Problème du logarithme discret sur des courbes elliptiques

Andy Russon

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Mathématiques et leurs interactions*

Par
Andy RUSSON

Problème du logarithme discret sur des courbes elliptiques

Thèse présentée et soutenue à Rennes, le 28 janvier 2022
Unité de recherche : IRMAR, (UMR CNRS 6625), Rennes

Rapporteurs avant soutenance :

Louis GOUBIN	Professeur, Université de Versailles-Saint-Quentin-en-Yvelines, Versailles
Damien VERGNAUD	Professeur, Sorbonne Université, Paris

Composition du jury :

Directeur de thèse	Sylvain DUQUESNE	Professeur, Université de Rennes 1, Rennes
Rapporteur	Louis GOUBIN	Professeur, Université de Versailles-Saint-Quentin-en-Yvelines
Examinatrice	Cécile PIERROT	Chargée de recherche, INRIA, Nancy
Examinateur	Arnaud TISSERAND	Directeur de recherche, CNRS, Lorient
Rapporteur	Damien VERGNAUD	Professeur, Sorbonne Université, Paris
Co-encadrant de thèse	Olivier VIVOLO	Directeur, Orange, Cesson-Sévigné

Problème du logarithme discret sur des courbes elliptiques

Andy Russon

Introduction

La cryptographie est une science qui fournit un ensemble d'outils pour la sécurité de l'information pour assurer la confidentialité, l'intégrité des données et l'authentification. Dans cette thèse on s'intéresse en particulier à la cryptographie à base de courbes elliptiques et s'intitule *Problème du logarithme discret sur des courbes elliptiques*. Le titre fait référence au problème mathématique du même nom sur lequel repose la sécurité de la cryptographie à base de courbes elliptiques. En particulier, cela fait partie du sous-ensemble de la cryptographie asymétrique aussi appelée cryptographie à clé publique.

Cryptographie à clé publique

Une communication sécurisée entre deux parties, Alice et Bob, nécessite la connaissance d'un clé secrète commune utilisée avec un algorithme de chiffrement symétrique. Cela soulève la problématique de distribution des clés, car leur nombre peut devenir élevé. En effet, si une troisième personne souhaite communiquer avec Alice sans que Bob ne puisse être en mesure de déchiffrer, une clé distincte est nécessaire. Il faut alors une clé pour chaque paire d'utilisateurs possibles. C'est dans l'article *New Directions in Cryptography* [DH76] que Diffie et Hellman proposent une solution par l'utilisation de clés publiques pour créer un secret partagé. Chaque utilisateur n'a alors besoin de conserver que sa propre clé.

Un exemple concret est donné, où les calculs sont effectués dans un corps premier $\mathbf{F}_p$ avec g un élément primitif (il génère le groupe multiplicatif du corps). D'un côté Alice choisit une valeur a et calcule $A = g^a$, et Bob fait de même avec une valeur b et calcule $B = g^b$. Alice et Bob s'échangent les valeurs A et B (leurs clés publiques) et peuvent désormais calculer une valeur commune :

$$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b.$$

Il est impératif que les valeurs a et b restent secrètes (ce sont les clés privées d'Alice et Bob), car en cas contraire, une personne tierce qui intercepte A et B serait en mesure de calculer le secret partagé. De plus, il est aussi nécessaire que a et b ne puissent être facilement calculés à partir des valeurs A et B : obtenir la clé privée à partir de la clé publique s'appelle le *problème du logarithme discret*.

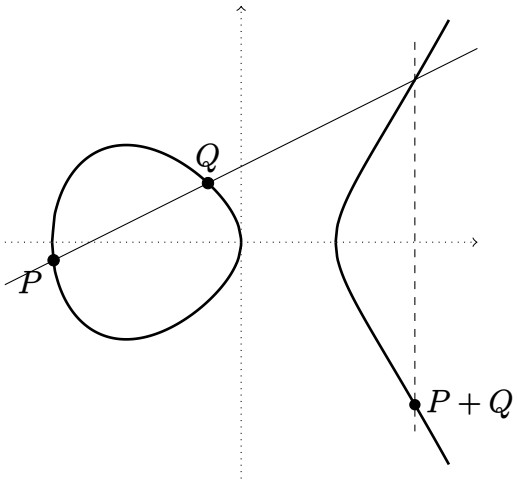
La difficulté de ce problème dépend du choix des paramètres et les travaux de recherches ont sans cesse amélioré les techniques de résolution. En conséquence il est actuellement recommandé d'utiliser des nombres premiers p de 2048 ou 3072 bits. Le choix d'un corps aussi grand tient au fait que les meilleurs méthodes de résolution sont

similaires à celles utilisées pour la factorisation, d'où des paramètres de tailles semblables par rapport à la cryptographie RSA. La complexité de ces méthodes est *sous-exponentielle* en la taille des paramètres, donc ces derniers augmentent plus vite que le niveau de sécurité.

Le partage de clé Diffie-Hellman décrit ci-dessus repose sur la notion de groupe fini et n'est pas exclusif à un groupe en particulier. Il est ainsi possible d'utiliser des groupes dont les paramètres sont davantage compacts, ce qui a comme conséquence des clés plus courtes (pour le stockage et l'échange), et potentiellement une meilleure efficacité dans les calculs. De toute évidence, il faut s'assurer que le problème du logarithme discret reste difficile à résoudre.

