisée dans l'analyse de dépendance de données pendant l'exécution d'un programme. Elle permet de trouver des vulnérabilités, d'améliorer la couverture des tests, de déboguer un programme (déréférencement de pointeur nul,...) [CGPDE06 ; FMP14 ; MPL04 ; BRS15]. Elle consiste à marquer, pour une modification d'une certaine variable, toutes celles qui sont impactées. La plus part du temps, ce sont les données d'entrée qui sont teintées. Lors de l'analyse, il faut également spécifier les propriétés que l'on souhaite vérifier, par exemple si une entrée influe sur la condition d'une boucle. Il est possible d'utiliser un outil automatique ou de le faire manuellement.

9.3 Récupération des données nécessaire à l'analyse

Nous allons dans cette section, présenter deux méthodes permettant de récupérer le support des analyses de programme : le désassemblage et la décompilation. Nous verrons en particulier que la récupération de données n'est pas évidente sur des codes qui présentent certaines particularités.

9.3.1 Désassemblage

Le désassemblage consiste à transformer un binaire de manière à récupérer les CFGs du programme ainsi que les instructions assembleurs utilisées. Deux cas peuvent alors se présenter. Si c'est le propriétaire du code qui réalise cette action, il peut le faire sur un binaire qui contient encore toutes les informations des tables de symboles intactes. Il a ainsi la connaissance du point d'entrée du programme, ainsi que les début et fin de fonctions. L'attaquant lui, n'aura à sa disposition qu'un binaire dénudé, où toutes ces informations ont été enlevées. La tâche est donc plus complexe. Il peut tenter un désassemblage statique et essayer de retrouver le code assembleur ou bien, il peut faire un désassemblage

dynamique qui consiste à faire tourner le programme afin de récupérer des traces utiles et à ne désassembler que les chemins qui l'intéresse. Une approche hybride est également possible. Certains outils facilitent le désassemblage statique avec des informations collectées lors de l'exécution du programme.

La difficulté de cette technique réside en partie dans les problématiques suivantes :

1. Les données sont souvent mélangées au code donc il peut arriver que l'on prenne une donnée pour une instruction.
2. Il faut déterminer les cibles des sauts indirects.
3. Retrouver le début d'une fonction qui n'est appelée que par des appels indirects est très difficile. L'utilisation des pointeurs de fonction perturbent fortement le désassemblage.
4. La fin d'une fonction peut également poser problème lorsqu'il n'y a pas d'instruction de retour dans le code. Cela est le cas par exemple pour les codes assembleurs écrits à la main. De plus, ce type de code est souvent atypique car ils ne présentent les sections de début et fin comme les autres fonctions.
5. Les codes qui s'auto-modifient sont également complexes.

9.3.2 Décompilation

L'étape initiale de la décompilation est le désassemblage vu précédemment. Avec les CFGs obtenus, l'objectif est de transformer chaque bloc basique en des séquences d'affectation et le graphe lui même doit être traduit en langage de haut niveau en identifiant les structures de contrôle.

Cette technique pose certains problèmes de correction du code obtenu. En effet comme l'étape préliminaire consiste à désassembler le programme, si cela induit une erreur dans la construction des CFGs elle sera répercutée sur l'étape de décompilation. De plus, les problématiques suivantes apparaissent :

1. Même si l'on considère que le code désassemblé est correct, certaines instructions peuvent ne correspondre à aucun code source du langage visé. Cela peut être le cas avec du code assembleur optimisé écrit à la main.
2. Il faut également identifier les appels aux fonctions de la librairie standard du langage utilisé, comme par exemple les `printf` pour le `C`.
3. Certains compilateurs utilisent des instructions spéciales lors de la compilation du code et rajoutent également des fragments de code nécessaires à la gestion de la mémoire. Il faut donc distinguer ces parties du code à examiner.
4. L'identification des structures de contrôle peut être rendue difficile par des imbrications de plusieurs niveaux.

5. De plus, il faut reconnaître les structures du code d'origine comme par exemple, les tableaux, les pointeurs, les structures, etc.

9.4 Outils d'analyse de code

Il est possible d'utiliser des outils qui sont spécialisés dans l'une ou l'autre des techniques rappelées précédemment. On peut par exemple utiliser comme débogueur `gdb`, comme profileur `gprof` ou comme traceur `ptrace`. Des outils comme `okteta` permettent d'éditer directement le code binaire. Pour le désassemblage, des outils comme `objdump`, `readelf` peuvent être utiles.

On peut également utiliser des outils plus complets qui ont plus de fonctionnalités et qui facilitent les techniques que nous avons rappelées. Nous allons maintenant présenter les résultats de construction de CFG de trois d'entre eux, IDA-pro[IDA], Metasm[Metasm] et Hopper[Hopper] avec comme exemple le code suivant.

```
#include <stdio.h>

int prem(int i)
{
    int q;
    int r;
    int (*XjjaExuf)();
    int (*QKqHdQZF[5])();
    int UCDQpgnT()
    {
        int (*bankUhHH)();
        r += 9;
    }

    int PoZjsyCs()
    {
        int (*uViMdUqU)();
        uViMdUqU = QKqHdQZF[(1 * 3) % 5];
        for (q = 9; q < 39; q++)
        {
            uViMdUqU();
        }
    }
```

```
    int aOlIFMKT()
    {
        int (*xoAmmSgt)();
        q = 6 + 9;
    }

    QKqHdQZF[(0 * 3) % 5] = PoZjsyCs;
    QKqHdQZF[(1 * 3) % 5] = UCDQpgnT;
    QKqHdQZF[(2 * 3) % 5] = aOlIFMKT;
    q = 9;
    int (*zNKgiSdT)();
    zNKgiSdT = QKqHdQZF[(0 * 3) % 5];
    int (*GJGtEPeN)();
    GJGtEPeN = QKqHdQZF[(2 * 3) % 5];
    for (r = 0; r < 9; r++)
    {
        zNKgiSdT();
        if (i < r)
        {
            break;
        }

        GJGtEPeN();
    }
    printf("q :%d\n", q);
}

int main()
{
    return prem(15);
}
```

Sur cet exemple, les outils `Metasm` et `Hopper` renvoient le même graphe de flot de contrôle (voir figure 9.7) alors que `IDA-Pro` n'a pas réussi à retourner un graphe.

L'observation du graphe nous montre cependant qu'une simple analyse statique ne permet pas de récupérer l'intégralité des informations sur le programme. Une analyse dynamique est donc nécessaire.

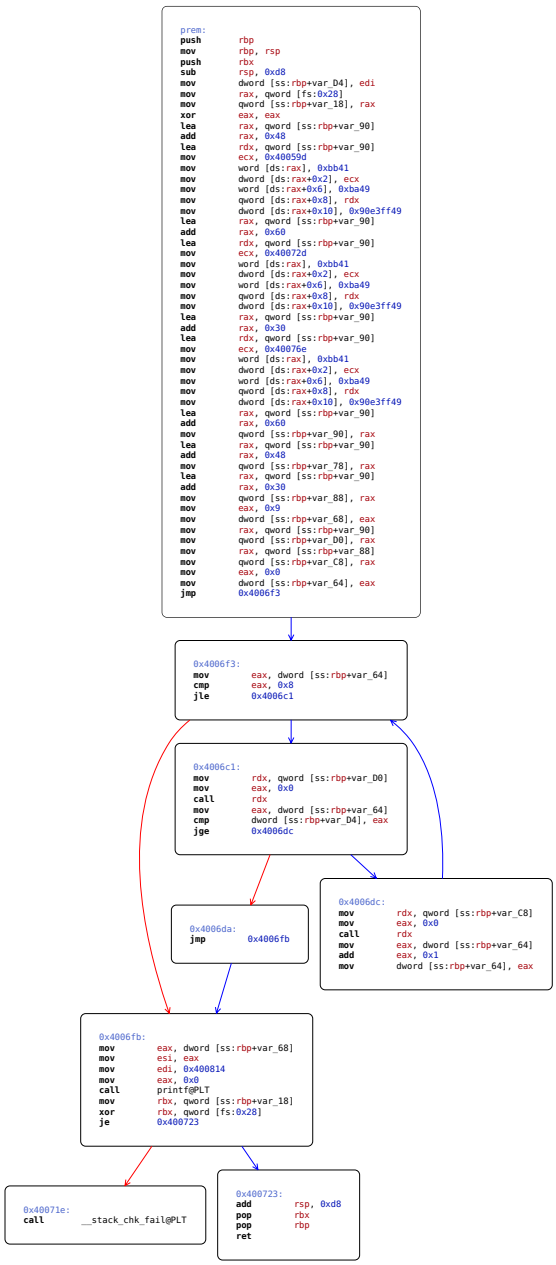


Figure. 9.7 – Graphe de flot de contrôle d'un code en présence de tableau de pointeurs de fonctions avec `Hopper`

9.5 Conclusion

Dans ce chapitre, nous avons présenté différentes techniques d'analyse de programme. L'approche statique permet de récupérer des informations qui valables pour toutes les exécutions d'un programme, cependant certains problèmes comme la présence de pointeurs rendent l'analyse très complexe. De même, le graphe produit peut contenir un grand nombre de chemins possibles alors que certains ne sont jamais exécutés. L'approche dynamique permet d'avoir des informations sur le comportement du programme, mais cela n'est valable que pour une exécution. Cependant, il est plus aisé de tracer une variable ou un pointeur et de détecter les parties de code mort ou similaires.

Le problème qui semble freiner le plus l'ensemble des approches est la présence de pointeurs, car l'analyse est en théorie NP-difficile, et la présence d'un grand nombre de branchements à l'intérieur du programme. Cela constitue donc une piste pour la création de transformations d'obscurcissement résistantes aux analyses de programme.

Parmi tous les outils disponibles, nous avons présenté certains d'entre eux et constaté les différences entre les résultats sur la compréhension d'un programme complexe qui contenait des pointeurs de fonctions. Notre choix d'outils pour la validation des techniques se limite pour l'instant à `Metasm` et `Hopper`. Cependant, il serait nécessaire, dans la suite des travaux, de mettre en place des analyses dynamiques afin de mieux appréhender la présence de pointeurs.

Chapitre 10

Mesure de complexité

Sommaire

Dans [CTL97], la qualité d'une transformation d'obscurcissement repose sur trois critères : la puissance, la résilience et le coût de la transformation. Soient $\mathcal{T}$ une transformation que l'on souhaite analysée et P le programme que l'on veut obscurcir et $P' = \mathcal{T}(P)$.

La puissance de la transformation $\mathcal{T}_{pot}(P)$ mesure la différence de complexité entre P et P'. Soit M une mesure de complexité.

$$\mathcal{T}_{pot}(P) = M(P')/M(P) - 1.$$

La résilience de la transformation $\mathcal{T}_{res}(P)$ mesure l'effort nécessaire afin de reconstruire P à partir de P'.

$$\mathcal{T}_{res}(P) = \begin{cases} \text{à sens unique} & \text{Si de l'information a été enlevée de } P \text{ de telle manière que } P \text{ ne puisse être reconstruit à partir de } P', \\ R(\text{effort } \mathcal{T}_{de-ob}, \text{ effort } \mathcal{T}_{prog}) & \text{Sinon.} \end{cases}$$

avec R une matrice définie par

effort $\mathcal{T}_{prog}$ / effort $\mathcal{T}_{de-ob}$	Local	Global	Intra-procédural	Inter-procédural
polynomial	trivial	faible	fort	complet
exponentiel	faible	fort	complet	complet

L'effort de $\mathcal{T}_{prog}$ est le temps nécessaire pour construire un outil de dé-obscurcissement, qui soit capable de réduire la puissance de la transformation. L'effort $\mathcal{T}_{de-ob}$ regroupe le temps d'exécution et l'espace mémoire nécessaire pour un tel outil.

Le coût de la transformation $\mathcal{T}_{cout}(P)$ est défini comme le surcoût en temps d'exécution et en occupation d'espace mémoire entre les programmes P' et P.

$$\mathcal{T}_{cout}(P) = \begin{cases} \text{très élevé} & \text{Si l'exécution de } P' \text{ requiert exponentiellement plus de ressources que } P, \\ \text{élevé} & \text{Si l'exécution de } P' \text{ requiert } O(n^p), p > 1 \text{ plus de ressources que } P, \\ \text{faible} & \text{Si l'exécution de } P' \text{ requiert } O(n) \text{ plus de ressources que } P, \\ \text{nul} & \text{Si l'exécution de } P' \text{ requiert } O(1) \text{ plus de ressources que } P, \end{cases}$$

où n représente le temps d'exécution et l'occupation espace mémoire de P.

Nous allons étudier dans ce chapitre les mesures de complexité qui permettent de définir la puissance d'une transformation. Les métriques peuvent être classées en fonction des trois propriétés fondamentales des programmes :

- les instructions : un programme devient plus complexe quand le nombre d'instructions (potentiellement exécutables) augmentent.
- le flot de contrôle : l'ordre selon lequel les instructions sont exécutées révèle le flot de contrôle du programme. Plus ce flot est compliqué, plus le programme est considéré comme complexe.
- le flot de données : cette propriété concerne la production et la consommation des valeurs numériques par chaque instruction.

Dans la littérature, on peut trouver des taxonomies de métriques pouvant être utilisées pour juger de la qualité d'une technique d'obscurcissement [AMSBBP07 ; CK94].

10.1 Instructions

Nous allons tout d'abord analyser une suite de mesures liées au nombre d'instructions d'un programme.

10.1.1 Métriques d'Halstead

Les métriques de complexité de Halstead ont été développées dans [Hal77]. Elles font parties des mesures les plus anciennes. Elles se focalisent sur la complexité liée aux instructions et ne prennent pas en compte le graphe de flot de contrôle.

Le code source est interprété comme une séquence d'opérateurs et d'opérandes. Une opérande est soit un nom de variable (pas de mot réservé), un nom de type, un type, ou une constante (numérique ou chaîne de caractères). Les opérateurs regroupent tous les autres mots réservés, les opérateurs arithmétiques ainsi que les caractères spéciaux.

À partir de l'analyse des opérateurs et des opérandes, quatre quantités sont dérivées :

- le nombre d'opérateurs distincts n_1.
- le nombre d'opérandes distinctes n_2.
- le nombre total d'opérateurs N_1.
- le nombre total d'opérandes N_2.

Ces quantités peuvent être combinées afin d'obtenir d'autre mesures :

- La taille du programme N est la somme totale d'opérateurs et d'opérandes dans celui-ci :
$$N = N_1 + N_2.$$
- La taille du vocabulaire n est la somme d'opérateurs et d'opérandes uniques :
$$n = n_1 + n_2.$$
- Le volume du programme V est la quantité d'information contenue dans celui-ci, il est mesuré en bits. Il est calculé à partir de taille du programme et de son vocabulaire :
$$V = N * \log_2(n).$$
- Le niveau de difficulté D du programme est proportionnel au nombre d'opérateurs uniques. Il est également proportionnel au ratio entre le nombre total d'opérandes et la quantité d'opérandes uniques :
$$D = (n_1/2) * (N_2/n_2).$$
- Le niveau du programme L est l'inverse du niveau de difficulté. Par exemple, un programme ayant un petit niveau est plus susceptible de provoquer des erreurs lors de l'analyse :
$$L = 1/D.$$
- L'effort d'implantation E ou de compression d'un programme est lié au volume et au niveau de difficulté du programme :
$$E = V * D.$$
- Le temps d'implantation ou de compréhension du programme T est proportionnel à l'effort E. Des résultats empiriques permettent de calibrer cette quantité.
Halstead a établi qu'en divisant l'effort par 18, on obtient une approximation du temps en secondes :
$$T = E/18.$$
- Le nombre de bugs reportés B est corrélé à la complexité globale du programme. Halstead a donné la formule suivante pour le calculer :
$$B = (E^{(2/3)})/3000.$$
Cette quantité donne une estimation du nombres erreurs qui se produisent lors de l'implantation.

Exemple 10.1.1
Pour la fonction de multiplication scalaire naïve pour les courbes elliptiques (voir figure 9.2), nous obtenons les résultats suivants.

n_1	n_2	N_1	N_2	n	N	V	D	L	E	T
15	27	70	51	42	121	652.470	14.167	0.071	9243.331	8 min 14 sec

En fonction du contexte dans lequel on se trouve, il faut interpréter différemment ces mesures. Par exemple, un développeur cherchera à minimiser le temps de compréhension du programme alors que s'il l'on cherche à obscurcir un programme, il est préférable que cette quantité soit élevée.

10.2 Flot de contrôle

Nous présentons dans cette section différentes méthodes qui utilisent le graphe du flot de contrôle afin d'extraire de l'information sur la complexité d'un programme.

10.2.1 Nombre cyclomatique

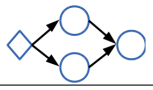
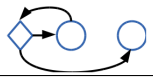
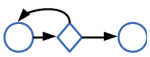
Dans [McC76], McCabe a proposé une méthode consistant à mesurer et à contrôler le nombre de chemins dans un programme. Pour ce faire, il utilise la notion de nombre cyclomatique d'un graphe.

Définition 10.2.1
Le nombre cyclomatique $CC(G)$ d'un graphe G ayant n sommets, e arcs et p parties connexes est défini par

$$CC(G) = e - n + 2p.$$

On peut définir le nombre cyclomatique d'un programme P comme celui du graphe de flot de contrôle qui lui est associé. On le notera $CC(P)$.

Exemple 10.2.2
Par exemple, voici les nombres cyclomatiques associés aux graphes de flot de contrôle des constructeurs usuels..

if		2
while		2
do while		2

Proposition 10.2.3 ([McC76])
Pour tout graphe G, son nombre cyclomatique $CC(G)$ vérifie les propriétés suivantes :

- $CC(G) \geqslant 1$.
- $CC(G)$ *est le nombre maximal de chemins linéairement indépendants dans G.*
- *Insérer ou supprimer des appels de fonction dans G ne change pas* $CC(G)$.
- *G a un unique chemin si et seulement si* $CC(G) = 1$.
- *Insérer un nouvel arc dans G augmente* $CC(G)$ *de* 1.
- $CC(G)$ *dépend seulement de la structure décisionnel de G.*

Simplifier en utilisant des prédicats McCabe a également lié le nombre cyclomatique d'un programme structuré P au nombre de prédicats qu'il contient. Pour ce faire, il utilise le résultat suivant de Mills [Mil72].

Lemme 10.2.4
Soient α le nombre de fonctions, β le nombre de prédicats. Alors, e le nombre d'arcs peut être défini par l'équation suivante

$$e = 1 + \alpha + 3\beta.$$

Pour tous les nœuds prédicats, il y a exactement un noeud collecteur et, il y a un unique noeud d'entrée et un unique noeud de sortie. Par conséquent, on peut écrire

$$n = \alpha + 2\beta + 2.$$

On peut supposer que $p = 1$ puisque nous avons un programme structuré. Nous obtenons donc :
$CC(P) = (1 + \alpha + 3\beta) - (\alpha + 2\beta + 2) + 2 = \beta + 1$.

Simplification utilisant la forme du graphe Cette simplification utilise la formule d'Euler suivante. Si G est un graphe connecté avec n sommets, e arcs, et r régions, alors $n - e + r = 2$. On peut donc déduire que le nombre de région est égal au nombre cyclomatique. Par conséquent, le calcul de $CC(G)$ peut se réduire au comptage des régions.

Amélioration de Myers

La complexité cyclomatique de McCabe ne permet pas de distinguer les cas où il y a une ou plusieurs conditions dans une instruction de contrôle. Ainsi, deux programmes avec des prédicats de complexité différente auront la même complexité cyclomatique.

D'après cette observation, Myers [Mye77] a suggéré qu'il fallait prendre en compte les conditions individuelles afin d'obtenir une mesure de complexité plus pertinente.

Sa mesure est donnée par un intervalle $[L, H]$ défini par :

- L est la complexité cyclomatique de McCabe,
- H est obtenu en ajoutant à L le nombre de conditions individuelles (0 pour un prédicat simple, $n-1$ pour un prédicat à n arguments) moins 1.

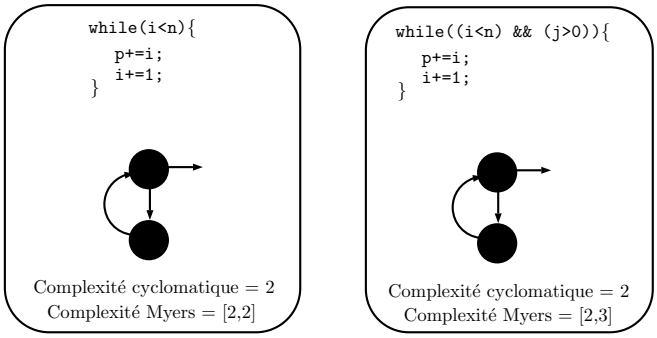


Figure. 10.1 – Exemple de la complexité de Myers

Cette mesure permet donc de distinguer des programmes ayant des graphes de flot de contrôle similaires.

10.2.1.1 Complexités cyclomatiques d'une fonction

Dans [MZK13], les auteurs ont proposé d'améliorer la mesure de McCabe [McC76] en analysant les interactions entre les fonctions. Ils ont ainsi introduit deux nouvelles métriques, la complexité cyclomatique totale (TCC) et la complexité cyclomatique couplée (CCC).

La mesure de McCabe ne tient pas compte des appels de fonctions. Or, une fonction peut appeler une autre ayant une complexité plus élevée. Il est donc intéressant de modifier la complexité d'une fonction en y ajoutant la complexité de toutes les fonctions appelées par celle-ci. La complexité cyclomatique totale est définie de la manière sui-

vante :

$$TCC(f) = CC(f) + \sum_{i=1}^{n} CC(f_i) - n,$$

où CC est la complexité cyclomatique et f_i les fonctions appelées par f.

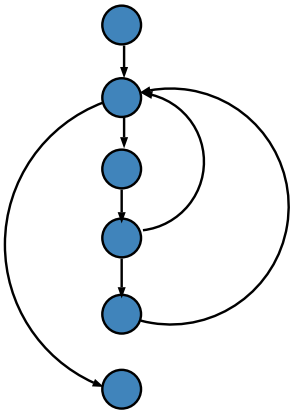
Considérons l'exemple suivant.

```
int f1(int u, int v) {
    int r = 1;
    int i = u;
    while(i > 0) {
       r +=v;
       if (i %2 == 0) {
          r +=u;
       }
    }
    return r;
}
```

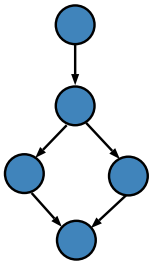
```
int f2(int u, int v) {
    int r = u;
    if(v > r) {
       r *=v;
    }else{
       r *=(u + v);
    }
    return r;
}
```

```
int f(int a, int b, int c) {
   int r = 0;
   if(a == 10)
   {
     r = f1(c, b + 10);
   }else{
     if( f2(b, c + 10) > 100) {
        r = 100;
     }
   }
} return r;
```

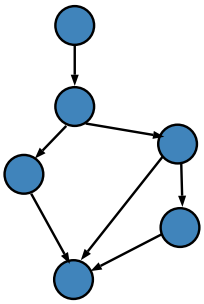
CFG de `f1`



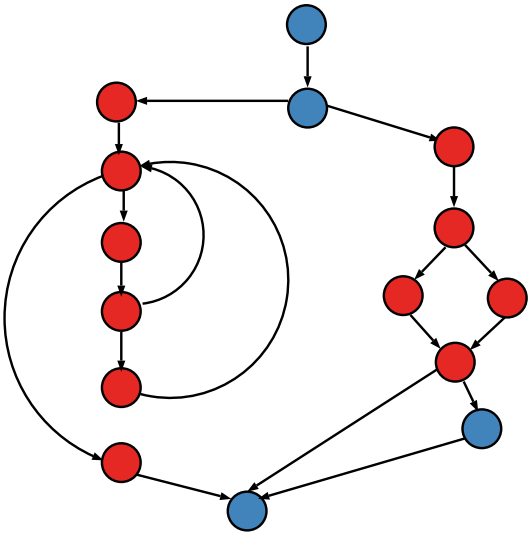
CFG de `f2`



CFG de `f`



CFG de `f` enrichi avec `f1` et `f2`



Nous avons $CC(f1) = 3$, $CC(f2) = 2$ et $CC(f) = 3$. En appliquant la formule de TCC, on obtient

$$TCC(f) = CC(f) + CC(f1) + CC(f2) - 2 = 6.$$

Si on utilise le CFG de f enrichi avec ceux de $f1$ et $f2$, on peut remarquer que le nombre cyclomatique de ce graphe est égal à $TCC(f)$.