Courbes elliptiques

L'usage des courbes elliptiques en cryptographie a été proposé par Miller [Mil85] et Koblitz [Kob87] dans les années 1980. Ces courbes sont dotées d'une structure de groupe abélien, par une construction géométrique de la loi de groupe, illustré dans la figure ci-dessous.



Depuis leur introduction, il a été montré qu'en dehors de certaines classes de courbes, les meilleures méthodes de résolution du problème du logarithme discret sont les algorithmes génériques, c'est-à-dire qui s'appliquent à n'importe quel groupe fini abélien. La complexité de ces méthodes est en la racine carrée de la taille du groupe, donc une courbe elliptique avec des paramètres de taille 200 bits nécessiterait une capacité de calculs d'environ 2^{100} afin de résoudre un logarithme discret. En bref, la structure de groupe sur une courbe elliptique apparaît comme un candidat idéal.

Les courbes elliptiques ont longuement été étudiées et peuvent prendre plusieurs formes (données par différentes équations). Par conséquent, la loi de groupe peut être donnée par de nombreuses formules algébriques, enrichies par l'usage de plusieurs systèmes de coordonnées pour représenter les points de la courbe. Cette richesse peut devenir une difficulté pour son utilisation en cryptographie. En effet, les développeurs doivent faire des choix parmi un ensemble de combinaisons possibles dans les paramètres, formules ou algorithmes, et cela peut être critique d'un point de vue sécurité de l'implémentation réalisée.

Les attaques d'implémentations

Si le problème du logarithme discret est difficile à résoudre en un temps raisonnable, des moyens alternatifs sont possibles par la récupération de données auxiliaires. C'est notamment le cas si de l'information relative aux valeurs secrètes comme la clé privée fuite par l'appareil qui exécute le code cryptographique.

On peut distinguer deux grandes catégories d'attaquants.

Espion passif. Ce type d'attaquant exploite les données obtenues en observant les fuites occasionnées lors de l'exécution. Le premier exemple est l'analyse du temps d'exécution du code, lorsque celui-ci dépend de la valeur secrète manipulée. Il y a aussi la possibilité d'exploiter la consommation de courant ou des émanations électromagnétiques, dont la corrélation avec les opérations réalisées permettent de déduire éventuellement les valeurs secrètes. La version classique est *Simple Power Analysis* (SPA), où le comportement de l'exécution peut s'observer sur une seule trace. Une version plus complexe est *Differential Power Analysis* (DPA) où plusieurs traces sont utilisées lorsqu'un secret constant est manipulé.

Espion actif. Contrairement au précédent, ce type d'attaquant interagit avec l'exécution. Cela inclut les attaques par entrées invalides, où l'attaquant choisit des valeurs qui produisent un comportement qui n'a pas été anticipé. Mais il y a aussi les attaques par injection de fautes. Le principe est de perturber le bon déroulement de l'exécution de telle sorte que les effets produits puissent révéler tout ou une partie des données secrètes. Les possibilités sont variées et peuvent être exploitées de différentes façons. Il y a les attaques par altérations de paramètres : pour les courbes elliptiques cela peut amener les opérations à être réalisées avec des paramètres où le problème du logarithme discret est plus facile à résoudre. Les attaques en faute différentielle, ou *Differential Fault Analysis* (DFA) consistent à produire une faute telle que la différence entre le résultat erroné obtenu et le résultat correct révèle des valeurs secrètes. Enfin, il y a les attaques *Safe-Error* dont le principe repose sur l'absence d'effet de la faute sur le résultat final afin d'extraire des données secrètes.

Contributions

Les résultats présentés dans cette thèse sont orientés sur la sécurité des implémentations de courbes elliptiques. Ces travaux ont été réalisés par une analyse du code source de plusieurs bibliothèques cryptographiques au regard de critères de sécurités liés aux différentes attaques citées plus haut, notamment par l'identification des choix effectués par les développeurs : algorithmes, formules, mécanismes de protection, etc.

Cela a abouti à la publication des quatre articles présentés ci-dessous.

[Rus20] Andy Russon, "Exploiting dummy codes in Elliptic Curve Cryptography implementations", in: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2020*, 2020, pp. 229–249.

Cet article montre qu'une attaque par injection de faute *Safe-Error* est possible sur plusieurs implémentations de courbes elliptiques dans des bibliothèques cryptographiques.

Pour une protection contre des attaques par un espion passif, des calculs factices sont introduits afin que l'exécution soit régulière quelle que soit la valeur secrète en entrée. Plus spécifiquement, il s'agit d'implémentations qui traitent la valeur secrète par morceaux de plusieurs bits consécutifs. Lorsque tous ces bits sont nuls, les opérations effectuées (calculs factices) n'ont aucun impact sur le résultat final. Une faute lors de ce moment n'a alors pas d'effet et révèle ainsi que les bits secrets valent zéro lors de ce moment.

[Rus21a] Andy Russon, "Return of ECC dummy point additions: Simple Power Analysis on efficient P-256 implementation", in: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 2-4 2021*, 2021, pp. 367–375.

En lien avec le précédent, cet article montre qu'une implémentation en particulier (et présente dans plusieurs bibliothèques) est vulnérable à un cas particulier d'attaque SPA. Cela est lié à la façon dont les calculs factices sont réalisés. Ceux-ci utilisent essentiellement des valeurs nulles et l'impact sur la consommation de courant peut être observé sur une trace.

[Rus21b] Andy Russon, "Threat for the Secure Remote Password Protocol and a Leak in Apple's Cryptographic Library", in: *ACNS 2021*, ed. by Kazue Sako and Nils Ole Tippenhauer, vol. 12727, LNCS, Springer, 2021, pp. 49–75, DOI: 10.1007/978-3-030-78375-4_3.

Dans ce troisième article présenté à la conférence *Applied Cryptography and Network Security*, on propose une attaque par dictionnaire sur le protocole d'authentification Secure Remote Password (SRP). Ce protocole est une construction similaire à un échange de clé Diffie-Hellman, mais en faisant intervenir un mot de passe. On y présente comment réaliser une attaque en exploitant une fuite éventuelle lors de l'exponentiation modulaire. Celle-ci fait notamment intervenir une clé privée générée de façon déterministe depuis un mot de passe et l'exploitation de la fuite permet d'obtenir un indicateur pour filtrer les potentiels mots de passe dans un dictionnaire.

On montre également une vulnérabilité relevée dans la bibliothèque cryptographique d'Apple qui peut être exploitée dans le contexte de SRP. Une méthode de protection contre les attaques DPA est présente : la clé privée est randomisée à chaque exécution par une division Euclidienne. C'est l'usage de cette méthode qui introduit une variabilité qui peut être mesurable par une analyse simple de consommation. Le recueil d'un grand nombre de mesures permet de faire une approximation de la clé privée.

Cette vulnérabilité concerne également les calculs sur courbes elliptiques et son exploitation est possible sur le protocole d'authentification par mot de passe SPAKE2+ qui est similaire à SRP.

La vulnérabilité a été transmise à Apple Product Security selon leur procédure. Des correctifs ont été introduits au Printemps 2021 à la suite des communications avec leurs ingénieurs.

[Rus21c] Andy Russon, "Differential Fault Attack on Montgomery Ladder and in the Presence of Scalar Randomization", in: *INDOCRYPT 2021, Proceedings*, ed. by Avishek Adhikari, Ralf Küsters, and Bart Preneel, vol. 13143, LNCS, Springer, 2021, pp. 287–310, DOI: 10.1007/978-3-030-92518-5_14.

Ce dernier article, présenté à la conférence *Indocrypt*, porte sur une attaque en faute différentielle. La première contribution est une nouvelle proposition d'attaque DFA sur l'algorithme de multiplication scalaire Montgomery ladder avec une faute liée à l'échange conditionnel des points manipulés au cours de l'exécution. Un regard particulier est portée vers l'usage de formules d'addition de points spécifiques, où certaines vérification peuvent ne pas suffire pour détecter la faute.

La seconde contribution principale de l'article met en avant que certaines méthodes de randomisation pour protéger contre les attaques DPA ne sont pas nécessairement suffisantes pour protéger contre des attaques DFA.

Présentation du plan

Le **chapitre 1** donne une définition des courbes elliptiques et présente les différentes formes les plus usuelles en cryptographie, ainsi que les représentations de coordonnées. Des protocoles utilisant des courbes elliptiques sont également présentés.

Le **chapitre 2** introduit le *problème du nombre caché* ou *Hidden Number Problem* (HNP) et les méthodes à base de réseau Euclidien pour le résoudre. Cela a une utilité pour, par exemple, exploiter des vulnérabilités pendant la génération de signatures, comme montré dans les chapitres 4 et 6.

Le **chapitre 3** étudie les formules courantes d'additions de points et les cas exceptionnels. Plusieurs algorithmes de multiplication scalaire sont présentés et on précise si ces exceptions peuvent avoir lieu ou non.

Le **chapitre 4** s'intéresse à la situation d'additions factices de points, qu'elles soient implicitement ou explicitement introduites au cours de l'exécution du code. Le concept de l'attaque par un espion actif perturbant le bon déroulement des calculs est présentée avec une sélection de bibliothèques cryptographiques vulnérables. Pour l'une d'entre elle, on montre également qu'un espion passif peut être en mesure de détecter une partie de ces additions factices.

Le **chapitre 5** concerne une méthode de randomisation du scalaire pour laquelle une vulnérabilité nouvelle est présentée. Telle qu'introduite originellement (ainsi que dans son utilisation dans la bibliothèque cryptographique d'Apple), il est possible d'exploiter une légère variation d'exécution aboutissant à l'approximation du secret.

Le **chapitre 6** détaille une nouvelle attaque en faute différentielle appliquée sur l'algorithme Montgomery ladder, et explore également comment les méthodes de randomisation du scalaire peuvent être insuffisantes pour protéger contre ce type d'attaque.

Le **chapitre 7** présente une attaque par dictionnaire contre SRP et SPAKE2+, deux protocoles d'authentification basée sur un mot de passe.

Table des matières

1

Cryptographie à base de courbes elliptiques

Dans ce chapitre on introduit les définitions et notations sur les courbes elliptiques qui sont essentielles pour les chapitres suivants. On y présente notamment la loi de groupe et quelques propriétés importantes. On termine avec des primitives cryptographiques basées sur les courbes elliptiques.