L'interaction entre deux fonctions peut être analysée en utilisant la notion de couplage. Il y a cinq types de couplage et à chacun est associée une valeur α.

Valeur de couplage	Type de couplage	description
1	Données	Une fonction passe des données en paramètre d'une autre fonction (scalaire ou tableau).
2	Empreinte	Une fonction passe une structure en paramètre d'une autre fonction.
3	Contrôle	Une fonction passe en paramètre un flag qui est utilisé pour contrôler la logique interne d'une autre fonction.
4	Global	Deux fonctions utilisent la même donnée globale.
5	Contenu	Une fonction fait référence aux variables internes d'une autre fonction.

Tableau 10.1 – Valeurs de couplage associées au comportement d'une fonction

La complexité cyclomatique couplée est définie par :

$$CCC(f) = CC(f) + \sum_{i=1}^{n} \alpha_i CCC(f_i) - n,$$

où CC est la complexité cyclomatique et f_i les fonctions appelées par f et α_i est la valeur de couplage entre f et f_i. S'il y a plusieurs interactions entre deux fonctions, on prend la valeur de couplage la plus élevée.

Dans notre exemple, `f` passe b et $c + 10$ en paramètre de `f2` donc $\alpha_2 = 1$. `f` passe c et $b + 10$ en paramètre de `f1` donc a priori $\alpha_1 = 1$. Cependant, la valeur de retour de `f1` contrôle un `if` dans `f`. Par conséquent, on a $\alpha_1 = 3$. On obtient donc

$$CCC(f) = CC(f) + 3 * CC(f1) + 1 * CC(f2) - 2 = 10.$$

10.2.2 Profondeur d'imbrication

L'utilisation des niveaux d'imbrication dans le calcul de la complexité d'un programme a été proposée par W. Harrison et K. Magel ([HM81]). Ils ont remarqué que les méthodes

précédentes basées sur l'analyse de flot de contrôle ne prenaient pas en compte tous les effets de deux facteurs majeurs de la complexité, à savoir la complexité des blocs individuels (magnitude du programme) et celle du programme, en particulier les niveaux d'imbrication.

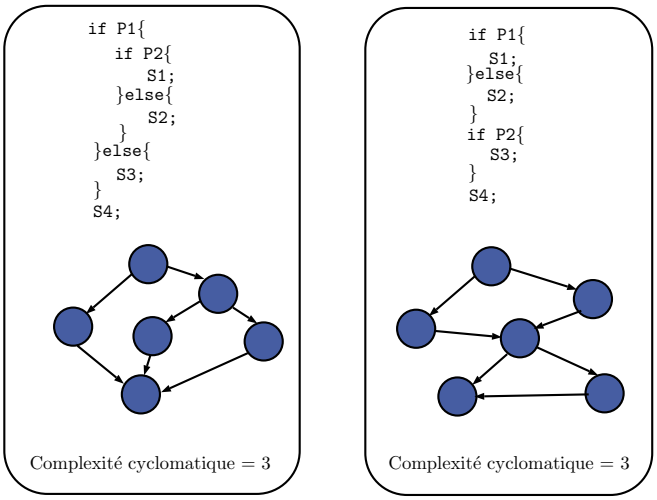


Figure. 10.2 – Exemple de la complexité cyclomatique

En réalité, comme nous l'avons vu précédemment, la méthode d'Halstead était axée sur la complexité de blocs individuels. Cependant, cette simple analyse ne suffit à pas expliquer dans son intégralité la complexité d'un programme. De même, le nombre cyclomatique ne tient pas compte de la magnitude du programme et compte uniquement les nombres de chemins basiques, comme illustré dans la figure 10.2.

Dans [HM81], ils ont proposé de prendre en compte à la fois les mesures de Halstead et les niveaux d'imbrication. Avant de présenter leur méthode, nous rappelons quelques définitions sur les graphes.

Rappels sur les graphes

Définition 10.2.5
On dit qu'un noeud est de sélection si son arité est supérieure ou égale à deux.

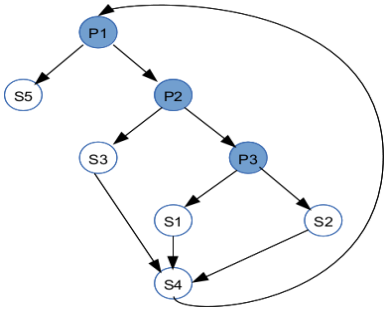
Harrison et Magel définissent une relation entre le noeuds qui n'est pas une relation d'ordre sur un ensemble ordonné.

Définition 10.2.6 (Relations entre les noeuds)
Un noeud x précède un noeud y s'il existe un chemin de x à y. Cette relation est notée $x < y$. S'il y a un arc de x à y, alors on dit que x précède immédiatement y et on écrit $x << y$.

Définition 10.2.7 (Bornes inférieure et supérieure)
Soit G' un sous-graphe d'un graphe e flot de contrôle G. Un élément m dans G est une borne supérieure pour G' si pour tout x élément dans G', la relation $m < x$ est vérifiée. À l'inverse, n est une borne inférieure pour G' si pour tout x élément dans G', la relation $x < n$ est vérifiée.

Mesure

La complexité est calculée en associant à chaque noeud une valeur qui correspond à la mesure d'Halstead de ce dernier. De plus, chaque noeud possède également une mesure de complexité additionnelle, la complexité ajustée. Le graphe suivant sera utilisé afin de montrer comment la méthode fonctionne.



La complexité ajustée est calculée de la manière suivante :

- Pour chaque noeud de sélection (arité $\geqslant 2$), on définit le sous-graphe G' comme l'ensemble des noeuds situés entre ce noeud, et la plus grande borne inférieure du sous-graphe formé de tous les noeuds qui succèdent immédiatement au noeud de sélection. Par exemple, pour notre graphe nous avons :
  - Le sous-graphe G' pour $P1$ contient les noeuds $S5$,$P2$,$S3$,$P3$,$S1$,$S2$,$S4$.
  - Le sous-graphe G' pour $P2$ contient les noeuds $S3$,$P3$,$S1$,$S2$.
  - Le sous-graphe G' pour $P3$ contient les noeuds $S1$,$S2$.
- La complexité ajustée est calculée en sommant la complexité d'Halstead de chaque noeud du sous-graphe G' et en ajoutant celle du noeud de sélection lui même.
- Pour tous les autres noeuds, la complexité ajustée correspond à la complexité de Halstead des instructions appartenant au noeud.

10.2.3 Points de croisement

Woodward, Hennell et Hedley [WHH79] ont proposé une mesure de complexité basée sur le comptage des points de croisement (knot) dans un programme. À l'origine

le langage de programmation ciblé était le `Fortran` mais il est possible d'appliquer leur méthode à d'autres langages de programme structuré.

Définition 10.2.8
Si un saut de la ligne a vers la ligne b est représenté par une paire ordonnée d'entiers (a, b), alors un saut (p, q) crée un point de croisement si l'une des situations apparaît :

- $\min(a,b) < \min(p,q) < \max(a,b)$ *et* $\max(p,q) > \max(a,b)$,
- $\min(a,b) < \max(p,q) < \max(a,b)$ *et* $\min(p,q) < \min(a,b)$.

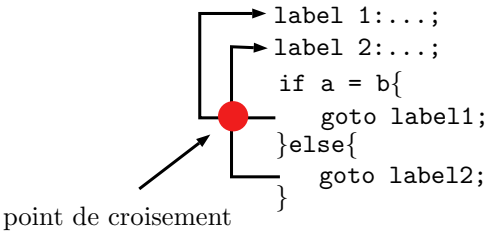


Figure. 10.3 – Point de croisement

Si l'on souhaite utiliser la définition directement avec les numéros des lignes, il faut bien vérifier que le programme est écrit correctement. Cependant, il est également possible d'utiliser le CFG du programme. Les paires ordonnées (a, b) et (p, q) sont alors des arcs dans le graphe.

10.3 Flot de données

Dans cette section, nous présentons différentes métriques liées aux données d'un programme. Nous nous intéresserons particulièrement aux flots de données et aux pointeurs.

10.3.1 Fan-in/Fan-out

Dans [HK81], les auteurs définissent la complexité d'un programme avec deux facteurs, la complexité d'une fonction et celle de l'interaction avec son environnement. Leur métrique utilise la notion de flot d'information.

Définition 10.3.1
Il y a flot global d'information de la fonction f vers la fonction g à travers une structure de données D, si f stocke de l'information dans D et que g en récupère de D.
Il y a un flot local d'information de la fonction f vers la fonction g si une ou plusieurs des conditions suivantes sont vérifiées :

(a) si f appelle g,

(b) si g appelle f et que f retourne une valeur qui sera utilisée ultérieurement par g,

(c) si h appelle à la fois f et g en passant la valeur de retour de f à g.

Un flot d'information local est direct si (a) est vérifiée et est indirect si (b) ou (c) est vérifiée.

Il est donc possible de construire la structure complète de flot d'un programme en utilisant une simple analyse des fonctions.

Deux notions sont introduites, le fan_{in} et le fan_{out} d'une fonction.

Définition 10.3.2
Le fan_{in} d'une fonction f est le nombre de flot local depuis f plus le nombre de structures de données à partir desquelles f récupère de l'information.
Le fan_{out} d'une fonction f est le nombre de flot local depuis f plus le nombre de structures de données que f met à jour.

À partir de ces définitions, les auteurs déduisent la métrique suivante :

$$length * (fan_{in} * fan_{out})^2.$$

Le produit $(fan_{in} * fan_{out})$ représente le nombre total de combinaisons possibles entre les données d'entrée et de sortie.

10.3.2 Découpage de programme

Dans [OT93], les auteurs ont développé plusieurs mesures de complexité d'après l'étude des découpes du programme.

Définition 10.3.3
Une découpe d'une fonction au niveau d'une instruction s par rapport à la variable v est l'ensemble des instructions et prédicats qui pourraient affecter la valeur de v au niveau de s. Les découpes peuvent être calculées en utilisant une analyse de flot de données ou la représentation par graphe de dépendance. Les découpes intra-procédurales sont restreintes à une fonction alors que les découpes inter-procédurales prennent en compte les appels de fonction.

On notera :

- V_f l'ensemble des variables utilisées par une fonction f et V_O le sous-ensemble contenant uniquement les variables de sorties de f,
- $SL(v)$ la découpe de retour, c'est-à-dire la découpe obtenue pour $v \in V_O$,

- $SL(V_O)$ l'intersection de tous les $SL(v)$, c'est-à-dire

$$SL(V_O) = \bigcap_{v \in V_O} SL(v),$$

- $\lambda(f)$ la longueur de la fonction f qui correspond au nombre d'instructions dans f hormis les déclarations et les instructions qui ne contiennent pas de variables.

Les métriques introduites sont les suivantes :

Coverage est une comparaison de la taille des découpes et de la longueur de la fonction.

$$Coverage(f) = \frac{1}{|V_O|} \sum_{v \in V_O} \frac{|SL(v)|}{\lambda(f)}.$$

Overlap est baséee sur le nombre d'instructions que les découpes ont en commun.

$$Overlap(f) = \frac{1}{|V_O|} \sum_{v \in V_O} \frac{|SL(V_O)|}{|SL(v)|}.$$

Tightness est basée sur le nombre d'instructions présentent dans toutes les découpes.

$$Tightness(f) = \frac{|SL(V_O)|}{\lambda(f)}.$$

Parallelism indique le nombre de découpe les plus distinctes des autres découpes.

$$Parallelism(f) = |\{SL(v) \text{ tel que } |SL(v) \cap SL(v')| \leqslant \tau \text{ pour tout } v \neq v'\}|.$$

MinCoverage est un ratio entre la longueur de la plus petite découpe dans une fonction et la longueur de cette dernière.

$$\frac{1}{\lambda(f)} \min_{v \in V_O} |SL(v)|.$$

MaxCoverage est un ratio entre la longueur de la plus grande découpe dans une fonction et la longueur de cette dernière.

$$\frac{1}{\lambda(f)} \max_{v \in V_O} |SL(v)|.$$

Métriques adaptées à l'obscurcissement

Dans [MDT07], les auteurs ont dérivé des métriques liées aux découpes dans le contexte de l'obscurcissement. Elles sont basées sur les points orphelins.

Définition 10.3.4
En reprenant les notions liées au découpage définies précédemment, les points qui restent après une découpe sont dits orphelins. Pour chaque $v \in V_O$, on définit :

$$res(f, v) = f \setminus SL(v).$$

Le résidu d'une découpe est défini comme l'ensemble des points orphelins.

Soit $res(f)$ l'union de tous les résidus. On a :

$$res(f) = \cup_{v \in V_O} res(f, v) = f \setminus SL(V_O).$$

Ils ont définis quatre nouvelles métriques qui s'inspirent de celles précédemment introduites.

Compactness calcule le rapport entre le nombre total de points orphelins et la taille de la fonction.

$$C(f) = \frac{|res(f)|}{\lambda(f)}.$$

MinDensity est le ratio du plus petit résidu par rapport à la taille de la fonction.

$$MinD(f) = \frac{1}{\lambda(f)} \min_{v \in V_0} |res(f, v)|.$$

Density compare la taille moyenne des résidus à celle de la fonction.

$$D(f) = \frac{1}{|V_O|} \sum_{v \in V_O} \frac{|res(f, v)|}{\lambda(f)}.$$

MaxDensity est le ratio du plus grand résidu par rapport à la taille de la fonction.

$$MaxD(f) = \frac{1}{\lambda(f)} \max_{v \in V_0} |res(f, v)|.$$

Ces métriques sont liées aux précédentes par :

$$\begin{aligned} C(f) &= 1 - Tightness(f), \\ MinD(f) &= 1 - MaxCoverage(f), \\ D(f) &= 1 - Coverage(f), \\ MaxD(M) &= 1 - MinCoverage(f). \end{aligned}$$

10.4 Complexité des pointeurs

Dans cette section, nous allons discuter de la complexité liée à l'utilisation de pointeurs généraux ou de fonctions. Tout d'abord, nous rappelons les résultats de Ramalingam sur la complexité des alias [Ram94].

Théorème 10.4.1 ([Ram94, Théorème 2.3])
Le problème intra-procédural concernant les propriétés may-alias(must-alias) est indécidable pour les langages munis d'instructions if, de boucles, d'allocation dynamique et de structures de données récursives.

Ramalingam a prouvé ce théorème en utilisant l'indécidabilité du PCP [HU79].

Définition 10.4.2
Le problème de Post-correspondance (PCP) est le suivant : Soient A et B deux listes quelconques de r chaînes de caractères dans $\{0,1\}^+$,

$$A = w_1, w_2, \ldots, w_r,$$

$$B = z_1, z_2, \ldots, z_r,$$

est-ce qu'il existe une séquence non vide d'entiers $i_1, i_2, \ldots, i_k$ *telle que*

$$w_{i_1}, w_{i_2}, \ldots, w_{i_k} = z_{i_1}, z_{i_2}, \ldots, z_{i_k}.$$

Nous allons maintenant analyser la complexité liée à l'utilisation de pointeurs généraux et celle liée aux pointeurs de fonction. Dans [WHKD00], les résultats sur les pointeurs généraux sont prouvés en utilisant une réduction avec un problème 3-SAT.

Théorème 10.4.3 ([WHKD00, Théorème 1])
En présence de pointeurs généraux, le problème de détermination des cibles de branches indirects est NP-difficile.

Dans [OSSM03], les auteurs se sont intéressés au cas des pointeurs de fonctions en utilisant également une réduction à un problème 3-SAT. Nous rappelons la preuve afin de voir comment la complexité se détermine sur un programme simple en présence de pointeurs de fonctions.

Théorème 10.4.4 ([OSSM03, Théorème 1])
Considérons que l'on est en présence d'affectation de pointeurs de fonction à partir d'un tableau de pointeurs de fonction, et d'appels de fonctions via des pointeurs qui retournent des entiers. Le problème qui consiste à déterminer si il existe une chemin d'exécution du programme pour lequel un pointeur de fonction donné pointe vers une fonction donnée à un point p du programme est NP-difficile.

Démonstration. La preuve s'obtient par une réduction polynomiale vers un problème 3-SAT. Considérons le problème 3-SAT avec les variables propositionnelles $v_1, v_2, \ldots, v_m$ pour lesquelles les valeurs peuvent être soit vrai soit faux. Soit la formule

$$F = \wedge_{i=1}^{n}(l_{i1} \vee l_{i2} \vee l_{i3}).$$

l_{ij} est un littéral qui correspond soit à v_k, soit à $\overline{v_k}$. La réduction se base sur le programme suivant.

```
int true() {return 1;}
int false() {return 0;}
main
L1 : int (*f_p)(),(*v_1)(),(*v_1_bar)(),...,(*v_n)(),(*v_m_bar)();
     int (*A[2])();
L2 : A[0] = false; A[1] = true;
L3 : si(-){v_1 = true;v_1_bar = false}else{v_1 = false;v_1_bar = true}
     si(-){v_2 = true;v_2_bar = false}else{v_2 = false;v_2_bar = true}
     ...
     si(-){v_m = true;v_m_bar = false}else{v_m = false;v_m_bar = true}
L4 : si(-) f_p = l_1,1; else si(-) f_p = l_1,2; else f_p = l_1,3;
     si(-) f_p = A[(f_p()&&l_2,1()];
           else si (-) f_p = A[(f_p()&&l_2,2()];
                       else f_p = A[(f_p()&&l_2,3()];
     ...
     si(-) f_p = A[(f_p()&&l_n,1()];
           else si (-) f_p = A[(f_p()&&l_n,2()];
                       else f_p = A[(f_p()&&l_n,3()];
L5 :
```

- $L1$: Déclaration des pointeurs de fonction $v_1, \ldots, v_n$ qui correspondent aux variables propositionnelles variables v_i du problème 3-SAT. Les pointeurs de fonction $\overline{v_1}, \ldots, \overline{v_n}$ correspondent à la négation de ces variables.
- l_{ij} dans le programme correspond au j^{e} littéral de la i^{e} clause dans la formule F.
- $L2$: $A[0]$ est initialisé avec l'adresse de fonction `true` et $A[1]$ pointe vers l'adresse de la fonction `false`.
- A est un chemin entre $L3$ et $L4$ qui représente une assignation à vrai des variables propositionnelles. Et l'inverse est également vrai.

Considérons que le problème 3-SAT a une solution. Par conséquent, toute clause a au moins un littéral vrai. Donc la variable correspondante dans le programme pointe vers la fonction qui retourne vrai. Ainsi, il existe un chemin exécution du programme pour lequel le pointeur de fonction f_p pointe vers la fonction `true` au niveau de $L5$. De même, si f_p pointe vers la fonction `true` au niveau de $L5$, alors il y a une solution au problème 3-SAT.

Maintenant, si le problème 3-SAT n'a pas de solution, alors il existe une clause pour laquelle tous les littéraux sont faux. Donc, le pointeur de fonction f_p pointe vers la fonction `false` au niveau de $L5$.

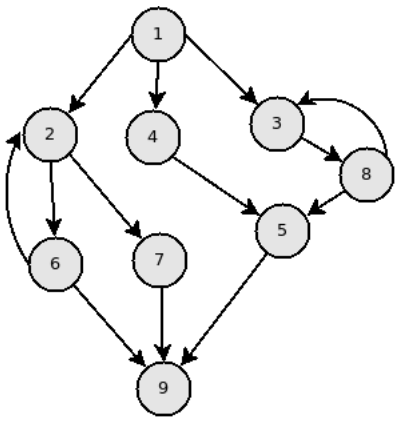
En conclusion, le problème 3-SAT possède une solution si et seulement si il est possible de déterminer un chemin d'exécution pour lequel le pointeur de fonction f_p pointe vers la fonction `true` au niveau de $L5$. □

10.5 Analyse de graphe : Isomorphismes et Automorphismes

Nous avons vu précédemment différentes métriques basées sur le graphe de flot de contrôle d'un programme. Il est possible d'utiliser d'autres caractéristiques d'un graphe comme ses automorphismes afin de déterminer la complexité de celui-ci. Avant de présenter les métriques, nous rappelons tout d'abord certaines notions liées aux graphes.

Définition 10.5.1
Un graphe G peut être représenté par sa matrice binaire d'adjacence $Adj(G)$. Si l'on considère que le noeuds sont indicés de 1 à n, alors $Adj(G)$ est une matrice $n \times n$ telle que $a_{i,j} = 1$ si et seulement si $(i, j) \in G$ et 0 sinon.

Exemple 10.5.2
Voici un graphe de flot de contrôle et sa matrice d'adjacence.



$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Définition 10.5.3
Soient G et H deux graphes. Une application ϕ de $V(G)$ dans $V(H)$ est un morphisme de graphe de G vers H si elle vérifie

$$(u,v) \in E(G) \Rightarrow (\phi(u),\phi(v)) \in E(H).$$

Définition 10.5.4
Soient G et H deux graphes. On dit que G et H sont isomorphes s'il existe un morphisme de graphe ϕ vérifiant :

- *ϕ est une bijection entre $V(G)$ et $V(H)$,*
- *$(u,v) \in E(G) \iff (\phi(u),\phi(v)) \in E(H)$.*

Si ϕ est un isomorphisme de G dans lui même, on dit que ϕ est un automorphisme. L'ensemble des automorphismes de G est appelé groupe d'automorphismes noté $Aut(G)$.

Intérêts

- La taille du groupe d'automorphismes donne une mesure directe de l'abondance de symétries dans un graphe. Par conséquent , il pourrait être intéressant de mesurer la complexité d'un code à partir de celle du groupe d'automorphismes du graphe associé à celui-ci.
- Nous pouvons analyser les isomorphismes possibles entre les sous-graphes ou le graphe dans son intégralité et regarder si une transformation de code supprime ces isomorphismes.
- De plus, mesurer la complexité de graphe « irréductible », c'est-à-dire de graphe caractéristique pour le code peut nous aider à déduire une approximation de la complexité globale.

10.5.1 Complexité de Kolmogorov

Dans [ZSTDL13], les auteurs utilisent la complexité de Kolmogorov pour analyser celle d'un graphe. Un lien est également fait avec le groupe d'automorphisme.

Définition 10.5.5
La complexité de Kolmogorov d'une chaîne de caractères s est la taille du plus petit programme P qui retourne s, quand il est exécuté sur une machine de Turing universelle U,

$$K_U(s) = \min\{|P|, U(P) = s\}.$$

Il n'existe pas d'algorithme effectif qui prend en entrée une chaîne s et qui retourne l'entier $K_U(s)$. Cependant, il y a des méthodes qui permettent de donner une approximation de la complexité de Kolmogorov, en particulier une qui utilise la mesure de probabilité algorithmique.

Théorème 10.5.6 (Théorème de Codage [Lev74])
Soit s une chaîne de caractère. Si on définit

$$m(s) = \sum_{P:U(P)=s} 1/2^{|P|},$$

alors pour c une constante fixée indépendante de s,

$$|\log_2 m(s) - K(s)| < c.$$

Dans [DZ12], une technique a été proposée afin de donner une approximation de $m(s)$ en utilisant une fonction qui considère toutes les machines de Turing de taille croissante. Soit (n,k) l'ensemble les machines de Turing à n états et k symboles en utilisant le formalisme de Busy Beaver [Rad62] et soit T une machine de Turing dans (n,k) sans entrée. On définit la notion suivante :

$$D(n,k) = \frac{|\{T \in (n,k) : T \text{ renvoie } s\}|}{|\{T \in (n,k) : T \text{ s'arrête}\}|}. \tag{10.1}$$

La définition 10.5.5 est donnée pour des chaînes de caractères mais on peut également l'appliquer au graphes. Pour ce faire, on utilise la représentation par matrice d'adjacence et on y applique la méthode de décomposition par blocs.

Méthode de décomposition par blocs

La méthode de décomposition par blocs (BDM) a été proposée dans [ZSTDG12]. Elle est utilisée afin de donner une approximation de la complexité de Kolmogorov d'objets de dimension d. Le principe est de décomposer des objets complexes en de plus petits pour lesquels nous avons déjà une estimation de complexité. Par la suite, la complexité de l'objet est obtenue en additionnant les complexités de chaque partie selon la règle de la théorie de l'information.

Dans notre cas, la matrice d'adjacence est un objet à deux dimensions. Une conséquence du théorème 10.5.6 permet donc de déduire que la complexité de Kolmogorov de la matrice d'adjacence d'un graphe peut être estimée en utilisant la fréquence qui a

été produite en faisant tourner des programmes aléatoires sur une machine de Turing de dimension deux.