Sommaire

1.1 Courbes elliptiques

Dans cette section on note K un corps fini, donc de cardinal une puissance d'un nombre premier. Plus particulièrement le corps premier $\mathbf{F}_p$ sera considéré.

1.1.1 Définition

Notons tout d'abord $\mathbf{P}^2(K)$ le plan projectif de dimension 2 sur le corps K. Il s'agit de l'ensemble des classes notées $(X : Y : Z)$ où X, Y, et Z sont dans K non tous nuls, selon la relation d'équivalence $(X, Y, Z) \sim (X', Y', Z')$ s'il existe λ non nul sur K tel que $(X, Y, Z) = (\lambda X', \lambda Y', \lambda Z')$.

Un point $(X : Y : Z)$ avec $Z \neq 0$ est identifié dans le plan affine par le point (x, y) où $x = X/Z$ et $y = Y/Z$. Les points $(X : Y : 0)$ sont appelés les *points à l'infini*.

Définition 1. Une *courbe elliptique* sur un corps K de caractéristique différente de 2 et 3 est une courbe projective sur $\mathbf{P}^2(K)$ donnée par l'équation de Weierstraß courte

$$Y^2Z = X^3 + AXZ^2 + BZ^3, \tag{1.1}$$

où les paramètres A et B sont tels que le discriminant $\Delta = 4A^3 + 27B^2$ est non nul. On note $E(K)$ l'ensemble des points qui satisfont l'équation de cette courbe.

Il est assez facile de voir que le seul point dont la coordonnée projective Z est nulle est $(0 : 1 : 0)$. Il ne possède donc pas de représentation affine et on le note $\mathcal{O}$. Ainsi on peut aussi définir la courbe par son équation sous forme affine et l'ensemble des points est :

$$E(K) = \{\, (x, y) \in K^2 \mid y^2 = x^3 + Ax + B \,\} \cup \{\, \mathcal{O} \,\}. \tag{1.2}$$

La condition spécifique sur le discriminant Δ est pour éviter la présence de points dits *singuliers* (on dit alors que la courbe est lisse). Dans le cas où celui-ci est effectivement nul, alors le polynôme $x^3 + Ax + B$ possède une racine double ou triple. La raison de l'exclusion de ce cas dans la définition est que la loi de groupe définie ci-dessous serait facilement réécrite sous la forme d'une addition ou d'une multiplication dans le corps K (éventuellement dans une extension de degré 2), ce qui ne serait que peu d'intérêt pour une utilisation en cryptographie.

1.1.2 Loi de groupe

Soit P et Q deux points sur une courbe elliptique E. On définit la somme $P + Q$ par la construction géométrique en deux étapes donnée ci-dessous avec le point $\mathcal{O}$ comme élément neutre.

1. Soit L la droite passant par les points P et Q. Cette droite intersecte la courbe en un troisième point R.
2. Soit V la droite passant par les points R et $\mathcal{O}$. Cette droite intersecte la courbe en un troisième point, il s'agit de $P + Q$.

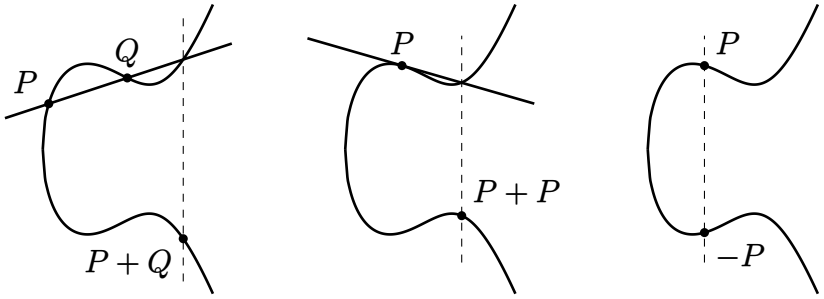


Figure 1.1 : Addition de points sur une courbe elliptique.

Le théorème de Bézout assure que le nombre de points d'intersection entre une droite et une courbe elliptique donnée par l'équation (1.1) est trois. Ainsi la construction ci-dessus est correcte.

L'élément neutre est bien le point à l'infini $\mathcal{O}$. En effet, pour cela on utilise la version projective des équations, dont celle de la droite L est

$$L : dX + eY + fZ = 0, \quad (d, e, f) \neq (0, 0, 0).$$

Si $Q = \mathcal{O}$, alors l'équation de L devient $dX + fZ = 0$, autrement dit il s'agit d'une droite verticale, dont le troisième point d'intersection est $R = (X_1 : -Y_1 : Z_1)$, symétrique du point $P = (X_1 : Y_1 : Z_1)$. La droite V passant par R et $\mathcal{O}$ est la même et on obtient $P + \mathcal{O} = P$. On observe ainsi au passage que l'inverse d'un point P est le point noté

$$-P = (X_1 : -Y_1 : Z_1).$$

L'associativité est moins évidente à prouver, mais peut être effectuée formellement.