Dans [ZSTDL13], les auteurs ont décidé de prendre $n = 5$ et $k = 2$ pour l'équation 10.1. Leur choix s'explique par le fait que les machines de Turing en dimension 2 avec quatre états et sans entrée produisent toutes les matrices carrées de taille 3 mais pas toutes celles de taille 4. Étant donné que l'approximation de K est meilleure si l'on considère de grandes matrices, prendre $n = 5$ permet de produire des matrices de taille 4 et cela s'exécute en temps raisonnable. Les auteurs n'excluent pas que d'autres valeurs pourraient être prises.

Définition 10.5.7
Soit $D(5,2)$ la distribution de fréquence construite à partir des machines de dimension deux en utilisant l'équation 10.1. Pour un vecteur u, on a

$$K_m(u) = -\log_2(D(5,2)(u)), \tag{10.2}$$

où $K_m(u)$ est une approximation de K au sens du théorème 10.5.6.

En appliquant la méthode de décomposition par bloc, on obtient donc la définition suivante.

Définition 10.5.8
Soient G un graphe et $Adj(G)$ sa matrice d'adjacence. Une approximation de complexité de Kolmogorov de G, $Klog_m(G)$,est définie comme :

$$Klog_m(G) = \sum_{(u,n_u)\in Adj(G)_{d\times d}} log_2(n_u) + K_m(u), \tag{10.3}$$

où $Adj(G)_{d\times d}$ représente l'ensemble des éléments (u, n_u) obtenus en décomposant la matrice d'adjacence de G en des matrices de taille $d \times d$ qui ne se chevauchent pas. u est donc une telle matrice carrée et n_u sa multiplicité.

L'article [ZSTDL13] met en lumière que si l'on souhaite avoir une mesure de complexité plus adéquate, il est préférable de considérer la complexité de Kolmogorov d'un graphe comme la plus petite valeur de $Klog_m$ pour toutes les permutations de la matrice d'adjacence.

De plus, afin de comparer la complexité de matrices de taille différente, ils ont également introduit la version normalisée de la méthode de décomposition par blocs. De cette manière, la taille de la matrice ne domine pas la mesure de complexité.

Méthode de décomposition par blocs normalisée

Afin de déterminer cette complexité normalisée, il faut introduire les valeurs minimale et maximale de la BDM d'un graphe.

Définition 10.5.9

$$MinBDM(n)_{d\times d} = \log_2(\lfloor n/d \rfloor) + \min_{x\in M_d(\{0,1\})} Klog_m(x), \tag{10.4}$$

et

$$MaxBDM(n)_{d\times d} = \sum_{r\in M_d(\{0,1\}), f_{n,d}(r)>0} \log_2(f_{n,d}(r)) + Klog_m(r), \tag{10.5}$$

où

- $M_d(\{0,1\})$ *est l'ensemble des matrices binaires de taille* $d \times d$,
- *Pour* $n, d \in \mathbb{N}$, $f_{n,d}$ *est une fonction* $f_{n,d} : M_d(\{0,1\}) \to \mathbb{N}$ *qui vérifie les propriétés suivantes :*

$$\sum_{r\in M_d(\{0,1\})} f_{n,d}(r) = \lfloor n/d \rfloor^2, \tag{10.6}$$

$$\max_{r\in M_d(\{0,1\})} f_{n,d}(r) \leqslant 1 + \min_{r\in M_d(\{0,1\})} f_{n,d}(r), \tag{10.7}$$

$$Klog_m(r_i) > Klog_m(r_j) \Rightarrow f_{n,d}(r_i) \geqslant f_{n,d}(r_j). \tag{10.8}$$

Pour tout $n \in \mathbb{N}/\{0\}$, $MinBDM(n)_{d\times d}$ retourne la valeur minimale de l'équation 10.3 pour des matrices carrées de taille n.

La valeur $f_{n,d}(r)$ indique le nombre d'occurrences de $r \in M_d(\{0,1\})$ dans la décomposition en matrices carrées $d \times d$ de la matrice carrée la plus complexe de taille $n \times n$. Cette fonction doit être définie de manière à assurer un calcul rapide de $MaxBDM(n)_{d\times d}$.

Remarque 10.5.10
La matrice la plus complexe dans $M_d(\{0,1\})$, notée Mc, est définie comme étant celle qui maximise $Klog_m(G)$ obtenue en utilisant l'équation 10.3. De manière à augmenter la somme des termes de droite, le résultat de la décomposition de Mc en matrices $d \times d$, doit contenir le plus de matrices différentes possibles. De plus, pour augmenter la somme des termes de gauche, les répétitions doivent être distribuées de manière homogène entre ces matrices.

Remarque 10.5.11
La distribution des complexités de matrices carrées de taille $d \in \{3,4\}$ dans $D(5,2)$ implique que $maxBDM$ est la complexité maximale pour une matrice carrée de taille n. Une autre valeur de d peut donner d'autres résultats.

On peut maintenant définir la complexité de Kolmogorov normalisée d'un graphe.

Définition 10.5.12
Soit G un graphe avec n sommets, la complexité de Kolmogorov normalisée de G en utilisant d comme paramètre la décomposition de la matrice d'adjacence, notée $NBDM(G)_d$, est définie comme

$$NBDM(G)_d = \frac{Klog_m(G) - MinBDM(n)_{d\times d}}{maxBDM(n)_{d\times d} - minBDM(n)_{d\times d}}. \tag{10.9}$$

Cette complexité permet d'avoir une mesure qui dépend de la complexité relative d'un graphe par rapport aux autres graphes de même taille.

Les auteurs ont également étudié le lien avec les groupe d'automorphismes du graphe. L'intuition était qu'étant donné que la taille de $Aut(G)$ mesure la proportion de symétries dans G, la complexité de Kolmogorov devrait être liée à celle-ci. Les tests effectués montrent les résultats suivants :

- Certains graphes ont à la fois un petit groupe d'automorphismes et un petite mesure de Kolmogorov.
- D'autres graphes présentant un gros groupe d'automorphisme, ont une petite valeur de mesure de Kolmogorov (la majorité).

10.5.2 Mesure via la structure du groupe des automorphismes

Au lieu de considérer uniquement la taille du groupe d'automorphisme, il serait intéressant d'analyser la structure du groupe. Nous avons donc réalisé des tests sur différents CFGs de programmes obscurcis ou non. Nous avons utilisé la bibliothèque `GAP` [GAP] pour l'analyse des groupes d'automorphisme. Les résultats ont montré qu'en appliquant certaines transformations d'obscurcissement, le groupe d'automorphismes était à la fois très gros par rapport à la version d'origine mais la structure quoique simple à écrire était également très complexe.

Prenons par exemple, le cas d'une implantation de `SHA-3`. Nous avons analysé les groupes d'automorphisme du code d'origine `autg2` et d'une version obscurcie `autg1` avec `GAP`.

```
gap> autg1:=AutomorphismGroup(g1);
<permutation group with 49 generators>
gap> autg2:=AutomorphismGroup(g2);
<permutation group with 3 generators>
gap> StructureDescription(autg2);
```

```
"C2 x C2 x C2"
gap> Order(autg1);
562949953421312
gap> Sautg1:=CompositionSeries(autg1);
[ <permutation group of size 562949953421312 with 49 generators>,
   <permutation group of size 281474976710656 with 48 generators>,
   <permutation group of size 140737488355328 with 47 generators>,
   <permutation group of size 70368744177664 with 46 generators>,
   <permutation group of size 35184372088832 with 45 generators>,
   <permutation group of size 17592186044416 with 44 generators>,
   <permutation group of size 8796093022208 with 43 generators>,
   <permutation group of size 4398046511104 with 42 generators>,
   <permutation group of size 2199023255552 with 41 generators>,
   <permutation group of size 1099511627776 with 40 generators>,
   <permutation group of size 549755813888 with 39 generators>,
   <permutation group of size 274877906944 with 38 generators>,
   <permutation group of size 137438953472 with 37 generators>,
   <permutation group of size 68719476736 with 36 generators>,
<permutation group of size 34359738368 with
     35 generators>, <permutation group of size 17179869184 with 34
generators>,
   <permutation group of size 8589934592 with 33 generators>,
<permutation group of size 4294967296 with
     32 generators>, <permutation group of size 2147483648 with 31 generators>,
   <permutation group of size 1073741824 with 30 generators>,
<permutation group of size 536870912 with
     29 generators>, <permutation group of size 268435456 with 28 generators>,
   <permutation group of size 134217728 with 27 generators>,
<permutation group of size 67108864 with
     26 generators>, <permutation group of size 33554432 with 25 generators>,
   <permutation group of size 16777216 with 24 generators>,
<permutation group of size 8388608 with
     23 generators>, <permutation group of size 4194304 with 22 generators>,
   <permutation group of size 2097152 with 21 generators>,
<permutation group of size 1048576 with
     20 generators>, <permutation group of size 524288 with 19 generators>,
   <permutation group of size 262144 with 18 generators>, <permutation
group of size 131072 with
     17 generators>, <permutation group of size 65536 with 16 generators>,
   <permutation group of size 32768 with 15 generators>, <permutation
group of size 16384 with
```

```
     14 generators>, <permutation group of size 8192 with 13 generators>,
   <permutation group of size 4096 with 12 generators>, <permutation
group of size 2048 with 11 generators>
     , <permutation group of size 1024 with 10 generators>,
<permutation group of size 512 with
     9 generators>, <permutation group of size 256 with 8 generators>,
<permutation group of size 128 with
     7 generators>, <permutation group of size 64 with 6 generators>,
<permutation group of size 32 with
     5 generators>, <permutation group of size 16 with 4 generators>,
<permutation group of size 8 with
     3 generators>, <permutation group of size 4 with 2 generators>,
<permutation group of size 2 with
     1 generators>, Group(()) ]
gap> Dautg1:=DerivedSubgroup(autg1);
<permutation group of size 32 with 5 generators>
gap> StructureDescription(Dautg1);
"C2 x C2 x C2 x C2 x C2"
gap> Fg1:=FactorGroup(autg1,Dautg1);
<pc group of size 17592186044416 with 44 generators>
gap> StructureDescription(Fg1);
"C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x
C2 x C2 x C2 x C2 x C2 x C2 x C2 x \
C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x C2 x
C2 x C2 x C2 x C2 x C2 x C2 x C2 x C\
2 x C2"
```

`autg1` est un gros groupe par rapport `autg2`. De plus, on peut remarquer qu'il est très tordu et qu'on peut le réaliser comme produit semi-direct de `Dautg1` par `Fg1`. On peut donc observer que l'application de techniques d'obscurcissement a complexifié et agrandi le groupe d'automorphismes.

10.6 Conclusion

Nous avons présenté dans ce chapitre différentes métriques qui permettent d'analyser la complexité des programmes. Elles permettent d'évaluer la puissance des techniques d'obscurcissement que nous introduirons dans le chapitre suivant.

Ces mesures ont été développées pour estimer la complexité du programme du point

de vue de la difficulté de programmation ou de revue de code. Cependant, elles peuvent être également appliquer pour juger de la difficulté de compréhension pour un attaquant. Ainsi, certaines ont été adaptées au contexte des transformations d'obscurcissement comme dans le cas des mesures liées au découpage de fonction.

La majorité des mesures présentées ici s'appuie sur une analyse statique du graphe de contrôle des programmes. Certaines d'entre-elles prennent en compte également les structures des conditions de contrôle, leurs niveaux d'imbrication ainsi que les interactions avec le programme afin d'avoir une mesure plus proche de la complexité du programme.

Chapitre 11

Obscurcissement de code source

Sommaire

Nous allons nous intéresser dans ce chapitre aux techniques liées à l'obscurcissement de code source en se focalisant en particulier sur le langage C. Pour ce faire, nous utiliserons la taxonomie de [CTL97]. Ils classent ces techniques en fonction de l'information qui est ciblée, nous avons donc les transformations

- de la structure lexicale,
- des données,
- du flot de contrôle,
- préventives.

Pour chaque méthode présentée, nous donnerons une analyse de complexité en fonction des mesures introduites dans le chapitre 10.

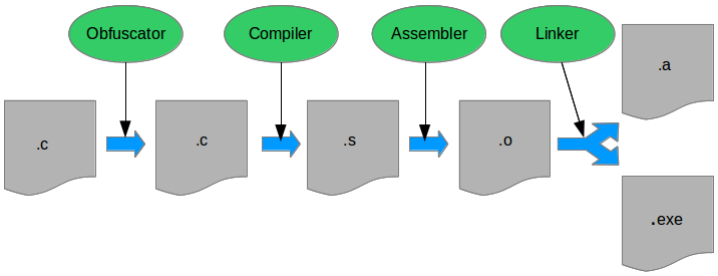


Figure. 11.1 – Obscurcissement de code source

Avant de présenter les différentes techniques que l'on peut trouver dans la littérature, nous allons tout d'abord revoir des définitions théoriques que l'on peut donner aux outils d'obscurcissement.

11.1 Définitions théoriques

Dans [CTL97], Collberg, Thomborson et Low ont donné une première définition d'une transformation d'obscurcissement de code source.

Définition 11.1.1
Soit $P \xrightarrow{\tau} P'$ une transformation de code source qui change P en P'. $P \xrightarrow{\tau} P'$ est une transformation d'obscurcissement, si P et P' ont le même comportement observable. Plus précisément, il faut que les deux conditions suivantes soient vérifiées :

- *Si P ne termine pas ou renvoie une erreur, alors il n'y a pas de contrainte sur la terminaison de P'.*
- *Sinon, P' doit se terminer et donner la même sortie que P.*

C'est une définition fonctionnelle qui n'inclut pas de critère de sécurité. La première à prendre compte cela a été donnée dans [Had00] et elle conduit à un résultat d'impossibilité. Dès lors différentes approches sont apparues afin de construire un outil d'obscurcissement que l'on pourrait appliquer à tout type de programme ou circuit.

11.1.1 Rappels

Dans la suite, nous allons considérer des ensembles probabilistes $D = \{D_n\}_{n\in\mathbb{N}}$ définis comme des séquences de variables aléatoires prenant leurs valeurs dans $\{0,1\}^{\ell(n)}$ avec $\ell : \mathbb{N} \to \mathbb{N}$ polynomiale en n. Ils sont dits constructibles en temps polynomial s'il existe une machine de Turing polynomiale probabiliste (PPTM) $\mathcal{M}$ telle que $D_n = \mathcal{M}(1^n)$.

Définition 11.1.2 (L'indistinguabilité calculatoire [GM82 ; Yao82])
Soit n le paramètre de sécurité. Deux ensembles probabilistes D et F sont indistinguables de manière calculatoire s'il existe une fonction négligeable ϵ tel que pour PPTM $\mathcal{M}$, qui reçoit 1^n et un échantillon s de D_n ou F_n, et qui renvoie 0 ou 1, pour des n assez grand, l'inéquation suivante est vérifiée ;

$$|Pr_{s\sim D_n}[\mathcal{M}(1^n, s) = 1] - Pr_{s\sim F_n}[\mathcal{M}(1^n, s) = 1]| \leqslant \epsilon(n).$$

La notation $s \sim D_n$ représente l'expérience de choisir $s \in X$ avec la distribution D_n.

Dans les définitions suivantes, $\mathcal{C}$ désignera une famille de circuits probabilistes de taille polynomiales et pour un paramètre d'entrée n, on notera $\mathcal{C}_n$ les circuits de $\mathcal{C}$ avec n comme taille d'entrée.

11.1.2 Obscurcissement en « boîte noire virtuelle »

Barak et al [Bar+01 ; Bar+12] ont donné une définition un peu moins restrictive que celle de [Had00] en utilisant le principe de « boîte noire virtuelle » mais ont également prouvé l'impossibilité d'existence d'un tel outil.

Définition 11.1.3
Un outil d'obscurcissement est un « compilateur » (efficace, probabiliste) qui prend en entrée un programme (ou un circuit) P et qui produit un nouveau programme $\mathcal{O}(P)$ qui satisfait les deux conditions suivantes :

- ***Fonctionnalité** : $\mathcal{O}(P)$ calcule la même fonction que P.*
- ***Propriété de « Boîte noire virtuelle »** : Tout ce qui peut être calculé de manière efficace à partir de $O(P)$ peut l'être également si l'on a un oracle lié à P.*

Il est possible de définir de manière plus précise ce que la seconde propriété implique pour un attaquant. En ordre décroissant de généralité nous avons :

- **L'indistinguabilité calculatoire** : On ne fait aucune restriction sur ce que l'attaquant essaie de calculer. Il est donc demander qu'il soit possible, en ayant juste un accès à un oracle de P, de produire une distribution de sortie qui est indistinguable de manière calculatoire de ce que l'attaquant calcule avec $\mathcal{O}(P)$.
- **La satisfaction d'une relation** : On peut se restreindre au fait que l'attaquant essaie de produire une sortie qui satisfasse une relation pré-déterminée (peut-être en temps polynomial) avec le programme d'origine P. On demande donc qu'il soit possible avec juste un accès à un oracle de P, de réussir avec à peu près la même probabilité qu'un attaquant avec $\mathcal{O}(P)$.
- **Calcul d'une fonction** : On peut également ne considérer que les relations qui sont fonctions. Dans ce cas, l'attaquant essaie de calculer une fonction pré-déterminée du programme d'origine.
- **Calcul d'un prédicat** : La dernière alternative restreint les fonctions aux fonctions à valeur binaire. L'attaquant essaie donc de décider d'une propriété sur le programme d'origine.

Pour les deux premiers cas, Barak et al [Bar+01 ; Bar+12] donnent une preuve d'impossibilité en utilisant une relation quelconque R entre deux programmes. Elle vérifie si deux programmes P et P' s'accordent sur plusieurs entrées choisies aléatoirement (dans $\{0,1\}^k$ où k est le paramètre de sécurité). En utilisant $P' = \mathcal{O}(P)$, un attaquant peut donc satisfaire la relation R. Or, il n'est pas possible de satisfaire R en ayant juste un accès à un oracle de P, si P est un programme qui est difficile à comprendre à partir des requêtes. Ils donnent ainsi l'exemple de fonctions pseudo-aléatoires.

Définition 11.1.4
Un algorithme $\mathcal{O}$ qui prend en entrée un circuit dans $\mathcal{C}$ et renvoie un nouveau circuit, est un outil d'obscurcissement en « boîte noire » pour la famille $\mathcal{C}$, s'il a les propriétés suivantes :

- ***Préservation de fonctionnalité** : Pour toute entrée de taille n, pour tout $C \in \mathcal{C}_n$:*

$$Pr[\text{il existe } x \in \{0,1\}^n \text{ tel que } \mathcal{O}(C)(x) \neq C(x)] \leqslant neg(n).$$

- ***Ralentissement polynomial** : Il existe un polynôme p tel que pour toutes les tailles d'entrée n, pour tout $C \in \mathcal{C}_n$, la taille du circuit obscurci sera limitée par p, c'est-à-dire que l'inégalité suivante doit être vérifiée :*

$$|\mathcal{O}(C)| \leqslant p(|C|).$$

- ***Boîte noire virtuelle*** *: Pour tout circuit de taille polynomial $\mathcal{A}$ (attaquant), il existe un circuit de taille polynomial $\mathcal{S}$ (simulateur) tel que pour toute taille d'entrée n et pour tout $C \in \mathcal{C}_n$:*

$$|Pr[\mathcal{A}(\mathcal{O}(C)) = 1] - Pr[\mathcal{S}^C(1^n) = n]| \leqslant neg(n).$$

$\mathcal{O}$ est dit efficace s'il s'exécute en temps polynomial en la taille du circuit d'entrée.

Afin de prouver leur résultat d'impossibilité pour les deux autres cas, ils se sont restreints au cas le moins restrictif à savoir le calcul de prédicat. Le théorème suivant est démontré.

Théorème 11.1.5 ([Bar+01 ; Bar+12, Théorème 3.10])
Si les fonctions à sens unique existent, alors il existe un ensemble de circuits qui ne peuvent être obscurcis.

Ils ont réussi à construire une famille de circuits booléens $\mathcal{P}$ à partir d'une famille à sens unique quelconque qui ne pouvait pas être obscurcie au sens où (en utilisant la terminologie des programmes) :

- si l'on avait un programme P' qui calculait la même fonction que $P \in \mathcal{P}$, alors on pourrait reconstruite le « code source » de P' en temps quasi quadratique en taille de P ;
- de plus, en ayant accès à un oracle d'un programme $P \in \mathcal{P}$(choisi aléatoirement), aucun algorithme efficace ne peut reconstruire P (ou même distinguer un certain bit du code d'un bit aléatoire) sauf avec une probabilité négligeable.

Remarque 11.1.6
On obtient également un résultat d'impossibilité en considérant un outil d'obscurcissement « approximatif », c'est-à-dire que le circuit obscurci n'est pas obligé de calculer exactement la même fonction que le circuit d'origine (voir [Bar+01 ; Bar+12, Théorème 4.3]).

Ce résultat n'implique pas la non existence d'outils d'obscurcissement que l'on pourrait utiliser en pratique dans le domaine de la protection logicielle. Il est possible de remplacer dans la définition précédente la propriété de « boîte noire virtuelle » par une condition plus faible.

11.1.3 Obscurcissement indistinguable

Dans [Bar+01 ; Bar+12], la notion d'outil d'obscurcissement indistinguable est ainsi introduite en garantissant que si deux circuits calculent la même fonction, alors leurs

versions obscurcies sont indistinguables en temps polynomial probabiliste. Plus formellement, la définition suivante est donnée pour les circuits :

Définition 11.1.7
Un algorithme $\mathcal{O}$ qui prend en entrée un circuit dans $\mathcal{C}$ et qui renvoie un nouveau circuit, est un outil d'obscurcissement indistinguable pour la famille $\mathcal{C}$, s'il garantit les deux propriétés de préservation de fonctionnalité et de ralentissement polynomial données dans la définition 11.1.4 mais également la suivante (qui remplace la boîte noire virtuelle) :

- ***L'indistinguabilité** : Pour des tailles d'entrée suffisante, pour tout circuit $C_1 \in \mathcal{C}_n$ et pour tout $C_2 \in \mathcal{C}_n$ qui calcule la même fonction que C_1 et tel que $|C_1| = |C_2|$, les deux distributions $\mathcal{O}(C_1)$ et $\mathcal{O}(C_2)$ sont indistinguables.*

Elle leur permet de construire un tel outil assez facilement ([Bar+01 ; Bar+12, Proposition 7.2]) mais celui-ci n'est pas en temps polynomial en la taille du circuit d'entrée.

Cependant comme le souligne Goldwasser et Rothblum [GR07 ; GR14], cette définition ne donne pas de garantie sur le fait que le circuit obscurci « cache réellement de l'information ». Ils ont donc proposé une nouvelle interprétation sous le nom de « meilleur obscurcissement possible » (Best-possible obfuscation).

Définition 11.1.8
Un algorithme $\mathcal{O}$ qui prend en entrée un circuit dans $\mathcal{C}$ et qui renvoie un nouveau circuit, est un outil d'obscurcissement « le meilleur possible » pour la famille $\mathcal{C}$, s'il garantit les deux propriétés de préservation de fonctionnalité et de ralentissement polynomial données dans la définition 11.1.4 mais également la suivante (qui remplace la boîte noire virtuelle) :

- ***Meilleure possibilité** : Pour tout circuit $\mathcal{L}$ de taille polynomiale qui tente d'extraire de l'information d'un circuit obscurci, il existe un simulateur $\mathcal{S}$ de taille polynomiale tel que pour une taille d'entrée suffisante n, pour tout circuit $C_1 \in \mathcal{C}_n$ et pour tout $C_2 \in \mathcal{C}_n$ qui calcule la même fonction que C_1 et tel que $|C_1| = |C_2|$, les deux distributions $\mathcal{L}(\mathcal{O}(C_1))$ et $\mathcal{S}(C_2)$ sont indistinguables.*

Goldwasser et Rothblum [GR07] donnent également une famille de circuit (les diagrammes de décision binaires ordonnés polynomiaux) qui permet de passer le paradigme de la boîte noire en utilisant leur définition. De plus, l'outil d'obscurcissement est efficace pour celle-ci [GR07, Propositions 3.2 et 3.3].

La distinction avec la définition 11.1.7 est plus complexe. Ils ont montré que dans le cas d'outils d'obscurcissement efficaces, les deux définitions sont équivalentes [GR07, Propositions 3.4 et 3.5]. Cependant la question d'équivalence pour des outils inefficaces reste ouverte.

Remarque 11.1.9
Si la famille de circuits $\mathcal{C}$ admet des formes canoniques calculables de manière efficace, alors le calcul de cette forme serait déjà un outil d'obscurcissement indistinguable aux sens des deux définitions [Bar+01 ; Bar+12 ; GR07].