Cependant il convient de regarder quelques cas particuliers, notamment pour la partie calculatoire. Pour calculer $P + P$, la droite L construite est une tangente à la courbe (elle intersecte la courbe avec une multiplicité double au point P). Tandis que pour calculer $P + (-P)$, il s'agit d'une droite verticale. Le calcul de l'équation de la droite nécessite de distinguer ces situations pour calculer sa pente.

Ces cas particuliers sont présentés dans la formulation affine de la loi de groupe ci-dessous, et font également l'objet du chapitre 3 où cela entraîne des cas exceptionnels (selon les formules et algorithmes utilisés) qui nécessitent d'être traités avec soin pour une implémentation sécurisée dans les bibliothèques cryptographiques.

Formules affines. Les différentes situations sont illustrées dans la figure 1.1.

Une reformulation de la construction géométrique avec une représentation affine des points donnent la loi de groupe ci-dessous. Notons $P = (x_1, y_1)$ et $Q = (x_2, y_2)$. La loi de groupe est :

1. $P + \mathcal{O} = \mathcal{O} + P = P$ pour tout P sur la courbe.
2. Si $x_1 = x_2$ et $y_1 = -y_2$ alors $P + Q = \mathcal{O}$.
3. Sinon, si $x_1 \neq x_2$ on pose

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

et si $x_1 = x_2$ alors on pose

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

Alors $y = \lambda(x - x_1) + y_1$ est l'équation de la droite L passant par P et Q (ou de la tangente si $P = Q$) et $P + Q = (x_3, y_3)$ est donné par

$$\begin{cases} x_3 = \lambda^2 - (x_1 + x_2) \\ y_3 = \lambda(x_1 - x_3) - y_1. \end{cases}$$

Multiplication scalaire. Étant donné un point P sur la courbe elliptique et un entier positif k, on note $[k]P$ la multiplication scalaire d'un point P par un entier k :

$$[k]P = \underbrace{P + P + \cdots + P}_{k \text{ fois}}.$$

En particulier on a $[0]P = \mathcal{O}$ et pour un entier k négatif on pose :

$$[-k]P = \underbrace{(-P) + (-P) + \cdots + (-P)}_{k \text{ fois}}.$$

Dans la pratique on ne calcule pas une multiplication scalaire par une addition répétée car cela deviendrait vite impossible pour de grands entiers k. Des algorithmes de complexité logarithmique par rapport au scalaire k permettent de réaliser cette opération efficacement. Plusieurs exemples sont donnés dans le chapitre 3.

1.1.3 Autres systèmes de coordonnées

Les systèmes de coordonnées affines et projectives homogènes ont déjà été donnés dans la section 1.1.1. Il existe cependant un autre système de coordonnées projectives souvent utilisé dans les implémentations de courbes elliptiques appelées *coordonnées Jacobiennes*.

Tout point est représenté par la classe $(X : Y : Z)$ mais tel que la représentation affine correspondante est $(x, y) = (X/Z^2, X/Z^3)$. Autrement dit la relation d'équivalence $(X, Y, Z) \sim (X', Y', Z')$ s'il existe λ non nul tel que $(X, Y, Z) = (\lambda^2 X, \lambda^3 Y, \lambda Z)$. Si on réécrit l'équation en injectant cette relation, on obtient

$$Y^2 = X^3 + AXZ^4 + BZ^6.$$

On en déduit qu'une représentation du point à l'infini est donnée par les triplets de la forme $(\lambda^2 : \lambda^3 : 0)$ pour $\lambda \neq 0$.

1.1.4 Formes alternatives

D'autres formes de courbes elliptiques ont été introduites au cours du temps. On présente ci-dessous les deux les plus utilisées en cryptographie en dehors de la forme de Weierstraß courte.

Forme de Montgomery. Cette forme a été introduite par Montgomery [Mon87] pour une utilisation dans la méthode de factorisation à base de courbes elliptiques. Elle est donnée par l'équation

$$By^2 = x^3 + Ax^2 + x,$$

et la condition sur les paramètres A et B est $B(A^2 - 4) \neq 0$.

Le choix de cette forme tient à l'efficacité des formules pour l'addition de points présentées dans le même article, qui n'utilisent pas la coordonnée y. Des formules analogues pour la forme de Weierstraß sont étudiées dans la section **3.1.4**.

Cette forme a été introduite pour un usage en cryptographie dans [Ber06] avec une courbe nommée `Curve25519`, dont le nom fait référence au nombre premier $2^{255} - 19$ qui définit le corps de base.

Forme d'Edwards. La forme d'Edwards a été introduite dans [Edw07]. Celle-ci est donnée par une équation de la forme suivante :

$$x^2 + y^2 = 1 + dx^2y^2,$$

où le paramètre d vérifie $d(1 - d) \neq 0$. Avec le changement de variable $x = u/v$ et $y = (u-1)/(u+1)$ on retrouve l'équation de la forme de Montgomery donnée ci-dessus avec $A = 2(1+d)/(1-d)$ et $B = 4/(1-d)$. Un avantage est que l'ensemble des points de la courbe elliptique ont une représentation affine sur la forme d'Edwards. En particulier, l'élément neutre est le point $(0, 1)$. Un second avantage démontré dans [BL07] est que les formules

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right).$$

pour la loi de groupe sont complètes : elles donnent le bon résultat pour tous points $P = (x_1, y_1)$ et $Q = (x_2, y_2)$.