La première construction d'un outil d'obscurcissement indistinguable a été donnée par Garg, Gentry et Halevi [GGHRSW13]. Puis celle-ci a été améliorée par les travaux suivants [BR14 ; BGKPS14 ; AB15].

11.1.4 Obscurcissement à différentes entrées ou extractible

Une dernière définition a été introduite pour décrire un outil d'obscurcissement dans [Bar+01 ; Bar+12]. Pour tous circuits C_0 et C_1, un outil d'obscurcissement à différentes entrées $di\mathcal{O}$ garantit que la non-existence d'une attaque qui puisse trouver une entrée sur laquelle C_0 et C_1 diffèrent, implique que $di\mathcal{O}(C_0)$ et $di\mathcal{O}(C_1)$ sont indistinguables de manière calculatoire.

Dans [ABGSZ13], les auteurs ont utilisé cette notion afin de construire différentes applications. Ils ont étudié le cas des machines de Turing et en particulier avec le temps d'exécution comme donnée d'entrée. Ils ont également fourni un schéma de chiffrement et un protocole d'échange de clés multi-partie non interactif. De même, dans [BCP14] ils ont étudié d'autres applications comme par exemple un schéma de chiffrement fonctionnel avec témoin.

Cependant Garg et al ont donné un résultat d'impossibilité sur cette notion [GGHW14]. Leurs travaux montrent que l'existence d'un $di\mathcal{O}$ générique avec des entrées auxiliaires génériques implique qu'un circuit spécifique $C^\star$ avec une entrée spécifique $aux^\star$, qui ne peut être obscurci de manière à cacher une information spécifique existe.

En forçant les entrées auxiliaires utilisées à être des chaînes de caractères aléatoires publiques, Ishai, Pandey et Sahai ont donné une définition qui permet de passer outre les résultats d'impossibilité précédents [IPS15].

11.2 Modifications de la structure lexicale

Ce type de modifications n'influe pas sur les mesures de complexité mais elles peuvent avoir un impact sur la compréhension basique du programme par un humain en altérant la sémantique des variables et des fonctions du programme.

Il est par exemple possible de changer les noms des variables locales et globales, des fonctions, des données sensibles (remplacer `key` par `hzhzi`). Pour que cela soit efficace, il faut transformer tous les noms car si l'attaquant se rend compte que seules certaines variables ont des noms complètement aléatoires, il pourra facilement les identifier et se concentrer dessus. Ces transformations sont en coût linéaire en la taille du programme.

11.3 Obscurcissement des données

Nous allons traiter ici du cas des transformations sur les données, entiers, chaînes de caractères, tableaux et structures. Ce type de techniques ne changent pas le comportement d'un programme s'il l'on modifie également les fonctions utilisant les données obscurcies. Dans [Dra04], on peut trouver un formalisme lié à l'obscurcissement des données.

11.3.1 Encodage

L'encodage des valeurs entières et des chaînes de caractères constantes peut être réalisé de différentes manières. Il est possible de diviser la variable que l'on souhaite cacher en plusieurs morceaux qui sont dispersés à travers le programme [SS99]. On peut par exemple découper la chaîne `"Enter your password"` en `"Enter"`, `"your"`, `"password"`. Cependant, le problème de reconstruction se pose. En effet, il faudrait éviter de retrouver la donnée en clair lors de l'exécution.

Il est également possible d'utiliser un code qui génère la réelle valeur au cours de l'exécution [Mea55].

11.3.2 Transformations des tableaux

Les transformations des tableaux peuvent être de deux natures :

- il peut s'agir d'une réorganisation des éléments au sein du tableau,
- ou d'un restructuration de celui en le décomposant, en le fusionnant avec d'autres tableaux ou en changeant le nombre de dimension.

Il est possible d'utiliser une permutation σ afin de réagencer les éléments d'un tableau A pour le transformer en un tableau B. On a donc $A[i] = B[\sigma[i]]$.

Exemple 11.3.1
Dans notre exemple, on utilise une permutation simple qui consiste à renverser l'ordre des éléments.

A	10	2	17	51	12	9	12	82	36	75	46	13

B	13	46	75	36	82	12	9	12	51	17	2	10

À place d'une permutation, il est possible d'utiliser un isomorphisme. Dans [ZTW06], ils utilisent une fonction homomorphe

$$f(i) = (i \times m) \bmod n,$$

où n est la dimension du tableau et m un entier premier avec n.

Exemple 11.3.2
Dans notre exemple, nous avons pris $n = 9$ et $m = 11$.

A	10	2	17	51	12	9	12	82	36

B	10	9	2	12	17	82	51	36	12

Des techniques de restructuration de tableaux ont été développées dans [CTL97 ; CTL98a ; Dra06].

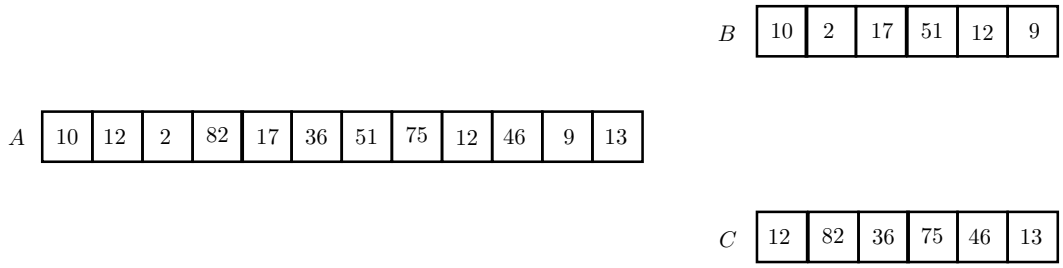
Découpage Le découpage de tableaux consiste à diviser un tableau A de taille n en plusieurs tableaux B_i de taille respective m_i. De manière théorique, il est nécessaire d'avoir une fonction *select* qui permet de déterminer dans quel tableau B_i un élément $A[j]$ doit aller. Il faut également des fonction p_i qui donnent la localisation de chaque élément dans les nouveaux tableaux.

$$\begin{aligned} select &: [0 \dots n) \to [0, 1] \\ p_i &: [0 \dots n) \to [0, m_i) \end{aligned}$$

La relation suivante est vérifiée :

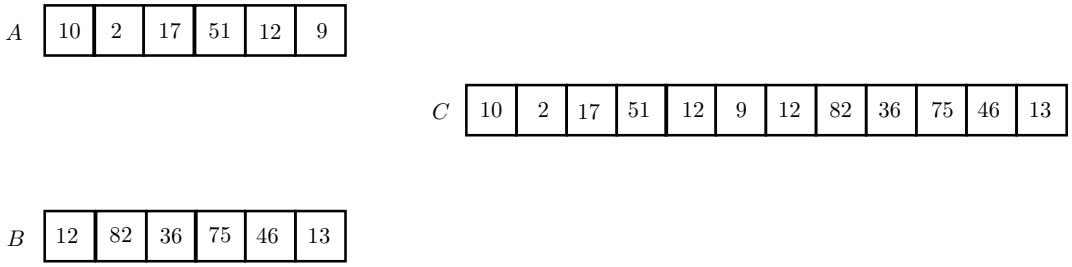
$$A[j] = B_{select(j)}[p_{select(j)}(j)].$$

Exemple 11.3.3
Le tableau A a été divisé en deux tableaux B et C en mettant les élément d'indice pair dans B et ceux d'indice impair dans C. L'ordre des éléments est conservé dans B et C.



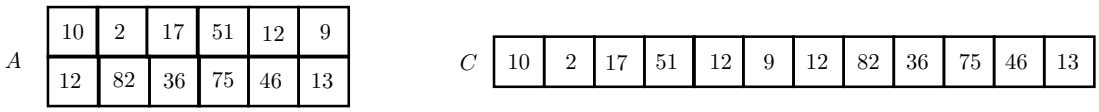
Fusion La fusion de tableaux est l'inverse du découpage. Elle consiste à réunir les éléments de plusieurs tableaux au sein d'un même tableau. Il faut comme précédemment des fonctions pour déterminer l'attribution des éléments.

Exemple 11.3.4
Les tableaux A et B ont réunis au sein du tableau C. La fusion est faite en mettant tout d'abord les éléments de A, puis ceux de B.



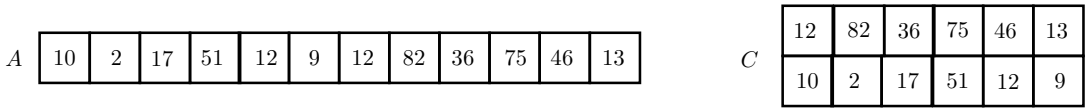
Aplatissement L'aplatissement d'un tableau à plusieurs dimensions consiste à le remplacer par un tableau avec moins de dimensions.

Exemple 11.3.5
Le tableau A qui a deux dimensions est transformé en un tableau C à une seule dimension en alignant les lignes de A.



Augmentation de la dimension Cette technique consiste à augmenter la dimension d'un tableau en réorganisant ces éléments.

Exemple 11.3.6
Le tableau A qui a une dimension est transformé en un tableau C à deux dimensions.



11.4 Obscurcissement du flot de contrôle

Dans cette section, nous allons présenter différentes techniques qui modifient le flot de contrôle d'un programme. Le comportement d'un programme obscurci est préservé si les conditions d'application des transformations sont bien respectées.

11.4.1 Prédicats opaques

Définition 11.4.1
Un prédicat est dit opaque si sa valeur est connue au moment de la transformation du programme mais qu'elle est difficile à déduire pour un outil de dé-obscurcissement. Un prédicat vrai (faux), qu'on notera P^T (P^F) est toujours évalué à vrai (faux). On peut également définir un prédicat $P^?$ dont la valeur est délibérément inconnue ou n'influe pas sur l'exécution du programme.

Certains prédicats peuvent se simplifier en utilisant une simple analyse statique locale. Par exemple, si l'on utilise

- des appels à des fonctions dont la signification est triviale,

```
if(random(1,5) < 0){ .. }
```

- des comparaisons arithmétiques simples sur des constantes,

```
int b = 9;
{... } pas de modifications de b
if(b < 5) { ... }
```

- certaines propriétés mathématiques faciles à évaluer,

```
if(((2*v) %2) == 0) { ... }
```

Il est possible de créer des prédicats arithmétiques plus complexes en utilisant des invariants ou bien des propriétés que l'on sait non évaluables pour des variables quelconques.

Exemple 11.4.2

- $(x * (x+1) * (x+2) \mod 2 == 0)$ est un prédicat toujours vrai.
- $(9 * x \mod 9 == 2)$ est prédicat toujours faux.
- $((x^2+1) \mod 3 == 1)$ est prédicat dont la valeur dépend de x.

Cependant, il faut considérer que si l'on peut créer de telles expressions à partir d'une base de propriétés mathématiques, un attaquant peut le faire aussi. De plus, une analyse abstraite peut également être utilisée pour retrouver la valeur du prédicat.

Une autre stratégie consiste à distribuer le calcul d'un prédicat sur plusieurs fonctions. L'attaquant est ainsi obliger de procéder à une analyse inter-procédurale afin d'analyser le prédicat.

Nous allons discuter d'une approche qui s'appuie sur la difficulté d'analyser des alias.

Utilisation de structure dynamique Dans [CTL98b], les auteurs proposent d'utiliser une structure dynamique afin de construire des prédicats opaques.

La méthode consiste à

- Ajouter au programme du code qui crée des structures complexes dynamiques $S_1, S_2, \ldots$.
- Garder un ensemble de pointeurs $p_1, p_2, \ldots$ dans ces structures.
- Le nouveau code ajouté doit occasionnellement mettre à jour les structures (en modifiant les pointeurs, en ajoutant des noeuds, en séparant et en réunissant des structures,etc). Cependant, il faut que ces transformations préservent des invariants comme par exemple « Il y a un chemin de p_1 à p_2 ».
- Utiliser ces invariants afin de construire des prédicats opaques.

En procédant de cette manière,

- le code introduit va ressembler au code d'origine quand l'usage de pointeur y est déjà important ; il sera donc **discret** et quasiment indétectable ;
- il est facile de construire des opérations de mise à jour destructives (suppression de noeuds) qu'une analyse de pile courante ne pourra pas traiter ; le code introduit sera donc **robuste** ;
- il est aisé de construire des invariants qui peuvent être évalués en temps constant ; le code introduit sera peu **coûteux**, il n'aura pas trop d'influence sur le temps d'exécution du programme obscurci.

La complexité de prédicats ainsi construits reposent donc sur un problème d'alias de pointeur que l'on sait NP-difficile.

Un choix de structures dynamiques complexes peut être d'utiliser des graphes. On peut par exemple construire

- des graphes aléatoires orientés,
- des graphes aléatoires non orientés (voir figure 11.2),
- des graphes aléatoires avec plusieurs composantes (voir figure 11.3),
- des graphes aléatoires non orientés avec des pivots (voir figure 11.4).

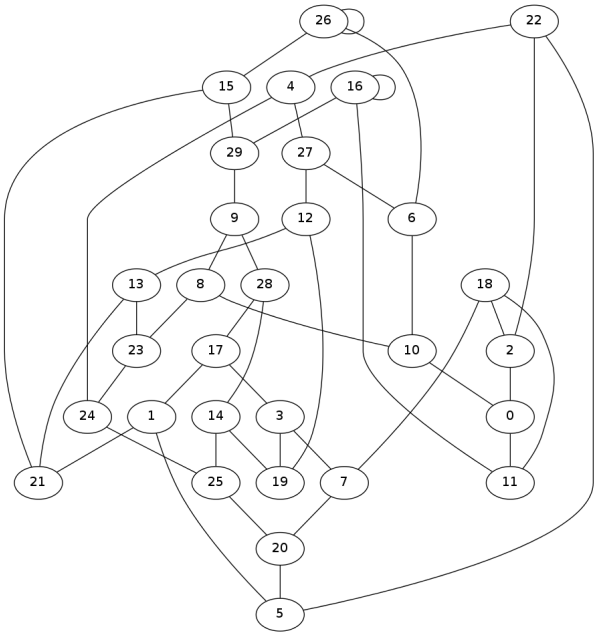


Figure. 11.2 – Graphe aléatoire non orienté

Une mise à jour d'un graphe avec pivots peut consister à supprimer des pivots.

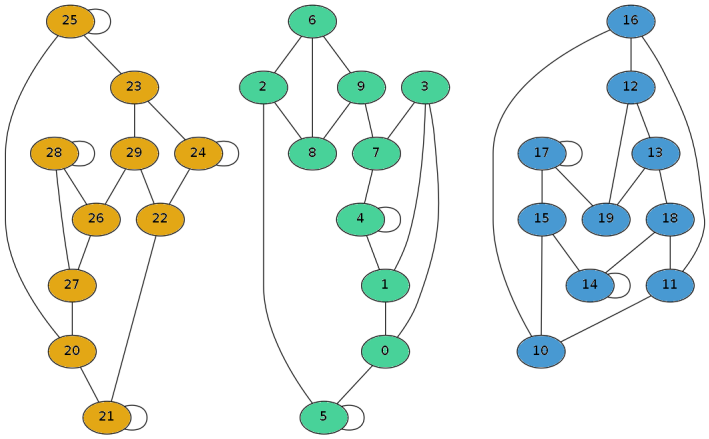


Figure. 11.3 – Graphe aléatoire non orienté avec plusieurs composantes connexes

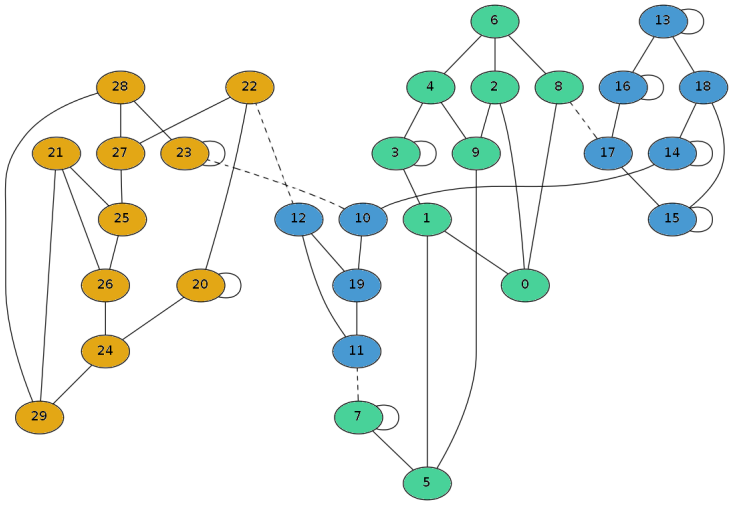


Figure. 11.4 – Graphe aléatoire non orienté avec des pivots

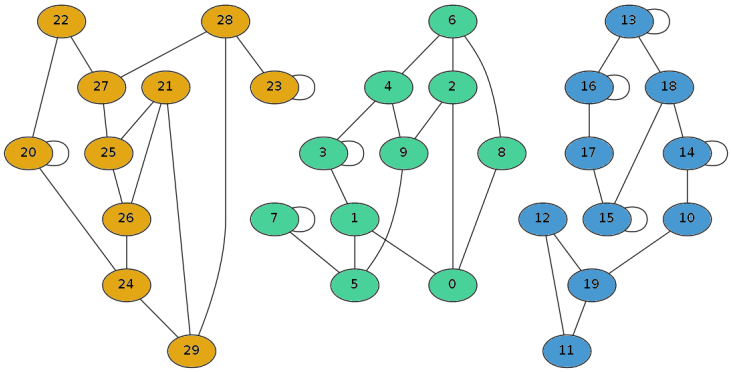
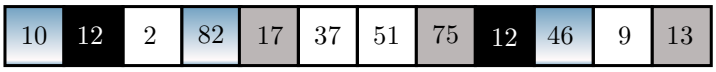


Figure. 11.5 – Graphe aléatoire non orienté après suppression des pivots

Utilisation de tableaux

Il est possible d'utiliser un tableau d'entiers comme base de construction de prédicats. Ce tableau est initialisé avec des valeurs aléatoires qui vérifient certains invariants. Puis, des mises à jour du tableau sont faites à l'intérieur du code afin de modifier les valeurs stockées tout en préservant les invariants.

Exemple 11.4.3
Prenons par exemple le tableau suivant :



Les invariants utilisés sont :

- Toutes les cases noires ont une même valeur.
- Toutes les cases blanches ont une valeur qui est congrue à 2 modulo 7.
- Toutes les cases grises ont une valeur qui est impaire.
- Toutes les cases en dégradé ont une valeur qui est congrue à 1 modulo 9.

Une autre construction consiste à utiliser des séquences de code parallélisées [CN09, section 4.4.3]

11.4.2 Aplatissement du flot de contrôle

L'aplatissement du flot de contrôle consiste à lui enlever la structure des fonctions, les niveaux d'imbrication ainsi que les structure conditionnelles [WHKD00 ; CGJZ01 ; LK07]. Chaque bloc basique se retrouve au même niveau que les autres. Le contrôle du flot se fait à l'aide d'un `switch` et chaque cas correspond à un bloc basique. La variable *switch_variable*, qui contrôle la structure, est mise à jour à la fin de chaque bloc afin d'indiquer le bloc suivant.

À l'issue de cette transformation la complexité de construction du CFG repose sur la détermination des cibles des sauts entre les différents blocs. Les cibles étant contrôlées par la variable *switch_variable*, la complexité repose sur un problème de flot données *use-n-def*. Dans le chapitre précédent, nous avons vu que dans ce contexte une simple analyse permet de reconstruire le CFG. Afin d'augmenter la complexité de construction, il faut donc utiliser une autre instance de problème de flot de données. On peut par exemple, introduire des alias non-triviaux dans le programme car on sait qu'en présence de pointeurs l'analyse est NP-difficile.

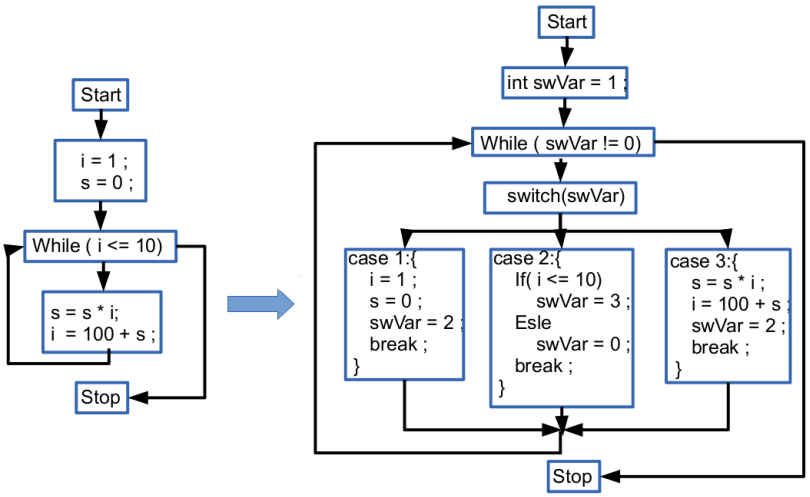


Figure. 11.6 – Aplatissement du flot de contrôle

Utilisation de pointeurs généraux

Dans [WHKD01], les auteurs ont proposé deux techniques utilisant des tableaux et des pointeurs.

Utilisation d'un tableau Cette méthode consiste à introduire un tableau global et la valeur de *switch_variable* est calculée à partir des éléments du tableau. Ce dernier peut par exemple être obscurci en utilisant des expressions complexes sur les indices ($f_i()$).

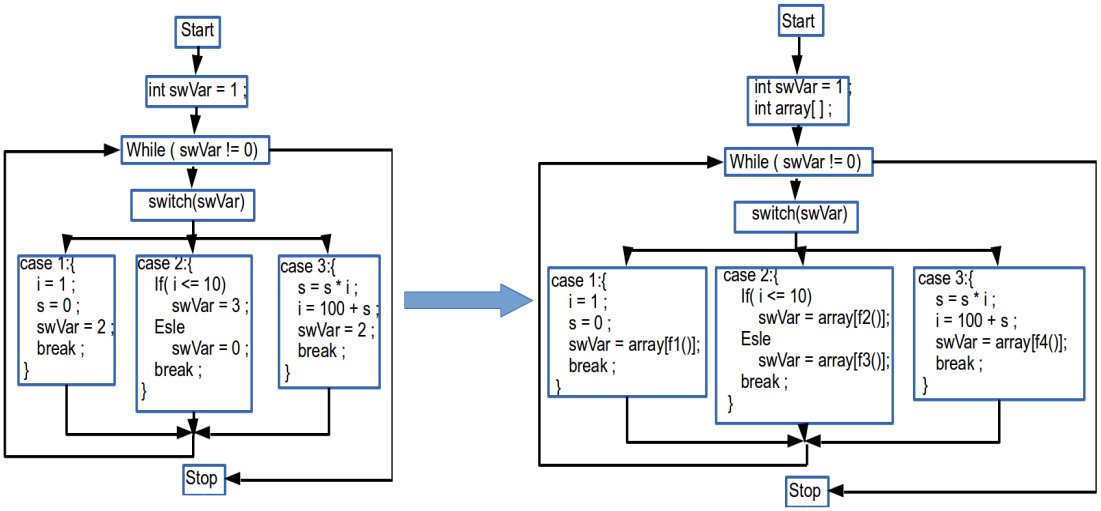


Figure. 11.7 – Utilisation d'un tableau pour l'aplatissement de code

Manipulation de pointeurs Cette méthode se met en place en trois étapes :

- Dans chaque fonction, un nombre arbitraire de pointeurs est introduit pour accéder à des variables déjà déclarées mais également aux éléments du tableau.
- Les références aux variables et aux éléments du tableau sont remplacées par des celles des pointeurs.
- Autant que possible, l'utilisation des pointeurs et leurs définitions sont placées dans différents blocs afin de rendre plus difficile une analyse *use-n-def*.

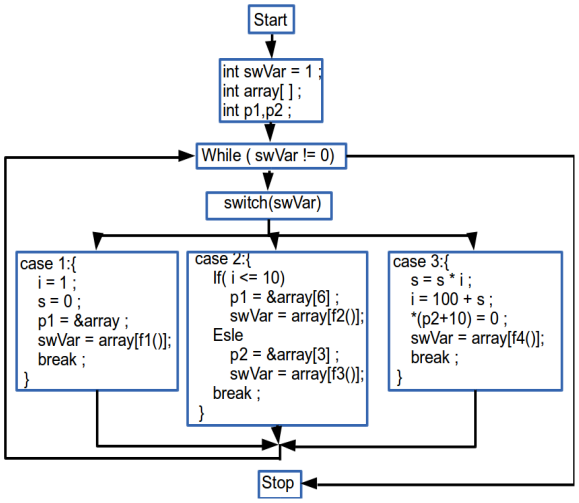


Figure. 11.8 – Manipulation de pointeur

Utilisation de fonction de transition complexe

Cappaert et Preneel [CP10] ont proposé une méthode d'aplatissement qui utilise des fonctions de transition complexe pour augmenter la complexité du CFG. Ils utilisent des fonctions à sens unique, comme par exemple des fonctions de hachage cryptographiques ou liées au logarithme discret.

11.4.3 Insertion de code mort ou non pertinent

Il est possible d'insérer du code non pertinent en utilisant un prédicat opaque. En effet, considérons une séquence S de blocs basiques $S_1, \ldots, S_n$. Nous découpons par exemple S en deux sous-séquences S^a et S^b. Nous pouvons utiliser un prédicat opaque P^T dans une structure conditionnelle `if` afin de contrôler l'exécution de la seconde séquence de

blocs. Il n'y a pas de `else` dans une telle construction. La figure 11.9 (a) illustre cette technique.

Nous pouvons rendre la transformation plus complexe en utilisant un prédicat $P^?$. Dans ce cas, nous dupliquons la séquence S^b en S'^b. Les deux sous-séquences peuvent être obscurcies séparément de manière à ce qu'un attaquant ne puisse pas déduire qu'elles réalisent la même fonction. La figure 11.9 (b) montre un exemple de cette méthode.

Finalement, un bug peut être introduit dans l'une des copies S^b. L'utilisation d'un prédicat P^T ou P^F permet de garantir que seule la versions qui est correcte sera exécutée. La figure 11.9 (c) donne un exemple.

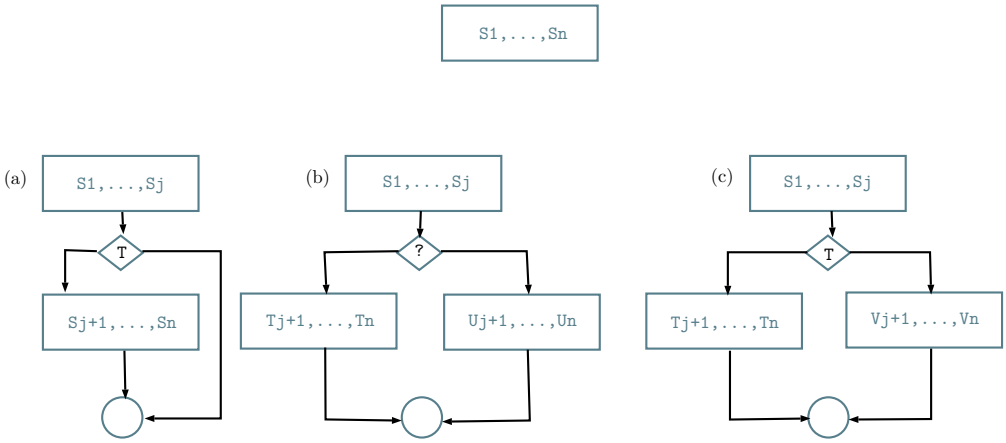


Figure. 11.9 – Insertion de code mort ou non-pertinent

La complexité de cette technique repose sur la difficulté à déduire la valeur du prédicat utilisé et elle augmente la complexité du graphe de contrôle. Cependant, il faut aussi prendre en compte qu'une analyse dynamique pourrait révéler l'insertion de code mort. De plus, même de la redondance de code pourrait être détecté si un attaquant change le prédicat par `true` et `false` et constate qu'il n'y a pas d'influence sur l'exécution.

11.4.4 Utilisation de fonctions de branchement

Cette méthode consiste à remplacer un saut inconditionnel par un appel à une fonction appelée fonction de branchement [Col+04 ; KRVV04 ; LD03b ; MASB05 ; MJ05 ; OSSM03]. La figure 11.10 illustre cette technique.

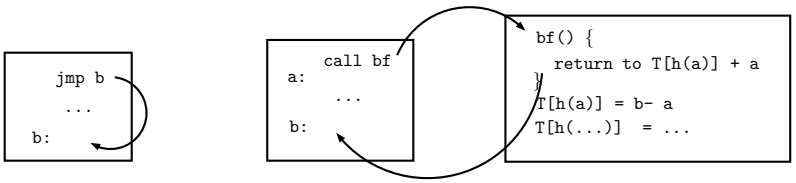


Figure. 11.10 – Utilisation de fonctions de branchement

11.4.5 Transformations de boucles

Parmi les mécanismes cryptographiques, beaucoup nécessitent l'implantation de boucle comme par exemple la multiplication scalaire sur une courbe elliptique. Plusieurs techniques peuvent être appliquées pour transformer ces structures. Nous allons présenter certaines d'entre elles.

Extension de la condition Cette transformation consiste à masquer la condition de terminaison, c'est-à-dire l'expression booléenne qui contrôle la terminaison de la boucle. L'idée est d'utiliser des prédicats opaques qui ne vont pas changer le nombre de fois que les instructions de la boucle sont exécutées.

Par exemple, considérons la boucle suivante avec comme condition de terminaison `q != 9`. Nous pouvons remplacer cette condition par `(q !=9) & P` avec avec `P` un prédicat opaque qui est toujours évalué à vrai.

```
p = 1; q = 1;
while(q != 9)
{
   p = p * q; q++;
}
```

$\xrightarrow{\mathcal{O}}$

```
p = 1; q = 1;
while((q != 9) & (p(p+1)%2 == 0))
{
   p = p * q; q++;
}
```

Pour une analyse statique, la complexité réside dans celle du prédicat opaque utilisé. Avec une analyse dynamique, un attaquant peut se contenter de compter le nombre de fois que les instructions de la boucle sont exécutées.

Découpage de boucle Cette transformation améliore le comportement de la boucle vis-à-vis du cache en découpant l'espace d'itération de manière à ce que le corps de la boucle tienne dans le cache.

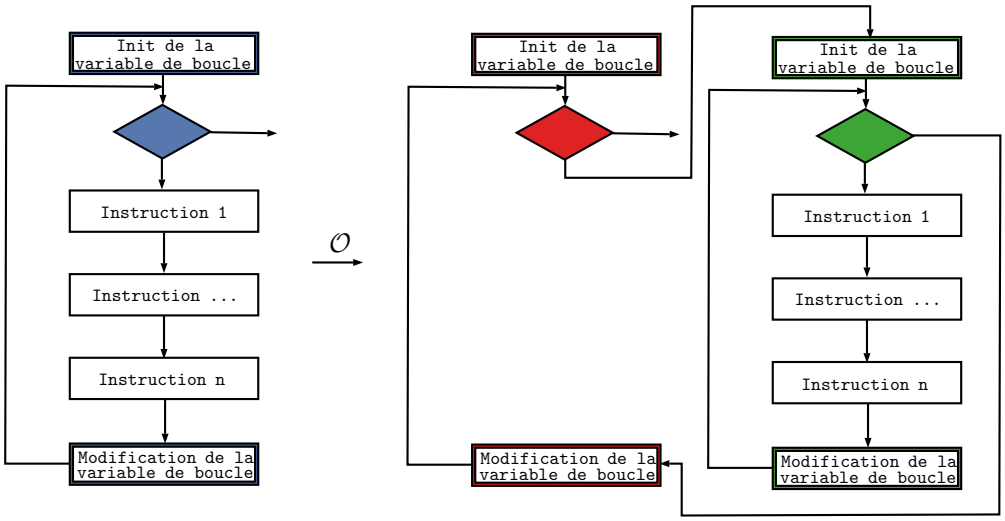


Figure. 11.11 – Découpage de boucle

```
for(i=1;i<= n;i++)
   for(j=1;j<= n;j++)
      a[i,j]=b[i,j];
```

$\mathcal{O}$ →

```
for(I=1;I<=n;I+=64)
   for(J=1;J<=n;J+=64)
      for(i=I;i<= min(I+63,n);i++)
         for(j=J;j<= min(J+63,n);j++)
            a[i,j]=b[i,j];
```

Déroulage de boucle Cette transformation consiste à dupliquer le corps de la boucle une ou plusieurs fois. Par exemple, si le corps est dupliqué k fois, la variable qui contrôle la boucle doit être modifiée en fonction de k. De plus, si l'on connaît la terminaison de la boucle, on peut la dérouler dans son intégralité.

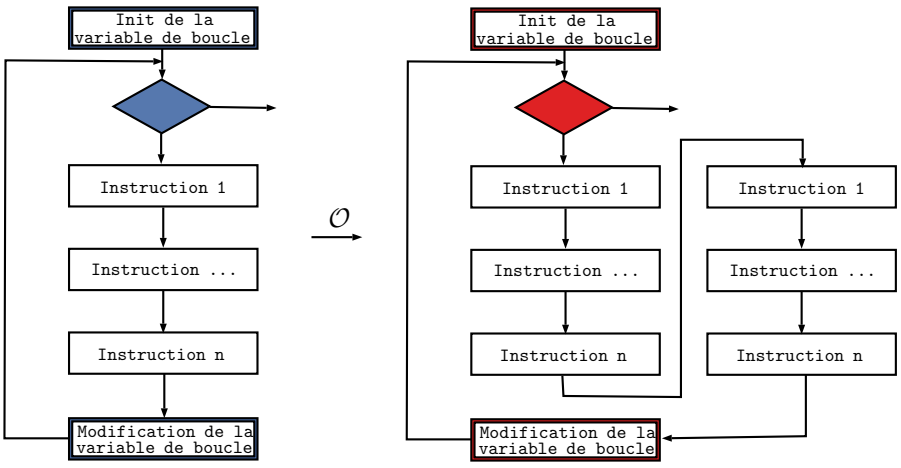


Figure. 11.12 – Déroulage de boucle

```
for(i = 1; i < (n-1); i++)
{
    a[i] += a[i-1]*a[i+1];
}
```

$\xrightarrow{\mathcal{O}}$

```
for(i = 1; i < (n-k); i += k)
{
    a[i] += a[i-1]*a[i+1];
    a[i+1] += a[i]*a[i+2];
    ...
    a[i+k-1] += a[i+k-2]*a[i+k];
}
```

Scindage Cette transformation décompose le corps de la boucle en plusieurs boucles ayant le même espace d'itération. Cependant, la décomposition doit prendre en compte les relations entre les instructions du corps de la boucle. Celui-ci doit être parallélisable. Cette décomposition peut être rendue aléatoire si la suite d'instructions le permet.

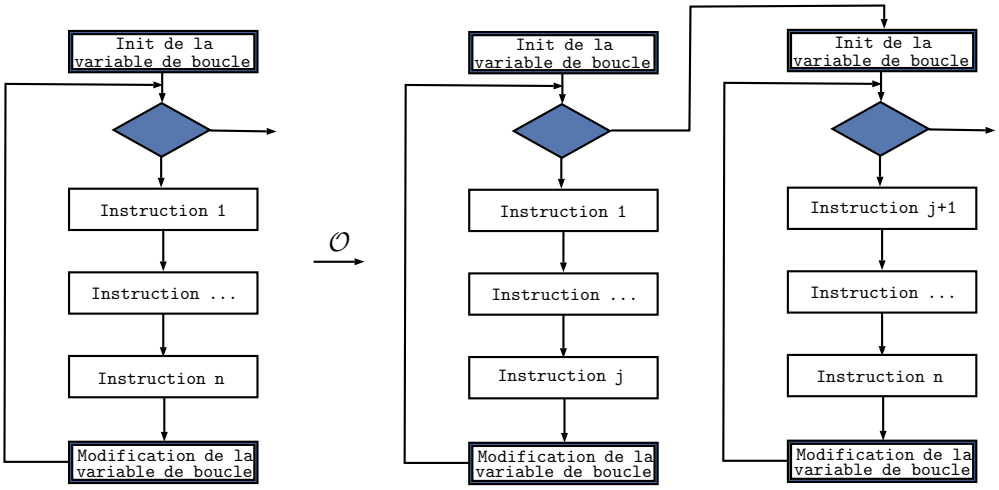


Figure. 11.13 – Scindage de boucle

```
for(i=1;i<n;i+=1)
   a[i] += c;
   b[i+1] += d*b[i-1]*a[i];
```

$\xrightarrow{\mathcal{O}}$

```
for(i=1;i<n;i+=1)
   a[i] += c;
for(i=1;i<n;i+=1)
   b[i+1] += d*b[i-1]*a[i];
```

Changement de direction : Les instructions doivent être insensibles à la direction de la boucle.

```
for(i = 1; i < n; i += 1)
{
    a[i] += c;
}
```

$\xrightarrow{\mathcal{O}}$

```
for(i = n - 1; i > 0; i -= 1)
{
    a[i] += c;
}
```

11.4.6 Parallélisation

La parallélisation est en général utilisée afin d'accroître les performance d'un programme. Cependant, elle peut également permettre de cacher le flot de contrôle en suivant l'une ou l'autre des méthode suivantes :

- Créer des processus qui exécutent des taches inutiles,
- Séparer une section séquentielle de code en plusieurs parties qui peuvent être exécutées en parallèle.

La seconde opération peut sembler difficile en cas de dépendance de données.

Données indépendantes La parallélisation est aisée. Des appels à la bibliothèque `pthread` peuvent être utilisés dans le code pour gérer la parallélisation.

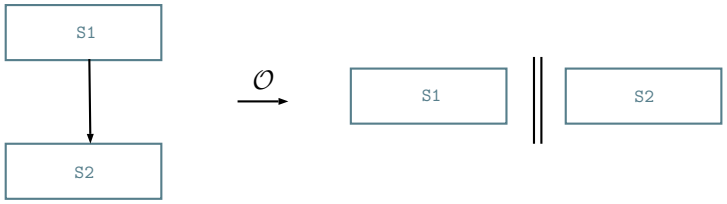


Figure. 11.14 – Parallélisation de sections de code avec indépendance de donnée

Données dépendantes Dans le cas, il faut utiliser la notion de concurrence entre les processus. Le nouveau programme sera exécuté séquentiellement mais le flôt de contrôle passera d'un processus à l'autre.

Figure. 11.15 – Parallélisation de sections de code avec dépendance de données

L'avantage de cette technique est qu'elle rend l'analyse statique plus difficile car le nombre de chemins d'exécution possibles augmente exponentiellement avec celui de

processus utilisés. Pour une analyse dynamique, cela est également plus complexe car l'attaquant doit analyser plusieurs processus en même temps.

11.5 Conclusion

Nous avons revu dans ce chapitre différentes transformations d'obscurcissement de code source. En fonction de la nature du code à obscurcir, certaines méthodes sont plus pertinentes que d'autres. Par exemple, pour des mécanismes de chiffrement en boîte blanche, il faut plutôt privilégier les techniques liées aux tableaux et aux boucles.

Notre analyse nous a révélé que la complexité de la plupart de techniques reposait sur celles des prédicats opaques. Il est donc indispensable d'avoir des méthodes de construction de prédicats assez robustes pour résister aux attaques statiques et dynamiques. L'utilisation de pointeurs ou d'inter-dépendance entre les prédicats apparaissent comme étant de bonne stratégies.

Même en ayant à notre disposition un large éventail de techniques, une problématique reste à être discuter. Dans quel ordre faut-il appliquer les transformations d'obscurcissement ? Nous répondrons à celle-ci dans le chapitre suivant avec le support de notre outil d'obscurcissement.

Chapitre 12

Implantations

Sommaire

Certains outils permettent d'obscurcir de manière automatique ou non du code source. La plupart sont payants comme les solutions de la société Irdeto [1] mais on peut également trouver des outils open source, comme `Obfuscator-LLVM` [JRWM15]. Ce dernier est compatible avec les langages `C`, `C++`, `Objective-C`, `Ada` et `Fortran` en travaillant sur un niveau de représentation intermédiaire. Les fonctionnalités disponibles sont :

- la complexification d'instructions binaires arithmétiques ou booléennes,
- l'insertion de code via l'utilisation de prédicats opaques,
- l'aplatissement du flot de contrôle sans utilisation de pointeurs ou de fonctions de branchement.

Nos travaux rentrent dans le cadre du développement de `LOPSI`. C'est un outil propriétaire d'obscurcissement qui regroupe un ensemble de modules `python` implantant différentes transformations statiques de code source `C`. Il a été développé à la base pour obscurcir un module de chiffrement en boîte blanche, ce qui explique l'utilisation de certaines techniques spécifiques aux tableaux d'entiers et aux boucles `For`. L'architecture a de plus été pensée en terme d'extensibilité. L'arborescence du projet est la suivante :

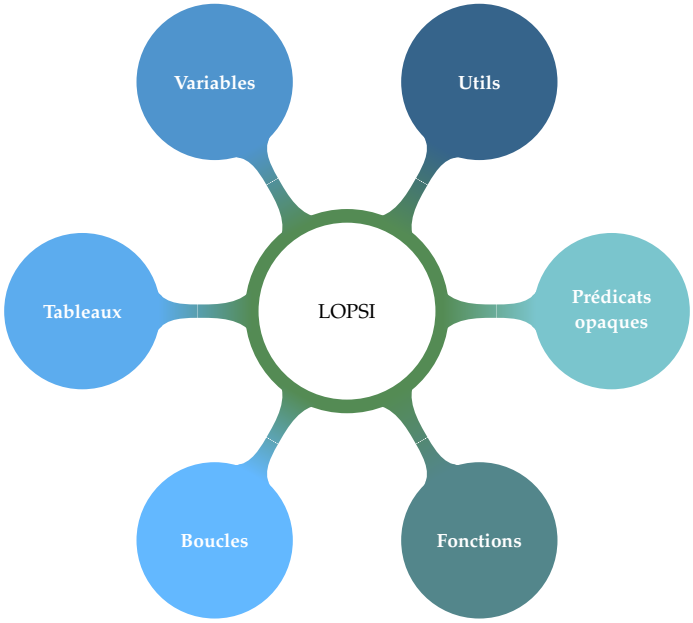


Figure. 12.1 – Architecture de l'outil LOPSI

Nous allons présenter dans ce chapitre les différentes transformations implantées. Nous étudierons également l'ordre d'optimal d'exécutions des celles-ci et les options de compilations à éviter. Nous détaillerons également le module en charge de la mesure de complexité.

1. http ://irdeto.com/index.html

12.1 Module `pycparser`

La mise en oeuvre de `LOPSI` requiert l'utilisation d'un module `python`, `pycparser` [pycparser]. Il s'agit d'un analyseur syntaxique de code `C` sous licence BSD 2, basé sur l'utilisation de deux outils :

- un pré-processeur `cpp`, `gcc -E` ou `clang -E`,
- `PLY` (Python Lex-Yacc) [ply] qui permet la transformation du code en arbre syntaxique (voir figure 12.2).

Définition 12.1.1
Abstract Syntax Tree (AST) Un arbre syntaxique abstrait *(AST en anglais) est un arbre dont les noeuds internes sont marqués par des opérateurs et dont les feuilles sont les opérandes de ces opérateurs. Un exemple d'un tel arbre est donné en figure 12.2.*

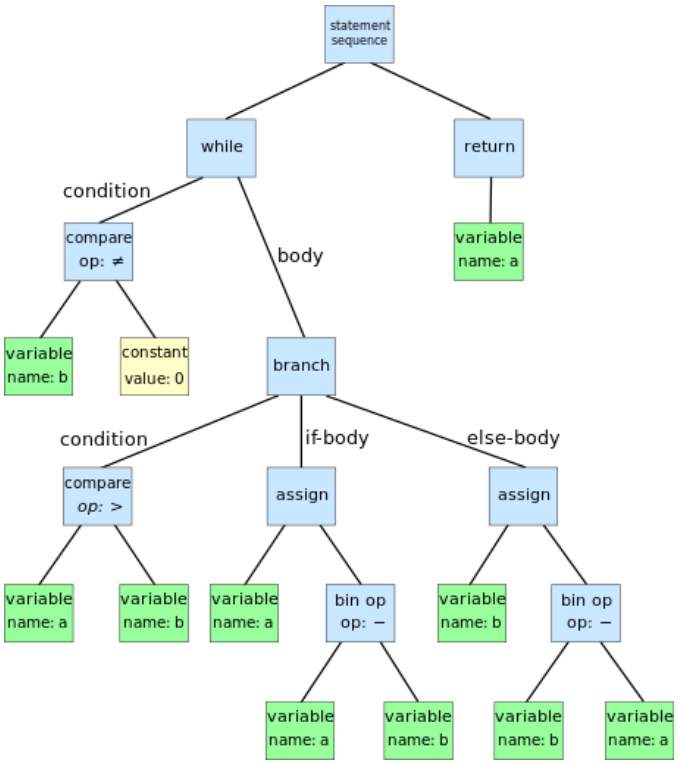


Figure. 12.2 – AST de l'algorithme d'Euclide pour le calcul de PGCD

`pycparser` est compatible ISO C99 mais ne supporte pas les extensions spéciales des compilateurs. Il s'agit d'un projet relativement mature et actif puisque la première version date de Novembre 2008 et qu'il bénéficie de trois mises à jour par an en moyenne.

Cette bibliothèque libre est déjà utilisée, d'après le site officiel, pour développer des outils d'obscurcissement de code `C`, pour vérifier statiquement du code `C`, générer des tests unitaires ou encore développer des extensions pour le langage `C`.

L'analyse d'un fichier source est réalisée en deux temps :

- Les instructions préprocesseur sont gérées par le programme `cpp`.
- Un Arbre Syntaxique Abstrait du code est généré à partir du code par `PLY`.

Lex-Yacc est la combinaison de deux analyseurs, lexical et syntaxique, permettant de produire l'arbre syntaxique d'un code à partir d'une grammaire définie, souvent utilisé par les compilateurs afin de produire du code intermédiaire avant la mise en forme binaire et l'édition des liens (voir figure 12.3)

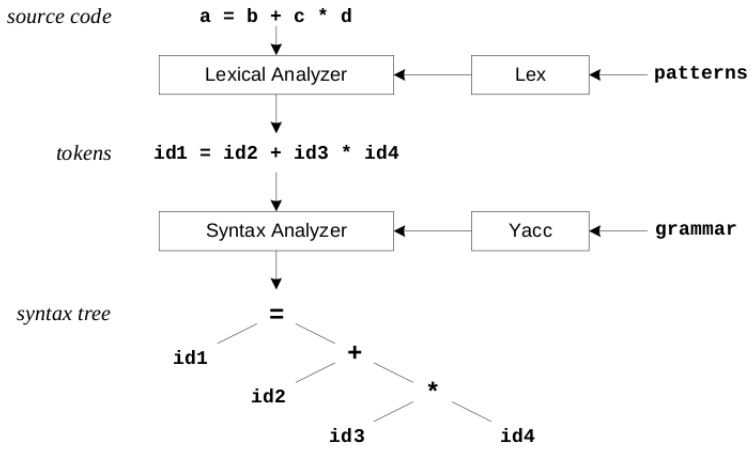


Figure. 12.3 – Construction d'arbre syntaxique à partir du code avec PLY

`pycparser` fournit une configuration pour PLY ainsi qu'une représentation objet de l'arbre manipulable en python via le pattern NodeVisitor (voir figure 12.4) ; afin d'effectuer une opération sur un arbre, il est possible de créer une classe héritant de la classe NodeVisitor, qui propose une manière de visiter l'arbre de manière récursive.

Le parcours d'un arbre par un noeud visiteur se déroule à l'aide des opérations suivantes :

- La méthode `visit` du noeud visiteur est appelée sur le noeud racine de l'arbre (ou du sous-arbre) à manipuler.
- Pour le noeud courant, cette méthode appelle la méthode spécialisée appropriée (`visit_For`, `visit_If`,...).
- En cas d'appel récursif, la méthode `generic_visit` est appelée : celle-ci sera simplement chargée de relancer récursivement la méthode `visit` sur chaque noeud fils du noeud courant.

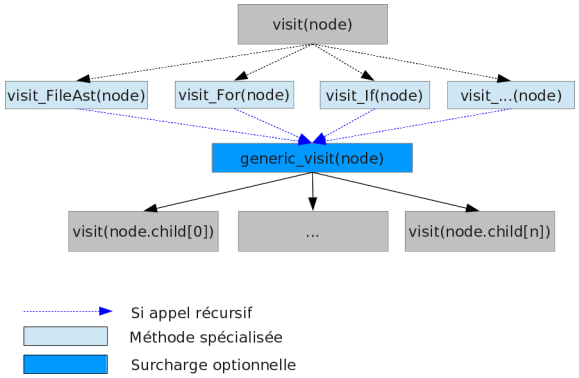


Figure. 12.4 – Le pattern NodeVisitor

Remarque 12.1.2
La surcharge de la méthode `generic_visit` est parfois nécessaire afin de mémoriser des informations sur le parcours effectué par notre noeud (*e.g* : connaître le parent du noeud courant).