1.1.5 Cardinalité

Comme le corps de base est fini, la courbe elliptique possède un nombre fini d'éléments, dont l'encadrement est donné par le théorème suivant.

Théorème 1 (Hasse). *Le cardinal d'une courbe elliptique $E(\mathbf{F}_p)$, noté $|E(\mathbf{F}_p)|$ satisfait l'inégalité*

$$p + 1 - 2\sqrt{p} \leq |E(\mathbf{F}_p)| \leq p + 1 + 2\sqrt{p}.$$

Ainsi il existe un plus petit entier N tel que $[N]P = \mathcal{O}$ et est appelé l'ordre de ce point. Il est conseillé que cet ordre soit un grand nombre premier pour un usage en cryptographie, ce qui est justifié dans la section suivante.

La plupart des courbes standardisées sont choisies de cardinal premier et donc tout point (hormis le point à l'infini) génère tout le groupe. Une exception sont les courbes sous forme de Montgomery et d'Edwards qui possèdent des points d'ordre 2, donc le cardinal est nécessairement pair. Avec la forme de Weierstraß courte, ces points sont ceux avec une ordonnée nulle, ce qui est équivalent à une abscisse racine du polynôme $x^3 + Ax + B$. Pour la forme de Montgomery, il y a toujours le point $(0, 0)$ comme point d'ordre 2.

En résumé, le cardinal des courbes elliptiques recommandées en cryptographie est $N = qh$ où q est un nombre premier et h est le cofacteur qui vaut généralement 1, 4 ou 8.

1.2 Problème du logarithme discret

Comme il a été vu dans l'introduction, la sécurité d'un échange de clé Diffie-Hellman repose sur la difficulté de résolution du logarithme discret dans le groupe sous-jacent.

On donne ci-dessous une définition où le groupe est celui associé à une courbe elliptique.

Définition 2. Soit P un point d'ordre N sur une courbe elliptique E définie sur un corps K et Q un point dans le sous-groupe généré par P. Il existe alors un entier k dans l'intervalle $[0, N-1]$ tel que $Q = [k]P$ et est appelé le *logarithme discret* de Q en base P.

On donne dans cette section les algorithmes génériques de résolution du problème du logarithme discret. Le premier justifie le fait de se placer directement dans un groupe d'ordre premier et les suivants sur le choix de la taille de cet ordre.

1.2.1 Algorithme de Pohlig-Hellman

Cet algorithme exploite la factorisation de l'ordre du point P pour se ramener à des logarithmes discrets dans des sous-groupes d'ordre premier [PH78]. Cela se réalise en deux étapes.

Réduction vers un sous-groupe d'ordre une puissance d'un nombre premier. Supposons que la décomposition en facteurs premiers de l'ordre N de P est $q_1^{\alpha_1} \cdots q_s^{\alpha_s}$. On pose alors

$$r_i = \prod_{\substack{j=1 \\ i \neq j}}^{s} q_j^{\alpha_j}, \quad P_i = [r_i]P, \quad Q_i = [r_i]Q.$$

Le nouveau point P_i est d'ordre $q_i^{\alpha_i}$ et on a $Q_i = [k_i]P_i$ où le logarithme discret k_i de Q_i en base P_i satisfait la relation $k_i = k \bmod q_i^{\alpha_i}$.

Lorsque ces logarithmes discrets individuels sont résolus, on obtient le système

$$\begin{cases} k & \equiv & k_1 & (\text{mod } q_1^{\alpha_1}) \\ & \vdots & & \\ k & \equiv & k_s & (\text{mod } q_s^{\alpha_s}), \end{cases}$$

qui peut être résolu par le théorème des restes chinois.

Réduction vers un sous-groupe d'ordre premier. L'étape précédente montre qu'on peut se ramener à un sous-groupe d'ordre une puissance d'un nombre premier. On suppose désormais que P est d'ordre q^α pour un entier q premier et donc $0 \leq k < q^\alpha$. On note la décomposition de k en base q :

$$k = k_0 + k_1 q + k_2 q^2 + \cdots + k_{\alpha-1} q^{\alpha-1},$$

avec $0 \leq k_i < q$ pour $0 \leq i < \alpha$. Les valeurs k_i sont calculées au fur et à mesure pour reconstruire k. En effet, on a

$$q^{\alpha-1} k = k_0 q^{\alpha-1} + q^\alpha (k_1 + k_2 q + \cdots + k_{\alpha-1} q^{\alpha-2}),$$

d'où la multiplication scalaire par $q^{\alpha-1}$ donne la relation

$$P' = [q^{\alpha-1}]P, \quad Q' = [q^{\alpha-1}]Q, \quad Q' = [k_0]P'.$$

Dès lors, k_0 est un logarithme discret dans un groupe d'ordre q.

Maintenant supposons que $k_0, k_1, \ldots, k_{j-1}$ ont été déterminés. On peut retrancher du point Q la partie déjà connue de k :

$$Q' = Q - [\sum_{i=0}^{j-1} k_i q^i]P = [\sum_{i=j}^{\alpha-1} k_i q^i]P.$$

Ensuite on peut itérer le mécanisme précédent avec une multiplication scalaire par $q^{\alpha-j-1}$:

$$P'' = [q^{\alpha-1}]P, \quad Q'' = [q^{\alpha-j-1}]Q', \quad Q'' = [k_j]P''.$$

Ainsi k_j est un logarithme discret dans un groupe d'ordre q.

Combiné avec l'étape précédente, le coût du logarithme discret est donc dominé par le plus grand facteur premier de l'ordre du point de base. Cela justifie le choix d'utiliser des courbes elliptiques dont le cofacteur est petit.

1.2.2 Pas de bébés – pas de géants

Dorénavant, on suppose que le point P est toujours d'ordre premier q. Une recherche exhaustive sur k nécessiterait au plus q opérations. L'idée principale de la méthode *pas de bébé – pas de géants* due à Shanks [Sha71] est de couper la poire en deux : l'entier k est séparé en deux parties de taille inférieures à $\sqrt{q}$ et la recherche est réalisée sur chacune d'elle.

Plus concrètement, posons $m = \lceil \sqrt{q} \rceil$ et notons $k = am + b$ la division Euclidienne de k par m. On calcule le point $R = [m]P$ et l'objectif devient alors de trouver a et b tels que $Q = [a]R + [b]P$. D'un côté on construit la liste des *pas de bébés* :

$$\{ [i]P \mid 0 \leq i < m \}.$$

De l'autre, la liste des *pas de géants*

$$\{ Q - [j]R \mid 0 \leq j < m \}.$$

Une collision entre les deux listes donne le logarithme discret recherché si les indices (i, j) sont conservés :

$$Q - [j]R = [i]P \quad \Rightarrow \quad Q = [i + mj]P \quad \Rightarrow \quad k = i + mj.$$

Une telle solution existe par construction, donc l'algorithme est déterministe et donne la solution en $2\lceil \sqrt{q} \rceil$ opérations dans le groupe au pire cas.

Plusieurs remarques peuvent être faites. Tout d'abord il n'est pas nécessaire de stocker la liste des *pas de géants*, car si un élément n'est pas dans la liste des *pas de bébés* alors il est inutile de le conserver. Cependant, la liste des *pas de bébés* est nécessaire donc cela implique un coût mémoire en $O(\sqrt{q})$.

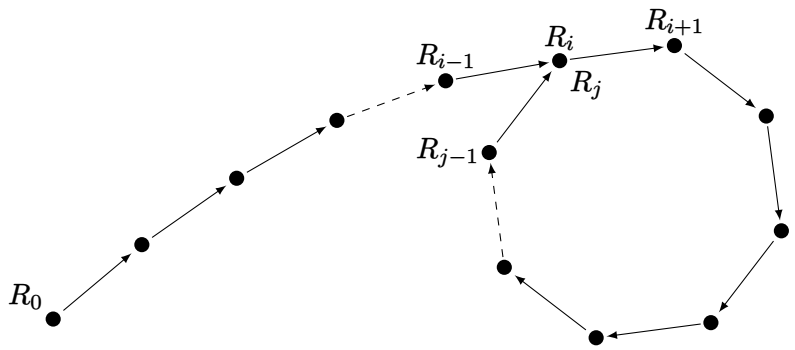


Figure 1.2 : Collision dans l'algorithme *rho* de Pollard.

Recherche dans un sous-intervalle. Il n'est pas toujours nécessaire de faire une recherche sur le groupe entier. En effet, s'il est connu que le logarithme discret k se situe dans un sous-intervalle $[B_1, B_2)$ alors on a la relation

$$\underbrace{Q - [B_1]P}_{Q'} = [k - B_1]P.$$

Ainsi, l'algorithme *pas de bébés – pas de géants* peut être appliqué avec P et Q' en posant $m = \lceil(\sqrt{B_2 - B_1}\rceil$. Son coût est alors $O(\sqrt{B_2 - B_1})$.

1.2.3 L'algorithme *rho* de Pollard

Le principe de cette méthode introduite par Pollard [Pol78] est de construire une suite de points dépendants de P et Q jusqu'à obtenir une collision qui révèle le logarithme discret avec une probabilité élevée.

Deux entiers a_0 et b_0 sont initialement choisis et on pose $R_0 = [a_0]P + [b_0]Q$. Une fonction f agissant comme marche pseudo-aléatoire est utilisée pour construire la suite de points $(R_i)_i$:

$$R_{i+1} = f(R_i) = [a_{i+1}]P + [b_{i+1}]Q.$$

Comme l'ordre des points est un entier fini q, on tombe éventuellement sur une égalité $R_i = R_j$ pour i et j distincts. La suite devient alors une boucle comme illustré dans la figure 1.2 (ce qui donne à la méthode son nom). Alors on a l'égalité

$$[a_i - a_j]P = [b_j - b_i]Q,$$

et, sous condition que $b_j - b_i$ est inversible modulo q, on a

$$k = (a_i - a_j)(b_j - b_i)^{-1} \bmod q.$$

Cependant on n'est pas plus avantagé par rapport à l'algorithme *pas de bébés – pas de géants* car la collision telle que décrite nécessite de sauvegarder les valeurs déjà calculées de la suite. Une méthode de recherche de cycle attribué à Floyd permet de s'affranchir du stockage. Une fois la boucle atteinte, si i est un multiple de la longueur L de la boucle, alors on a l'égalité $R_i = R_{2i}$. Il suffit alors de construire deux suites en parallèle :

$$R_{i+1} = f(R_i), \quad R_{2(i+1)} = f(f(R_{2i})).$$

Dès que ces deux suites sont dans la boucle, on finit par tomber éventuellement sur une collision $R_i = R_{2i}$.