Les opérations sur les structures seront donc effectuées en surchargeant les fonctions spécialistes (fonctions sous la forme `visit_XXX`).

12.2 Fonctionnalités implantées

Les transformations sont implantées directement sur l'AST du code obtenu à l'aide la bibliothèque `pycparser` et le code obscurci est recréé à partir de l'arbre modifié, en utilisant les fonctionnalités de celle-ci. Différentes techniques ont été implémentées dans le but d'obscurcir du code `C`. Elles utilisent pour la plupart le pattern noeud visiteur (`NodeVisitor`), et se classent en deux catégories principales : les transformations systématiques (ou automatiques), et les transformations semi-automatiques.

12.2.1 Transformations automatiques

Les transformations automatiques s'appliquent à un arbre entier. Chacune des classes implantées propose des filtres afin de ne pas appliquer les transformations sur les structures concernées. Ces filtres sont passés sous la forme d'expressions régulières. Les techniques implantées sont les suivantes :

- Le renommage de fonctions et de variables,
- L'ajout de faux champs aux structures avec des instructions les manipulant,

- L'ajout de variables aléatoires,
- Duplication de code, insertion de code mort ou inutile à l'aide des prédicats opaques,
- La fusion de méthodes de signature identique,
- L'aplatissement du graphe du flot de contrôle,
- L'utilisation d'un tableau pointeurs de fonctions afin de masquer les appels ; un prédicat opaque peut également être utilisé afin de complexifier et multiplier les chemins d'exécution potentiels.

Certaines de ces transformations (automatiques ou semi-automatiques) reposent sur l'utilisation de prédicats opaques. Nous avons implanté deux types de prédicats : les prédicats arithmétiques et les prédicats basés sur les parcours de graphe.

12.2.2 Transformations semi-automatiques

Pour les transformations semi-automatiques, nous utilisons un système d'étiquette afin de délimiter les endroits du code source où appliquer les transformations. Les techniques implantées sont les suivantes :

- Le renommage de tableaux,
- Le mélange et l'extension de tableaux d'entiers (avec changement des déclaration et changement des accesseurs),
- La transformation de boucles `for`.

Actuellement les boucles gérées doivent être des boucles simples, c'est à dire des boucles dont l'entête est complète (initialisation, condition, incrémentation), ne porte que sur une seule variable (c'est à dire que la condition porte sur la variable initialisée) et où la variable n'est pas modifiée au sein du corps de la fonction. Trois techniques sont implantées : le scindage de boucles avec changement d'orientation, la fission et le partitionnement aléatoire.

Afin d'obtenir un comportement similaire au programme original, le scindage de boucles avec orientation aléatoire doit être appliqué à une boucle *sans état* (*i.e* une boucle dont l'ordre d'exécution est quelconque). Pour la même raison, la fission de boucles doit être appliquée à une boucle dont les instructions sont *parallélisables*.

Enfin, le partitionnement aléatoire de boucles doit être utilisé sur une boucle *sans état*, mais requiert que l'initialisation et la condition de la boucle porte sur une constante connue à la compilation (éventuellement exprimée sous forme d'opération arithmétique simple, c'est à dire composée exclusivement des opérateurs $+, -, *, /$ et de constantes).

Cette technique est donc plus spécialement adaptée à la protection d'algorithmes cryptographiques tels que AES, SHA-1, SHA-3, ...

Afin d'appliquer des transformations non systématiques sur le code, il est nécessaire de délimiter le code avec des étiquettes. Étant donné qu'après utilisation de cpp par pycparser, tous les commentaires et macros sont supprimés, les étiquettes utilisées sont des structures avec des noms prédéfinis et elles sont regroupées dans un fichier entête tags.h.

La macro OBFUSCATION_TAG permet d'introduire les différents étiquettes tout en conservant la possibilité de compiler de manière normale le code original. Dans le cadre de l'obscurcissement d'un code, une étiquette correspond à une structure d'un type précis. La macro permet de créer une instance de la structure, numérotée en fonction de la ligne ou celle-ci est placée : cela permet de créer des instances sans conflit de nom au sein d'un même fichier, et ce de manière automatique.

Lorsque l'on désire appliquer une transformation semi-automatique TRANSFORMATION_SA à une partie d'un code, il suffira d'inclure dans le fichier source le fichier tags.h et d'encadrer la portion de code de la manière suivante :

```
OBFUSCATION_TAG(TRANSFORMATION_SA_BEGIN);
  code
OBFUSCATION_TAG(TRANSFORMATION_SA_END);
```

Une seule paire d'étiquette n'est pas utilisée pour délimiter l'effet d'une transformation mais la partie du code à obscurcir : la paire (OBFUSCATION_BEGIN, OBFUSCATION_END). En effet, après application de cpp sur le code source à obscurcir, les instructions hors du cadre de délimitation de cette paire sont supprimées.

Du point de vue python, les classes implantant les transformations semi-automatiques vérifient la présence des tags avant de s'exécuter sur le noeud courant. À la fin, toutes les étiquettes sont supprimées de l'AST avant la conversion vers le code C.

12.2.3 Permutations des transformations

En premier lieu, nous allons discuter des permutations possibles théoriques des différentes transformations sus-citées puis nous définirons un flot aléatoire pour l'obscurcissement. Ces transformations sont, en théorie, permutables. Cependant, certaines transforment de manière plus complexe le code obscurci et par conséquent, certains ordonnancements pourraient diminuer drastiquement la performance du programme original.

Par exemple, les méthode de rajout de variables/instructions ou de rajout de branches sont réalisées de manière probabiliste et en fonction du nombre d'instructions déjà présentes dans le code. Dans ce cas, il est conseillé de réaliser les transformations qui augmentent le nombre d'instructions avant ces dernières, afin de ne pas dupliquer les insertions opérées et de les créer par la suite.

Le rajout des pointeurs de fonctions doit, comme précédemment expliqué, être appliqué sur un code dont le nombre de blocs n'a pas été augmenté de manière trop importante. De plus, le tableau de pointeurs créé doit être obscurci : pour ces raisons là, il est conseillé d'appliquer cette transformation après le découpage et la fission de boucles, mais avant l'obscurcissement des tableaux.

L'aplatissement de graphe supprimant les boucles, il est nécessaire d'appliquer les méthodes de transformation de boucles en amont.

Il est aussi nécessaire de prendre en compte la complexité au niveau de l'outil d'obscurcissement : par exemple, il peut paraître indifférent d'appliquer la transformation des symboles exportés avant ou après une méthode quelconque. Cependant, l'appliquer avant permettrait très certainement d'avoir moins de remplacements à effectuer pour l'outil d'obscurcissement.

Par rapport aux considérations précédentes, le flot aléatoire proposé est le suivant :

1. Transformation des symboles exportés.
2. Rajout de branches aléatoire.
3. Indifféremment on peut alterner d'obscurcissement de boucles (scindage et fission).
4. Rajout de pointeurs de fonctions.
5. Indifféremment on peut alterner les méthodes de ré-indexage de tableau et le partitionnement aléatoire de boucles.
6. Aplatissement du code.
7. Rajout de variables/instructions.

12.3 Compilation et optimisation

Même si un code est obscurci et que l'on sait qu'il y un impact sur ses performances, on peut vouloir mettre des options d'optimisation lors de la compilation. Cependant, il faut vérifier que celles-ci ne cassent pas certaines transformations introduites.

Prenons par exemple le cas de `gcc`. Les options `-finline-functions-called-once` et `-finline-functions` pourraient retirer les fonctions de branchement.

12.4 Exemples d'utilisation

Nous avons réalisé des tests sur une implantation de l'algorithme `SHA-3` ainsi que sur certaines fonctions de la bibliothèques `MPHELL`. Nous présentons ici le cas de la multiplication scalaire naïve pour les courbes elliptiques. Nous avons appliqué une partie de la séquence des transformations d'obscurcissement définie précédemment. Pour la multiplication scalaire, nous avons utilisé `Hopper` afin de récupérer les graphes de flot de contrôle (voir figures 12.5 et 12.6).

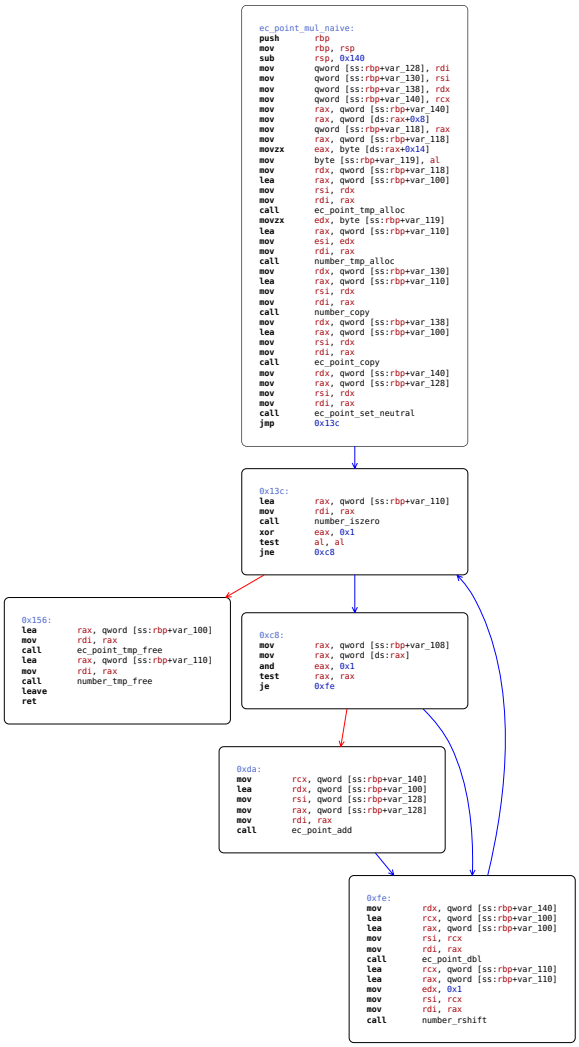


Figure. 12.5 – Graphe de flot de contrôle de la fonction de multiplication naïve sur les courbes elliptiques

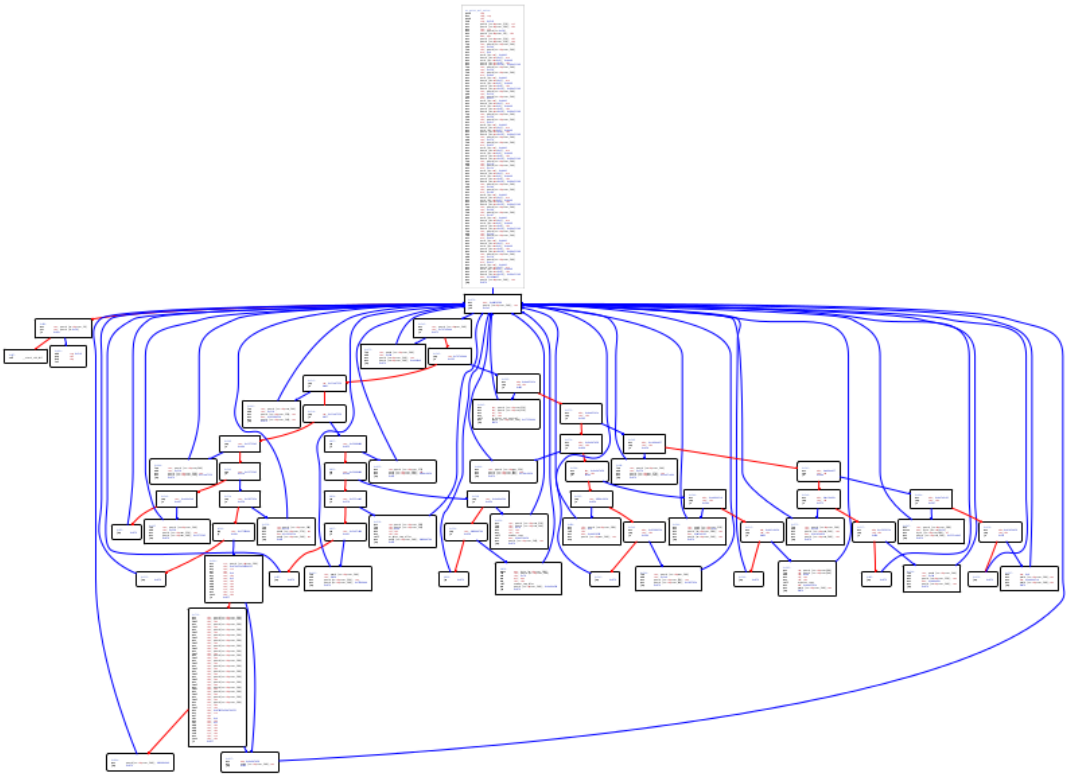


Figure. 12.6 – Graphe de flot de contrôle de la fonction de multiplication naïve sur les courbes elliptiques obscurcie

Pour `SHA-3`, nous avons exploité les résultats pour la fonction `main`. Les figures 12.7 et 12.8 présentent les graphes de flot de contrôle et les figures 12.9 et 12.10 les graphes d'appels. Ils ont été obtenus avec `Metasm`.

En analysant le graphe d'appel après obscurcissement, on peut remarquer qu'il n'est plus correct car de nombreuses fonctions n'apparaissent plus. De plus, de manière purement empirique, on pourrait dire que les graphes de flot de contrôle sont plus complexes après application de nos techniques d'obscurcissement.

En utilisant notre module de calcul de complexité, nous avons accès à des indicateurs concrets de la puissance des techniques appliquées.

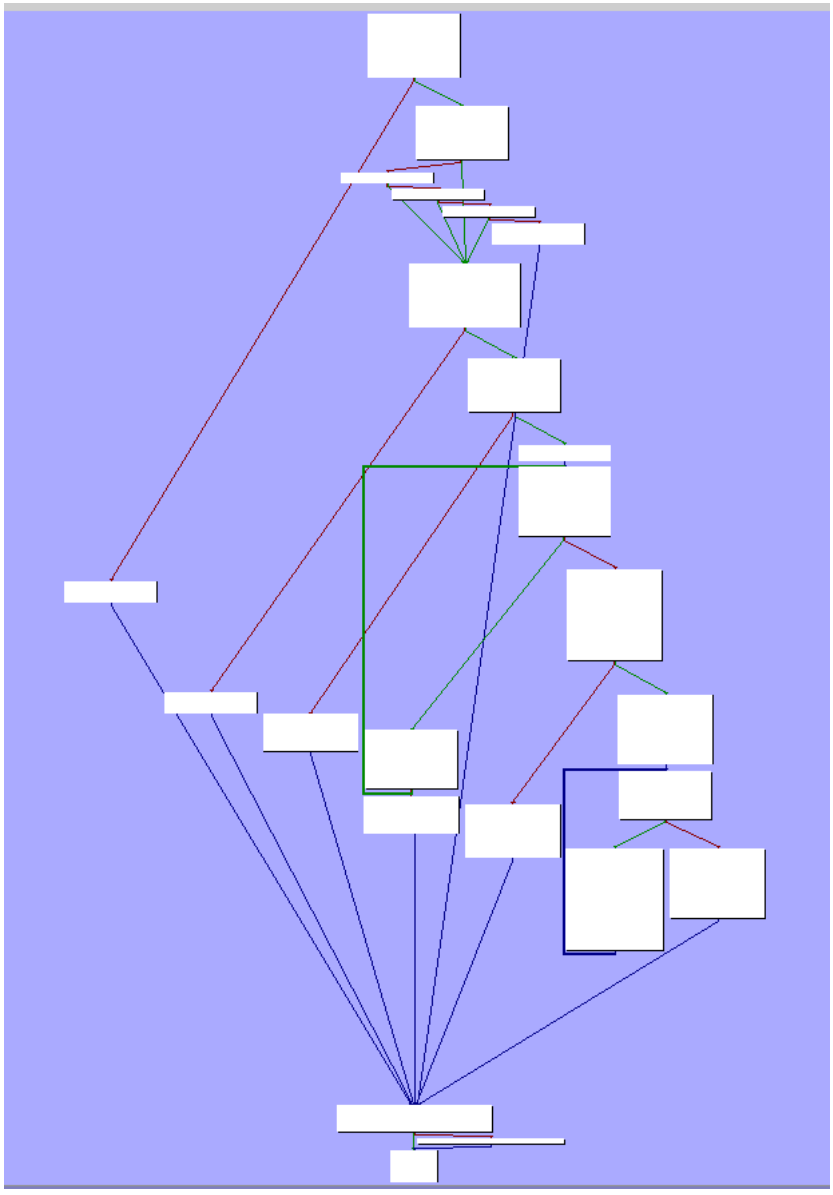


Figure. 12.7 – Graphe de flot de contrôle de la fonction main de `SHA-3`

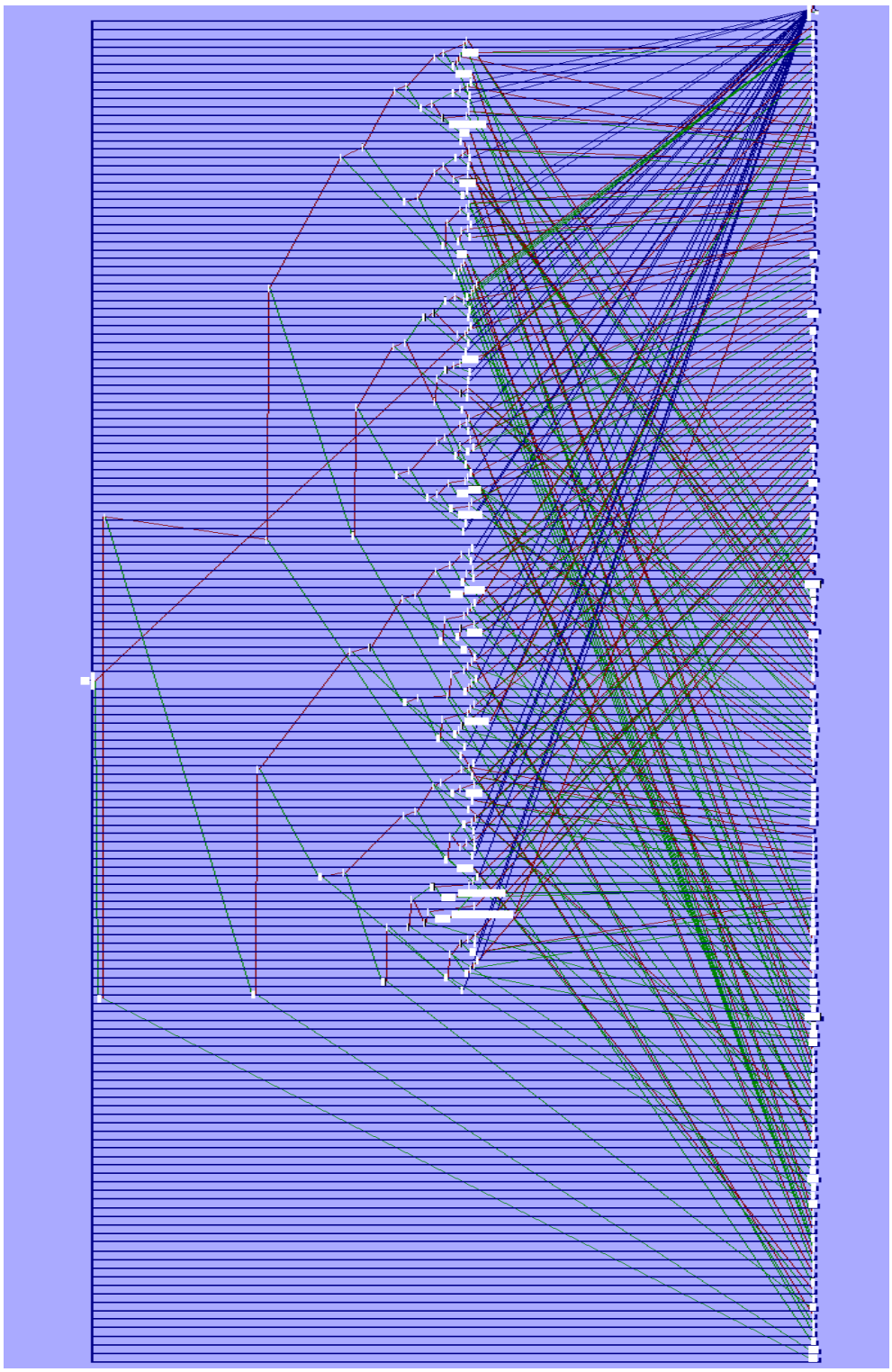


Figure. 12.8 – Graphe de flot de contrôle de la la fonction main de SHA-3 obscurcie

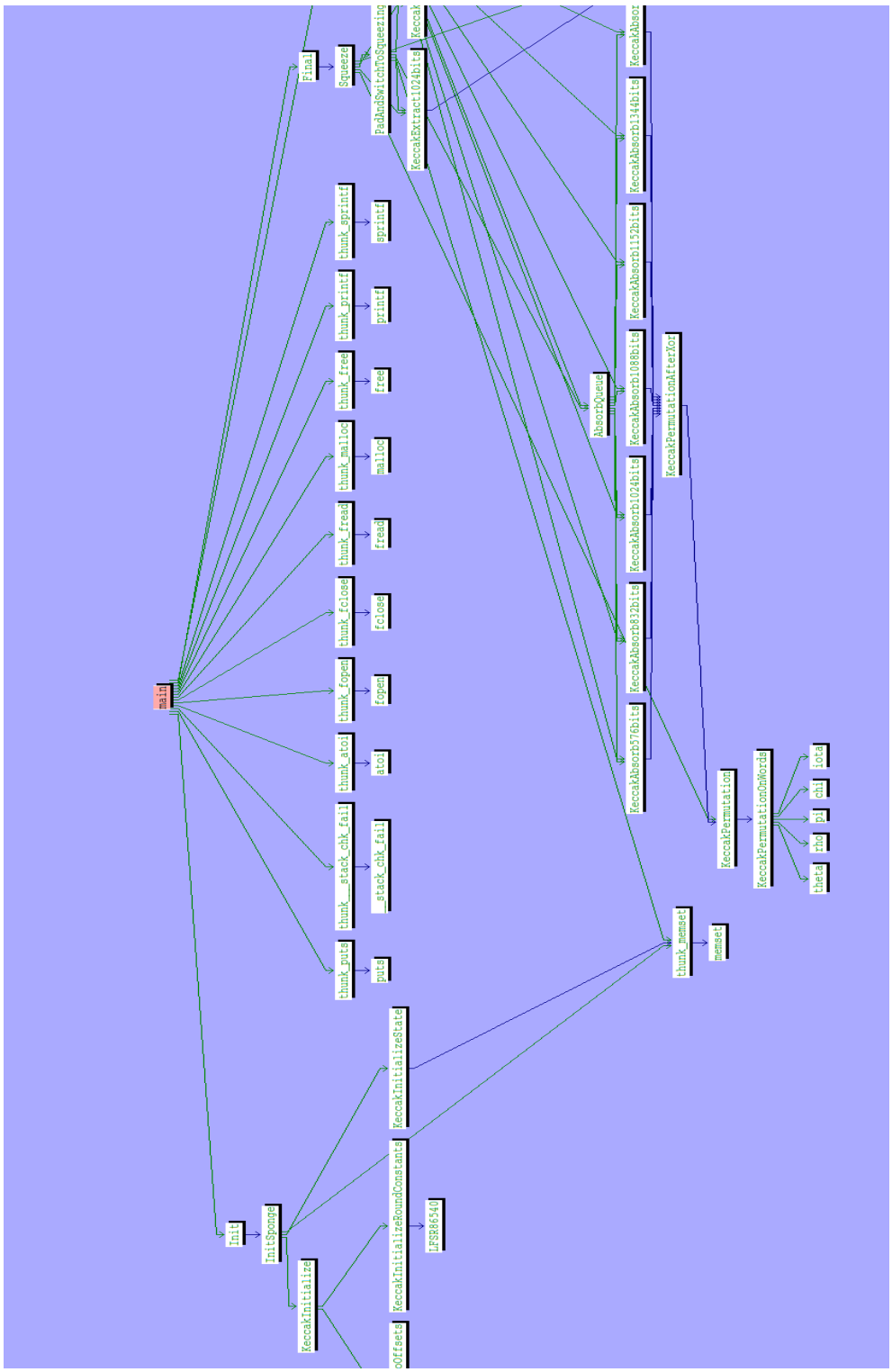


Figure. 12.9 – Graphe d'appels de la fonction main de SHA-3

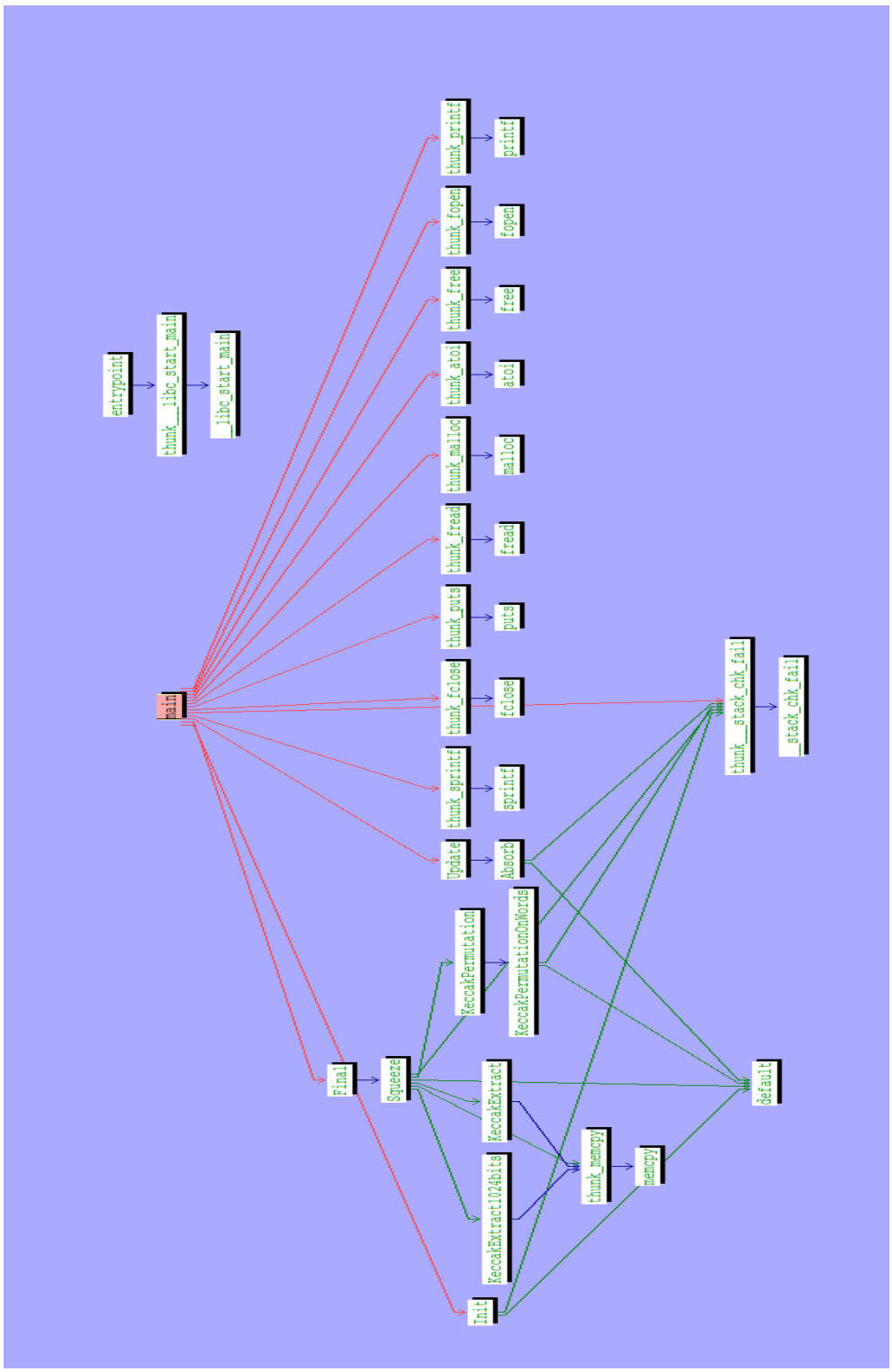


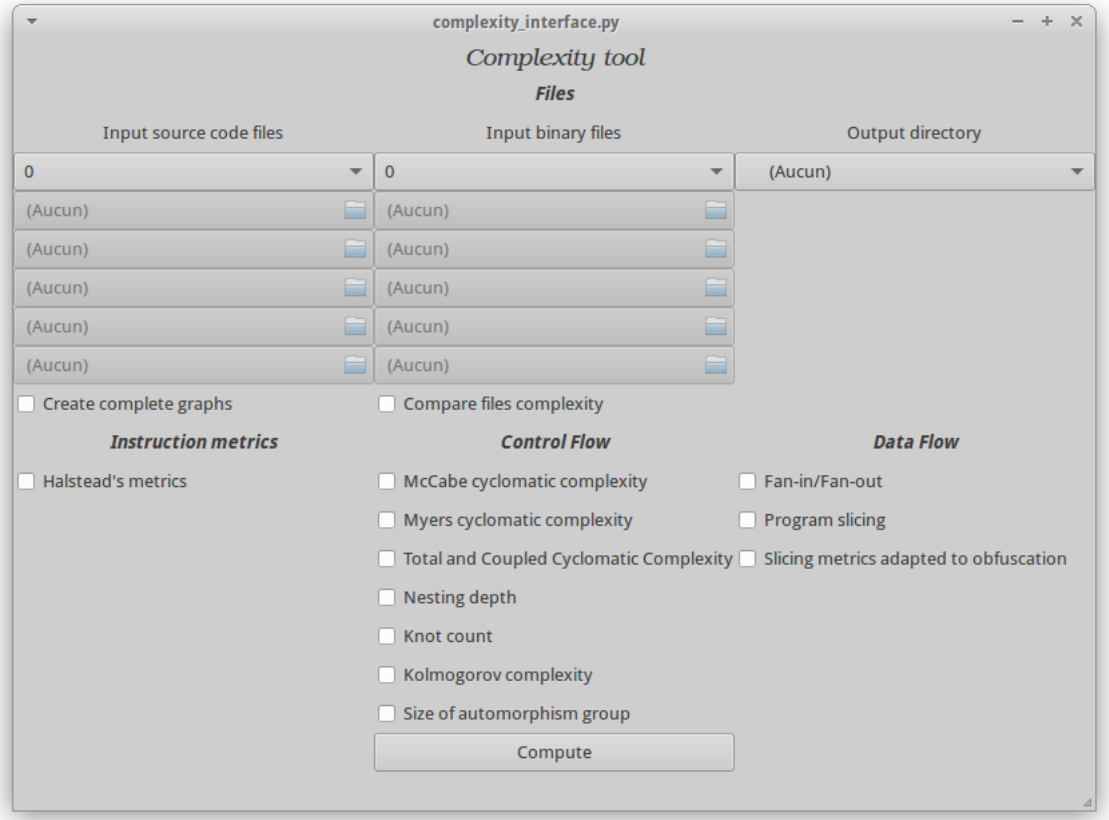
Figure. 12.10 – Graphe d'appels de la la fonction main de SHA-3 obscurcie

12.5 Module de complexité

Ce module peut agir directement sur l'AST utilisé lors de l'obscurcissement mais également sur les graphes produits par l'outil `Metasm`. Cela nous permet de voir la différence de reconstruction de graphe de flot de contrôle mais également, de voir la complexité si un attaquant avait accès aux codes sources, donc à un graphe de flot de contrôle correct.

D'autres outils existent pour calculer la complexité d'un programme, comme par exemple `Complexity`[Complexity] et `C# Metrics tool`[CMT]. En général, seule une métrique est calculée par ces outils, les plus complets étant payants. De plus, on ne peut pas leur donner directement comme entrée un graphe issu de `Metasm` ou l'arbre utilisé par notre outil d'obscurcissement. D'autre part, la complexité de Kolmogorov n'est pas calculée hormis par l'outil développé par les auteurs de [ZSTDL13] qui n'est pas encore disponible à l'heure actuelle pour les graphes. Notre module permet donc de palier aux désavantages des outils existants même si nous restreignons également le format des entrées.

Ce module peut être appelé directement lors de l'application des techniques d'obscurcissement mais également a posteriori à l'aide de l'interface suivante.



Nous avons implanté les métriques suivantes :

- les métriques d'Halstead,
- les différentes complexités cyclomatiques,
- les niveaux d'imbrication,
- la complexité de Kolmogorov.

Nos expérimentations nous ont permis de quantifier l'influence de l'aplatissement de flot de contrôle et des modifications de boucles sur la complexité cyclomatique.

Avec l'aplatissement du flot de contrôle, pour chaque bloc basique, on introduit un nouveau prédicat. De plus, on rajoute un `switch`. De ce fait, la complexité cyclomatique va être augmentée du nombre de blocs basiques + 1. L'utilisation des versions d'aplatissement de flot de contrôle avec fonctions de branchement va augmenter également les valeurs des complexités cyclomatiques totale et couplée.

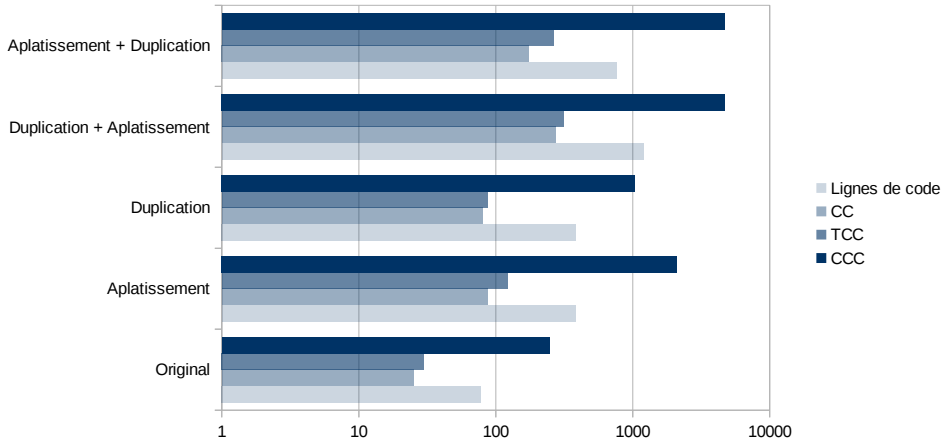
Le changement de direction ne modifie pas la complexité cyclomatique. L'extension de la condition d'une boucle ne modifie pas non plus cette métrique, mais on peut mesurer les effets avec l'adaptation de Myers. Le déroulage de boucles peut modifier la complexité si le corps de la boucle contient une structure de contrôle. Dans ce cas, la métrique est augmenté de $n \times CC(SC)$ où n est le nombre de fois que la boucle est déroulée et $CC(SC)$ la complexité cyclomatique du corps de la boucle. Le scindage de boucles augmente la métrique du code de $n \times CC(B)$ où n est le nombre de boucles créées et $CC(B)$ la complexité de la boucle initiale. Le déroulage de boucles va à la fois augmenter cette métrique et le niveau d'imbrication.

Concernant les métriques d'Halstead, il était prévisible que les techniques implantées allaient fortement augmenter ces complexités car beaucoup d'instructions sont rajoutées.

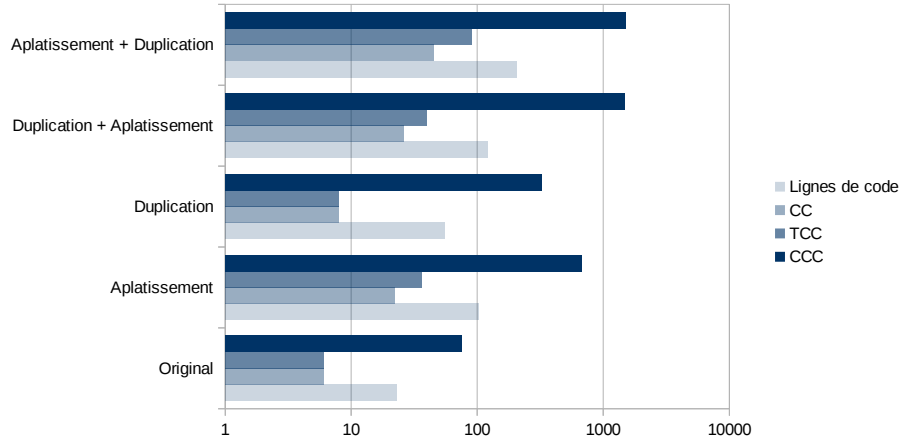
Le tableau 12.1 présente une comparaison de différentes métriques sur certaines fonctions de `SHA-3` sur lesquelles nous avons appliqué des techniques d'obscurcissement. *CC* désigne la complexité cyclomatique de McCabe, *TCC* la complexité cyclomatique totale, *CCC* la complexité cyclomatique couplée. Nous avons également indiqué la longueur du code. Les techniques appliquées sont l'aplatissement de code (Apl) et l'ajout de branches aléatoires (Dup). Nous avons également testé différentes combinaisons. La figure 12.11 présente ces résultats sous forme graphique.

Technique	Fonction	lignes de code	*CC*	*TCC*	*CCC*
.	Absorb	78	25	30	247
	AbsorbQueue	23	6	6	76
	Final	4	1	8	626
	Hash	4	10	21	4195
	Squeeze	31	8	9	313
	Main	57	18	29	4203
Apl	Absorb	381	87	122	2105
	AbsorbQueue	102	22	36	680
	Final	13	4	33	5585
	Hash	66	22	51	36133
	Squeeze	126	30	51	2791
	Main	217	59	88	36170
Dup	Absorb	387	80	87	1041
	AbsorbQueue	55	8	8	323
	Final	4	1	18	2646
	Hash	34	14	38	17606
	Squeeze	93	18	25	1323
	Main	233	67	91	17659
Dup + Apl	Absorb	1201	273	312	4714
	AbsorbQueue	121	26	40	1489
	Final	13	4	76	12235
	Hash	118	38	93	80252
	Squeeze	299	73	117	6116
	Main	336	92	147	80306
Apl + Dup	Absorb	761	175	265	4687
	AbsorbQueue	207	45	91	1519
	Final	29	9	69	12472
	Hash	134	46	96	80506
	Squeeze	255	61	109	6232
	Main	428	119	169	80579

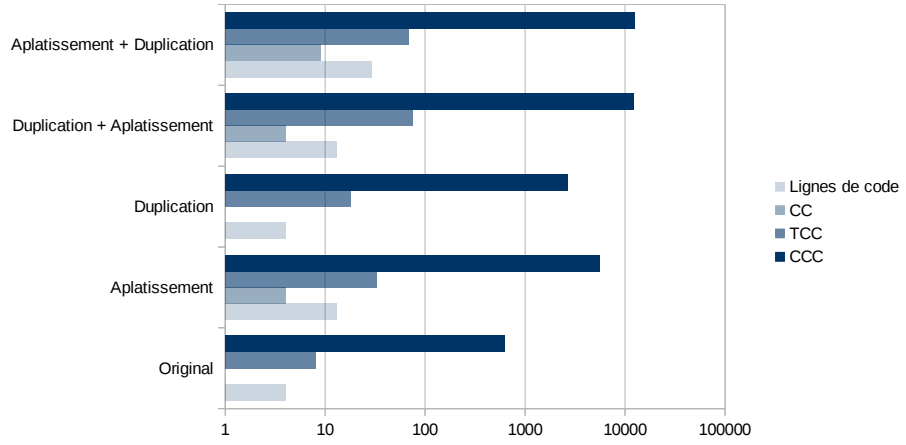
Tableau 12.1 – Différentes mesures de complexité sur certaines fonctions de [SHA-3]



(a) Fonction `Absorb`

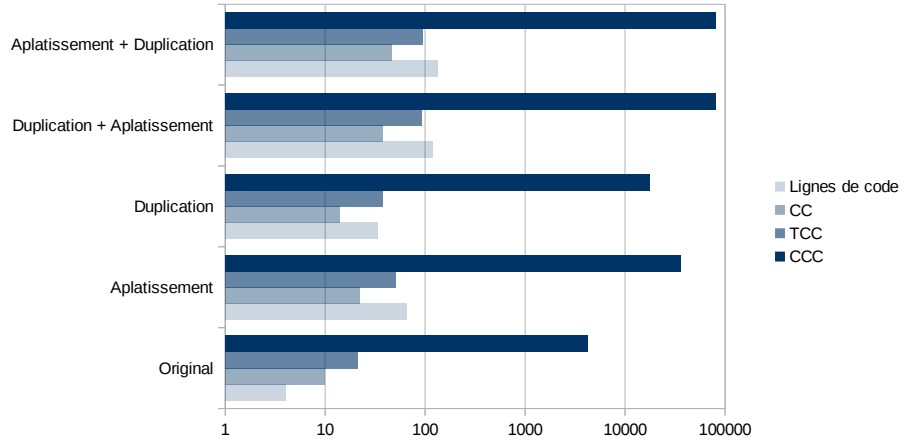


(b) Fonction `AbsorbQueue`

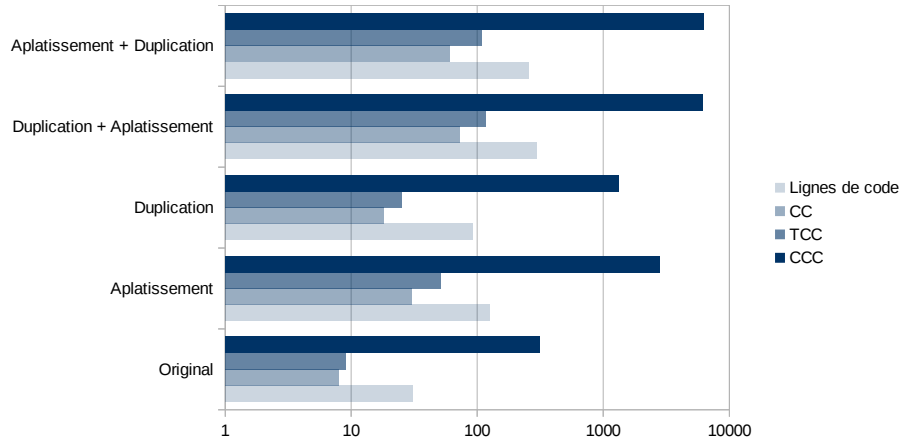


(c) Fonction `Final`

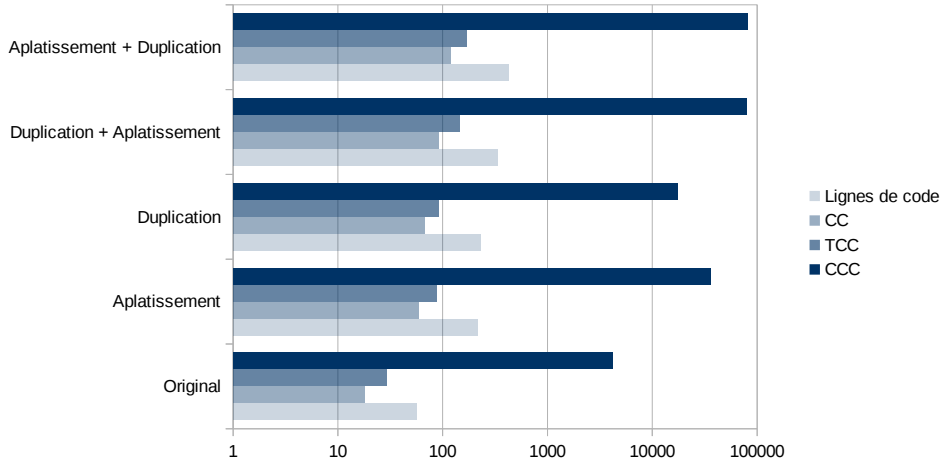
Figure. 12.11 – Comparaison de différentes métriques de complexité sur des fonctions de `SHA-3`



(d) Fonction `Hash`



(e) Fonction `Squeeze`



(f) Fonction `Main`

Figure. 12.11 – Comparaison de différentes métriques de complexité sur des fonctions de `SHA-3`

Si l'on compare les résultats pour les métriques *CC*, *TCC* et *CCC* on peut remarquer que la dernière permet de mieux appréhender la difficulté d'une fonction. En effet, *CC* ne prend pas en compte les appels de fonction, par conséquent même si la fonction *main* fait appel à la fonction *Absorb*, son score est moins élevé. De plus, *TCC* prend en compte les appels de fonctions, cependant on ne comptabilise que la valeur de *CC* des fonctions appelées. De ce fait, nous n'avons accès qu'à un niveau d'abstraction. Étant donné que, la métrique *CCC* utilise les *CCC* des fonctions appelées, on peut remonter tout le graphe d'appel grâce à cette récursivité.

On peut également remarquer que l'aplatissement de code multiplie par un facteur compris entre 2 et 4 la *CC* d'une fonction. La duplication étant faite de manière aléatoire, le facteur n'est pas constant et peut varier entre 1 et 14. Pour finir, l'ordre d'application des techniques fait également une différence sur la valeur de *CC*. On peut noter que si l'on duplique le code avant l'aplatissement, les résultats sont meilleurs cependant le code est plus gros. La différence est moins nette sur *CCC* car les effets se compensent.

L'analyse statique des codes obscurcis donnés sous la forme d'un exécutable est complexe car les graphes que nous récupérons avec nos outils d'analyse ne fournissent pas des graphes corrects. Par conséquent, les mesures que l'on observe sur les codes sources, nous donnent une borne inférieure quant à la difficulté qu'un attaquant aurait à rétro-concevoir le code à partir du binaire en utilisant une analyse statique.

Les évolutions possibles pour ce module sont :

- Le rajout des complexités liées aux données,
- Le développement de scripts permettant aux outils d'analyse de donner un graphe de flot de contrôle plus proche de la réalité,
- La poursuite de l'étude de la complexité de Kolmogorov et celle des groupes d'automorphisme,
- La mise en place des complexités liées aux données.

Chapitre 13

Conclusion et perspectives

Dans cette partie, nous nous sommes intéressés à l'obscurcissement de code source qui est une solution technique à la protection logicielle.

Afin de poser le contexte de travail et de présenter certains moyens à la disposition d'un attaquant ou d'une attaquante, nous avons rappelé certaines techniques d'analyse de programme. Elles permettent de déduire de l'information d'un programme en utilisant une approche statique ou dynamique cependant elles peuvent devenir très complexes en présence de code ne respectant pas les règles standard de programmation, ou de pointeurs généraux et de fonctions. Ce dernier point apparaît comme étant une base possible de techniques visant à complexifier l'analyse de programme. Nous avons également présenté différents outils mettant en œuvre ces techniques et comparé leurs résultats sur un code a priori complexe.

Nous avons ensuite étudié différentes mesures de complexité de programme. Ces métriques nous permettront par la suite de juger de la qualité des techniques d'obscurcissement en vérifiant si elles complexifient bien les programmes d'entrée. Nous avons distingué les mesures basées sur les instructions, le flot de contrôle et le flot de données. Nous avons également rappelé les résultats de complexité sur l'utilisation des pointeurs qui justifient le fait que l'analyse de programme soit plus complexe en leur présence. En plus des métriques classiquement utilisées, nous avons présenté la complexité de Kolmogorov qui permet d'avoir une autre mesure sur le graphe de flot de contrôle et nous avons donné d'autres pistes pour de futures mesures de complexité liées à la structure du groupe d'automorphisme du graphe.

Concernant les techniques d'obscurcissement, après un rappel théorique des définitions, nous avons présenté des transformations opérant sur le code source en ciblant les données ou le flot de contrôle. Ces techniques préservent le comportement du pro-

gramme si les règles d'application sont respectées. Par exemple, l'obscurcissement d'un tableau nécessite de modifier l'ensemble des fonctions accédant à celui-ci. Nous avons également pu vérifier que pour certaines transformations, les variantes utilisant des pointeurs permettaient d'augmenter la complexité du programme obscurci.

Nous avons continué l'implantation de l'outil d'obscurcissement `LOPSI` en apportant de nouvelles techniques mais également un module de calcul de complexité qui permet à l'utilisateur d'avoir une estimation de la qualité des transformations appliquées. Nous avons également étudier l'ordre d'application des techniques permettant de concilier la complexité et la performance. Notre outil a été testé sur une implantation de l'algorithme `SHA-3` et sur une partie de notre bibliothèque `MPHELL`.

Comme perspective d'évolution au niveau des mesures de complexité, il faudrait dériver une nouvelle métrique analysant le groupe d'automorphisme et étudier le lien avec la complexité de Kolmogorov. Pour ce faire, il serait nécessaire de comparer l'évolution de la taille du groupe mais également sa structure sur des codes présentant des structures différentes mais également suite à l'application de diverses techniques d'obscurcissement. De plus, il serait intéressant d'établir un lien plus précis entre ces deux notions dans le contexte des graphes que nous avons à traiter. Il serait intéressant d'étudier des techniques d'obscurcissement agissant sur le binaire et celles permettant d'obtenir un code auto-modifiable afin de diversifier notre implantation. D'autre part il faudrait analyser précisément l'effet du compilateur sur les techniques appliquées. De plus, la prise en charge d'autres langages comme l'`Objective-C` ou le `C++` serait un avantage pour une future utilisation industrielle. Enfin, le portage de notre outil en `C` nous permettrait d'avoir un exécutable plus facilement distribuable.

Liste des Figures

Bibliographie

Bibliothèques logicielles

[CMT] DESIGNS, S. *C# Metrics tool*. URL : http://www.semdesigns.com/Products/Metrics/CSharpMetrics.html. [239]

[Complexity] KORB, B. *GNU Complexity*. URL : https://www.gnu.org/software/complexity/. [239]

[GAP] . *GAP – Groups, Algorithms, and Programming, Version 4.7.9*. The GAP Group. 2015. URL : http://www.gap-system.org. [197]

[Hopper] APPS, C. *Hopper,The OS X and Linux Disassembler*. URL : http://www.hopperapp.com/. [171]

[IDA] HEX-RAYS. *IDA, the multi-processor disassembler and debugger*. URL : https://www.hex-rays.com/products/ida/. [171]

[Metasm] . *Metasm, The METASM assembly manipulation suite*. URL : http://metasm.cr0.org/. [171]

[ply] . *PLY (Python Lex-Yacc)*. URL : http://www.dabeaz.com/ply/. [227]

[pycparser] . *pycparser, Complete C99 parser in pure Python*. URL : https://github.com/eliben/pycparser. [227]

Articles et livres

[AB15] APPLEBAUM, B. et BRAKERSKI, Z. « Obfuscating Circuits via Composite-Order Graded Encoding. » In : *TCC (2)*. Sous la dir. de DODIS, Y. et NIELSEN, J. B. T. 9015. Lecture Notes in Computer Science. Springer, 2015, p. 528–556. [207]

[ABGSZ13] ANANTH, P., BONEH, D., GARG, S., SAHAI, A. et ZHANDRY, M. « Differing-Inputs Obfuscation and Applications. » In : *IACR Cryptology ePrint Archive* 2013 (2013), p. 689. [207]

[AMSBBP07] ANCKAERT, B., MADOU, M., SUTTER, B. D., BUS, B. D., BOSSCHERE, K. D. et PRENEEL, B. « Program obfuscation : a quantitative approach. » In : *QoP*. Sous la dir. de KARJOTH, G. et STØLEN, K. ACM, 2007, p. 15–20. [177]

[Bar+01] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S. et YANG, K. « On the (Im)possibility of Obfuscating Programs ». In : *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '01. London, UK, UK : Springer-Verlag, 2001, p. 1–18. [203–207]

[Bar+12] BARAK, B., GOLDREICH, O., IMPAGLIAZZO, R., RUDICH, S., SAHAI, A., VADHAN, S. P. et YANG, K. « On the (im)possibility of obfuscating programs. » In : *Journal of the ACM* 59.2 (2012), p. 6. [203–207]

[BCP14] BOYLE, E., CHUNG, K.-M. et PASS, R. « On Extractability Obfuscation. » In : *TCC*. Sous la dir. de LINDELL, Y. T. 8349. Lecture Notes in Computer Science. Springer, 2014, p. 52–73. [207]

[BGEC04] BILLET, O., GILBERT, H. et ECH-CHATBI, C. « Cryptanalysis of a white box AES implementation ». In : *Selected Areas in Cryptography*. Springer. 2004, p. 227–240. [156]

[BGKPS14] BARAK, B., GARG, S., KALAI, Y. T., PANETH, O. et SAHAI, A. « Protecting Obfuscation against Algebraic Attacks. » In : *EUROCRYPT*. Sous la dir. de NGUYEN, P. Q. et OSWALD, E. T. 8441. Lecture Notes in Computer Science. Springer, 2014, p. 221–238. [207]

[BR14] BRAKERSKI, Z. et ROTHBLUM, G. N. « Virtual black-box obfuscation for all circuits via generic graded encoding ». In : *Theory of Cryptography*. Springer, 2014, p. 1–25. [207]

[BRS15] BLAZY, S., RIAUD, S. et SIRVENT, T. « Data tainting and obfuscation : Improving plausibility of incorrect taint. » In : *SCAM*. Sous la dir. de GODFREY, M. W., LO, D. et KHOMH, F. IEEE, 2015, p. 111–120. [169]

[CEJVO02] CHOW, S., EISEN, P., JOHNSON, H. et VAN OORSCHOT, P. C. « White-box cryptography and an AES implementation ». In : *Selected Areas in Cryptography*. Springer. 2002, p. 250–270. [156]

[CGJZ01] CHOW, S., GU, Y. X., JOHNSON, H. et ZAKHAROV, V. A. « An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. » In : *ISC*. Sous la dir. de DAVIDA, G. I. et FRANKEL, Y. T. 2200. Lecture Notes in Computer Science. Springer, 2001, p. 144–155. [215]

[CGPDE06] CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L. et ENGLER, D. R. « EXE : automatically generating inputs of death. » In : *ACM Conference on Computer and Communications Security*. Sous la dir. de JUELS, A., WRIGHT, R. N. et VIMERCATI, S. D. C. di. ACM, 2006, p. 322–335. [169]

[CK94] CHIDAMBER, S. R. et KEMERER, C. F. « A Metrics Suite for Object Oriented Design ». In : *IEEE Trans. Softw. Eng.* 20.6 (juin 1994), p. 476–493. [177]

[CN09] CHRISTIAN, C. et NAGRA, J. *Surreptitious Software : Obfuscation, Watermarking and Tamperproofing for Software Protection*. Addison Wesley, 2009. [215]

[Col+04] COLLBERG, C. S., CARTER, E., DEBRAY, S. K., HUNTWORK, A., KECECIOGLU, J. D., LINN, C. et STEPP, M. « Dynamic path-based software watermarking. » In : *PLDI*. Sous la dir. de PUGH, W. et CHAMBERS, C. ACM, 2004, p. 107–118. [218]

[CP10] CAPPAERT, J. et PRENEEL, B. « A general model for hiding control flow. » In : *Digital Rights Management Workshop*. Sous la dir. d'AL-SHAER, E., JIN, H. et JOYE, M. ACM, 2010, p. 35–42. [217]

[CT00] COLLBERG, C. et THOMBORSON, C. « Watermarking, tamper-proofing, and obfuscation-tools for software protection ». In : *Software Engineering, IEEE Transactions on* 28 (2000), p. 735–746.

[CTL97] COLLBERG, C., THOMBORSON, C. et LOW, D. *A Taxonomy of Obfuscating Transformations*. Technical Report 148. Department of Computer Science, University of Auckland, New Zealand, juil. 1997. [176, 202, 209]

[CTL98a] COLLBERG, C., THOMBORSON, C. et LOW, D. « Breaking Abstractions and Unstructuring Data Structures ». In : *IEEE International Conference on Computer Languages*. Chicago, mai 1998, p. 28–38. [209]

[CTL98b] COLLBERG, C., THOMBORSON, C. et LOW, D. « Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs ». In : *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA : ACM, 1998, p. 184–196. [212]

[Dra04] DRAPE, S. « Obfuscation of Abstract Data Types ». PhD thesis. St John's College, University of Oxford, 2004. [208]

[Dra06] DRAPE, S. « Generalising the array split obfuscation. » In : *Information Science* 177.1 (2006), p. 202–219. [209]

[DZ12] DELAHAYE, J.-P. et ZENIL, H. « Numerical evaluation of algorithmic complexity for short strings : A glance into the innermost structure of randomness. » In : *Applied Mathematics and Computation* 219.1 (2012), p. 63–77. [194]

[FMP14] FEIST, J., MOUNIER, L. et POTET, M.-L. « Statically detecting use after free on binary code. » In : *J. Computer Virology and Hacking Techniques* 10.3 (2014), p. 211–217. [169]

[Fog16] FOG, A. *Instruction tables : Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. 2016. URL : http://www.agner.org/optimize/instruction_tables.pdf. [64]

[GGHRSW13] GARG, S., GENTRY, C., HALEVI, S., RAYKOVA, M., SAHAI, A. et WATERS, B. « Candidate indistinguishability obfuscation and functional encryption for all circuits ». In : *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE. 2013, p. 40–49. [207]

[GGHW14] GARG, S., GENTRY, C., HALEVI, S. et WICHS, D. « On the Implausibility of Differing-Inputs Obfuscation and Extractable Witness Encryption with Auxiliary Input. » In : *CRYPTO (1)*. Sous la dir. de GARAY, J. A. et GENNARO, R. T. 8616. Lecture Notes in Computer Science. Springer, 2014, p. 518–535. [207]

[GM82] GOLDWASSER, S. et MICALI, S. « Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information ». In : *STOC*. Sous la dir. de LEWIS, H. R., SIMONS, B. B., BURKHARD, W. A. et LANDWEBER, L. H. ACM, 1982, p. 365–377. [203]

[GMQ07] GOUBIN, L., MASEREEL, J.-M. et QUISQUATER, M. « Cryptanalysis of white box DES implementations ». In : *Selected Areas in Cryptography*. Springer. 2007, p. 278–295. [156]

[GR07] GOLDWASSER, S. et ROTHBLUM, G. N. « On Best-Possible Obfuscation. » In : sous la dir. de VADHAN, S. P. T. 4392. Lecture Notes in Computer Science. Springer, 2007, p. 194–213. [206, 207]

[GR14] GOLDWASSER, S. et ROTHBLUM, G. N. « On Best-Possible Obfuscation. » In : *Journal of Cryptology* 27.3 (2014), p. 480–505. [206]

[Had00] HADA, S. « Zero-Knowledge and Code Obfuscation. » In : *ASIACRYPT*. Sous la dir. d'OKAMOTO, T. T. 1976. Lecture Notes in Computer Science. Springer, 2000, p. 443–457. [203]

[Hal77] HALSTEAD, M. H. *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA : Elsevier Science Inc., 1977. ISBN : 0444002057. [177]

[HK81] HENRY, S. et KAFURA, D. « On the Improvements of Cyclomatic Complexity Metric ». In : *IEEE Transactions on Software Engineering* SE-7.5 (sept. 1981). [186]

[HM81] HARRISON, W. A. et MAGEL, K. I. « A complexity measure based on nesting level ». In : *SIGPLAN Not.* 16.3 (mar. 1981), p. 63–74. [183, 184]

[HU79] HOPCROFT, J. E. et ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusets, USA : Adison-Wesley Publishing Company, 1979. [190]

[IPS15] ISHAI, Y., PANDEY, O. et SAHAI, A. « Public-Coin Differing-Inputs Obfuscation and Its Applications. » In : *TCC (2)*. Sous la dir. de DODIS, Y. et NIELSEN, J. B. T. 9015. Lecture Notes in Computer Science. Springer, 2015, p. 668–697. [207]

[JRWM15] JUNOD, P., RINALDINI, J., WEHRLI, J. et MICHIELIN, J. « Obfuscator-LLVM – Software Protection for the Masses ». In : *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*. Sous la dir. de WYSEUR, B. IEEE, 2015, p. 3–9. DOI : 10.1109/SPRO.2015.10. [226]

[KRVV04] KRÜGEL, C., ROBERTSON, W. K., VALEUR, F. et VIGNA, G. « Static Disassembly of Obfuscated Binaries. » In : *USENIX Security Symposium*. Sous la dir. de BLAZE, M. USENIX, 2004, p. 255–270. [218]

[Lan92] LANDI, W. « Undecidability of static analysis ». In : *ACM Lett. Program. Lang. Syst.* 1.4 (déc. 1992), p. 323–337.

[LD03a] LINN, C. et DEBRAY, S. « Obfuscation of executable code to improve resistance to static disassembly ». In : *Proceedings of the 10th ACM conference on Computer and communications security*. CCS '03. Washington D.C., USA : ACM, 2003, p. 290–299.

[LD03b] LINN, C. et DEBRAY, S. K. « Obfuscation of executable code to improve resistance to static disassembly. » In : *ACM Conference on Computer and Communications Security*. Sous la dir. de JAJODIA, S., ATLURI, V. et JAEGER, T. ACM, 2003, p. 290–299. [218]

[LDK03] LINN, C., DEBRAY, S. et KECECIOGLU, J. « Enhancing software tamper-resistance via stealthy address computations ». In : *In Proceedings of the 19th Annual Computer Security Applications Conference*. IEEE. 2003.

[Lev74] LEVIN, L. A. « Laws of information conservation (nongrowth) and aspects of the foundation of probability theory ». In : *Problems of Information Transmission* 10 (1974), p. 206–210. [194]

[LK07] LÁSZLÓ, T. et KISS, Á. « Obfuscating C++ Programs via Control Flow Flattening ». In : *Proceedings of the 10th Symposium on Programming Languages and Software Tools (SPLST 2007)*. Dobogókő, Hungary, 2007, 15–29. [215]

[MASB05] MADOU, M., ANCKAERT, B., SUTTER, B. D. et BOSSCHERE, K. D. « Hybrid static-dynamic attacks against software protection mechanisms. » In : *Digital Rights Management Workshop*. Sous la dir. de SAFAVI-NAINI, R. et YUNG, M. ACM, 2005, p. 75–82. [218]

[McC76] MCCABE, T. J. « A complexity measure ». In : *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. San Francisco, California, United States : IEEE Computer Society Press, 1976. [179–181]

[MDT07] MAJUMDAR, A., DRAPE, S. J. et THOMBORSON, C. D. « Slicing obfuscations : design, correctness, and evaluation ». In : *Proceedings of the 2007 ACM workshop on Digital Rights Management*. DRM '07. New York, NY, USA : ACM, 2007, p. 70–81. [189]

[Mea55] MEALY, G. H. « A Method for Synthesizing Sequential Circuits ». In : *Bell System Technical Journal* 34.5 (1955), p. 1045–1079. [208]

[Mil72] MILLS, H. D. *Mathematical foundations for structured programming*. report FSC 72-6012. IBM Federal Systems Division, Gaithersburg, Md, 1972. [180]

[MJ05] MYLES, G. et JIN, H. « Self-validating Branch-based Software Watermarking ». In : *Proceedings of the 7th International Conference on Information Hiding*. IH'05. Barcelona, Spain : Springer-Verlag, 2005, p. 342–356. [218]

[MPL04] MASRI, W., PODGURSKI, A. et LEON, D. « Detecting and Debugging Insecure Information Flows. » In : *ISSRE*. IEEE Computer Society, 2004, p. 198–209. [169]

[Mye77] MYERS, G. J. « An extension to the cyclomatic measure of program complexity ». In : *SIGPLAN Not.* 12.10 (1977), p. 61–64. [181]

[MZK13] MADI, A., ZEIN, O. K. et KADRY, S. « On the Improvements of Cyclomatic Complexity Metric ». In : *International Journal of Software Engineering and its Applications* 7.2 (mar. 2013). [181]

[OSSM03] OGISO, T., SAKABE, Y., SOSHI, M. et MIYAJI, A. « Software Obfuscation on a Theoretical Basis and Its Implementation ». In : *IEICE Trans Fundam Electron Commun Comput Sci* E86-A.1 (2003), p. 176–186. [190, 218]

[OT93] OTT, L. M. et THUSS, J. J. « Slice Based Metrics for Estimating Cohesion ». In : *In Proceedings of the IEEE-CS International Metrics Symposium*. IEEE Computer Society Press, 1993, p. 71–81. [187]

[PMBG06] PREDA, M. D., MADOU, M., BOSSCHERE, K. D. et GIACOBAZZI, R. « Opaque Predicates Detection by Abstract Interpretation. » In : *AMAST*. Sous la dir. de JOHNSON, M. et VENE, V. T. 4019. Lecture Notes in Computer Science. Springer, 2006, p. 81–95. [167]

[Rad62] RADO, T. « On Non-Computable Functions ». In : *Bell System Technical Journal* 41.3 (1962), p. 877–884. [194]

[Ram94] RAMALINGAM, G. « The undecidability of aliasing ». In : *ACM Trans. Program. Lang. Syst.* 16.5 (sept. 1994), p. 1467–1471. ISSN : 0164-0925. [190]

[SS99] SHAMIR, A. et SOMEREN, N. van. « Playing Hide and Seek with Stored Keys. » In : *Financial Cryptography*. Sous la dir. de FRANKLIN, M. K. T. 1648. Lecture Notes in Computer Science. Springer, 1999, p. 118–124. [208]

[WHH79] WOODWARD, M. R., HENNELL, M. A. et HEDLEY, D. « A Measure of Control Flow Complexity in Program Text. » In : *IEEE Trans. Software Eng.* 5.1 (1979), p. 45–50. [185]

[WHKD00] WANG, C., HILL, J., KNIGHT, J. et DAVIDSON, J. *Software Tamper Resistance : Obstructing Static Analysis of Programs*. Rapp. tech. Charlottesville, VA, USA, 2000. [190, 215]

[WHKD01] WANG, C., HILL, J., KNIGHT, J. C. et DAVIDSON, J. W. « Protection of Software-Based Survivability Mechanisms ». In : *Proceedings of the 2001 International Conference on Dependable Systems and Networks*. DSN '01. Washington, DC, USA : IEEE Computer Society, 2001, p. 193–202. [216]

[Yao82] YAO, A. C.-C. « Theory and Applications of Trapdoor Functions (Extended Abstract) ». In : *FOCS*. IEEE Computer Society, 1982, p. 80–91. [203]

[ZSTDG12] ZENIL, H., SOLER-TOSCANO, F., DELAHAYE, J.-P. et GAUVRIT, N. « Two-Dimensional Kolmogorov Complexity and Validation of the Coding Theorem Method by Compressibility ». In : *CoRR* abs/1212.6745 (2012). [194]

[ZSTDL13] ZENIL, H., SOLER-TOSCANO, F., DINGLE, K. et LOUIS, A. A. « Graph Automorphism and Topological Characterization of Synthetic and Natural Complex Networks by Information Content. » In : *CoRR* abs/1306.0322 (2013). [193, 195, 239]

[ZTW06] ZHU, W., THOMBORSON, C. D. et WANG, F.-Y. [209]
« Obfuscate arrays by homomorphic functions. » In : *GrC*. IEEE, 2006, p. 770–773.

Conclusion générale

Dans ce mémoire, nous avons répondu aux deux problématiques soulevées dans l'introduction générale :

- **Mettre en place une implantation performante et robuste d'un mécanisme cryptographique sur des cibles logicielle et matérielle**
- **Comment peut-on sécuriser ce mécanisme et empêcher qu'il ne soit corrompu au point qu'il ne remplisse plus sa fonction de protection ?**

Dans la première partie, nous avons étudié les implantations logicielle et matérielle de l'arithmétique des courbes elliptiques. Nous nous sommes en particulier concentrés sur une forme de courbe permettant d'obtenir une opération unifiée pour l'addition et le doublement de points : les courbes quartiques de Jacobi étendues. Cette propriété permet d'avoir une résistance par rapport aux attaques par canaux cachés de type SPA. Pour la performance des implantations, nous avons analysé les différentes stratégies d'arithmétique et de parallélisation pour l'implantation de $\mathbb{F}_p$, $\mathbb{F}_{p^2}$ et $\mathbb{F}_{p^3}$. Nous avons développé une implantation de génération de courbes elliptiques sous différentes formes, qui est paramétrable en fonction des tests et des algorithmes de génération que l'utilisateur souhaite utiliser. Notre contribution à la bibliothèque `MPHELL` supporte différentes formes de courbes et a été conçue en terme de flexibilité et d'inter-opérabilité avec d'autres bibliothèques comme `GMP` et `PARI/GP`. Elle fournit une implantation performante des opérations ciblées. Notre implantation matérielle s'appuie sur les parallélisations d'algorithmes étudiées ainsi que sur une multiplication modulaire basée sur l'utilisation d'unités de calcul performantes, les DSPs. Elle permet également de réaliser une addition de points pour une courbe quartique de Jacobi définie sur $\mathbb{F}_{p^3}$ avec un meilleur coût qu'une implantation classique Weierstrass en coordonnées jacobiennes, ce qui nous permet de traiter les cas des courbes ne pouvant se mettre sous cette forme sur $\mathbb{F}_p$.

Dans la seconde partie, nous avons présenté nos travaux sur l'obscurcissement de code source. Nous avons étudié différentes techniques d'analyse de programme et certaines mesures de complexité. Cela nous a permis d'identifier des techniques qui pourraient complexifier un code, comme par exemple l'ajout de pointeurs. Nous avons contri-

bué à l'implantation d'un outil d'obscurcissement spécialisé dans le code `C` en implantant différentes transformations mais également un module de calcul de complexité qui nous permet d'évaluer la puissance de celles-ci. Il permet également vérifier certains effet de la compilation en comparant notamment la complexité du code source à celle du binaire après compilation. Nous avons également étudié l'ordre d'application des transformations en fonction des contraintes liées à aux techniques et à l'impact sur le temps d'exécution du code obscurci. Des travaux sont en cours concernant les optimisations possibles sur un code obscurci qui ne détruisent pas les modifications du code qui ont été opérées.

Résumé

La protection des mécanismes cryptographiques constitue un enjeu important lors du développement d'un système d'information car ils permettent d'assurer la sécurisation des données traitées. Les supports utilisés étant à la fois logiciels et matériels, les techniques de protection doivent s'adapter aux différents contextes.

Dans le cadre d'une cible logicielle, des moyens légaux peuvent être mis en oeuvre afin de limiter l'exploitation ou les usages. Cependant, il est généralement difficile de faire valoir ses droits et de prouver qu'un acte illicite a été commis. Une alternative consiste à utiliser des moyens techniques, comme l'obscurcissement de code, qui permettent de complexifier les stratégies de rétro-conception en modifiant directement les parties à protéger. Concernant les implantations matérielles, on peut faire face à des attaques passives (observation de propriétés physiques) ou actives, ces dernières étant destructives. Il est possible de mettre en place des contre-mesures mathématiques ou matérielles permettant de réduire la fuite d'information pendant l'exécution de l'algorithme, et ainsi protéger le module face à certaines attaques par canaux cachés.

Les travaux présentés dans ce mémoire proposent nos contributions sur ces sujets tes travaux. Nous étudions et présentons les implantations logicielle et matérielle réalisées pour le support de courbes elliptiques sous forme quartique de Jacobi étendue. Ensuite, nous discutons des problématiques liées à la génération de courbes utilisables en cryptographie et nous proposons une adaptation à la forme quartique de Jacobi étendue ainsi que son implantation. Dans une seconde partie, nous abordons la notion d'obscurcissement de code source. Nous détaillons les techniques que nous avons implantées afin de compléter un outil existant ainsi que le module de calcul de complexité qui a été développé.

Mots clés : Offuscation de codes • Mesure de complexité • Mécanismes cryptographiques • Arithmétique des courbes elliptiques • Opérateurs arithmétiques sur les corps finis • Implantation haute performance logicielle et matérielle (FPGA)

Abstract

The protection of cryptographic mechanisms is an important challenge while developing a system of information because they allow to ensure the security of processed data. Since both hardware and software supports are used, the protection techniques have to be adapted depending on the context.

For a software target, legal means can be used to limit the exploitation or the use. Nevertheless, it is in general difficult to assert the rights of the owner and prove that an unlawful act had occurred. Another alternative consists in using technical means, such as code obfuscation, which make the reverse engineering strategies more complex, modifying directly the parts that need to be protected. Concerning hardware implementations, the attacks can be passive (observation of physical properties) or active (which are destructive). It is possible to implement mathematical or hardware countermeasures in order to reduce the information leakage during the execution of the code, and thus protect the module against some side channel attacks.

In this thesis, we present our contributions on theses subjects. We study and present the software and hardware implementations realised for supporting elliptic curves given in Jacobi Quartic form. Then, we discuss issues linked to the generation of curves which can be used in cryptography, and we propose an adaptation to the Jacobi Quartic form and its implementation. In a second part, we address the notion of code obfuscation. We detail the techniques that we have implemented in order to complete an existing tool, and the complexity module which has been developed.

Keywords : Code obfuscation • Complexity measure • Cryptographic mechanisms • The arithmetic of Elliptic Curves • Arithmetic operators on finite fields • High performance software and hardware implementation