Par le paradoxe des anniversaires, il est estimé que la collision a lieu après $O(\sqrt{\pi q/2})$ étapes (à condition d'une fonction f qui agit comme une marche suffisamment aléatoire), ce qui correspond à la complexité de *pas de bébés – pas de géants*.

Amélioration avec des endomorphismes. Une particularité des courbes elliptiques est la possibilité de réduire le coût de l'algorithme *rho* si des endomorphismes faciles à calculer sont disponibles. Pour toute courbe il y a déjà l'endomorphisme $[-1]$ qui envoie tout point sur son opposé. Pour une courbe sous forme de Weierstraß courte, il est donné par

$$\begin{array}{rccc} [-1]: & E & \longrightarrow & E \\ & (x,y) & \longmapsto & (x,-y). \end{array}$$

Le principe est de regrouper les points sous forme de classes $\{P, -P\}$. Il est facile d'obtenir $-P$ à partir de P, donc cela permet de choisir l'un des deux comme représentant de la classe (par exemple celui dont la coordonnée y est la plus petite en tant qu'entier). L'algorithme *rho* ne se fait plus sur tous les points, mais seulement sur la moitié. La complexité passe alors de $\sqrt{\pi q/2}$ à $\sqrt{\pi q/4}$, soit une réduction d'un facteur $\sqrt{2}$.

Plus généralement, si l'on dispose d'un endomorphisme ϕ d'ordre s, c'est-à-dire tel que pour tout point P et Q on a $\phi(P+Q) = \phi(P) + \phi(Q)$ et $\phi^s(P) = P$, alors on peut regrouper les points par classe :

$$\{R, \phi(R), \phi^2(R), \ldots, \phi^{\ell-1}(R)\},$$

où ℓ est un diviseur de s. Si la plupart des classes sont de taille s, alors l'exécution de l'algorithme *rho* s'effectue sur q/s classes au lieu de q points et cela réduit la complexité d'un facteur $\sqrt{s}$.

L'endomorphisme ϕ doit être calculable efficacement pour trouver le représentant d'une classe. Cela est le cas lorsque son degré est petit. Celui-ci est le degré maximal entre le numérateur et le dénominateur de la fraction rationnelle définissant l'abscisse de $\phi(x,y)$. Par exemple, le degré de $[-1]$ donné plus haut est 1. Un autre exemple est celui de la courbe `secp256k1` qui possède un endomorphisme de degré 3 :

$$\begin{array}{rccc} \phi: & \texttt{secp256k1} & \longrightarrow & \texttt{secp256k1} \\ & (x,y) & \longmapsto & (\beta x, y), \end{array}$$

où β est un élément d'ordre 3 dans le corps $\mathbf{F}_p$. Il est facile à calculer car ne nécessite qu'une multiplication dans le corps de base, et permet de réduire d'un facteur $\sqrt{3}$ l'algorithme *rho* (soit une réduction d'un facteur $\sqrt{6}$ en combinant avec l'endomorphisme $[-1]$).

1.2.4 Choix des paramètres pour la cryptographie

Les algorithmes précédents donnent certains critères sur le choix des paramètres à respecter afin que le logarithme discret soit suffisamment difficile. Il existe cependant des classes de courbes pour lesquelles le problème devient plus facile quand bien même le groupe généré est d'un grand ordre premier.

Les recommandations actuelles de l'ANSSI (Agence nationale de la sécurité des systèmes d'information) [ANS20] sont des courbes elliptiques définies sur un corps premier, ou sur un corps binaire $\mathbf{F}_{2^n}$ avec n premier, dont le cardinal possède un facteur premier q d'au moins 250 bits.

Le critère n premier sur $\mathbf{F}_{2^n}$ provient de l'existence de sous-corps non triviaux qui peuvent être exploités afin de transférer le problème du logarithme discret vers un autre groupe disposant d'algorithmes plus efficaces [GHS02], ou par des méthodes algébriques [Gau09].

Il existe d'autres cas particuliers comme les courbes anomales qui permet le transfert vers le groupe additif du corps de base [SA98, Sem98, Sma99, ST20, GJ21]. Enfin, il y a également un transfert vers le groupe multiplicatif du corps [MOV93].

1.3 Primitives cryptographiques

Dans cette section on présente les principales primitives cryptographiques couramment employées. Il y a le partage de clé Diffie-Hellman qu'on retrouve notamment dans TLS, mais dont le principe est également à la base du chiffrement *Integrated Encryption Scheme* dont le principe est basé sur le partage de clé. On présente aussi le schéma de signature *Elliptic Curve Digital Signature Agreement* (ECDSA).

1.3.1 Partage de clé Diffie-Hellman

Le principe a été donné dans l'introduction et on le reproduit ici avec la notation des courbes elliptiques. Les étapes sont les suivantes :

1. Alice et Bob se mettent d'accord sur les paramètres d'une courbe elliptique E sur un corps fini $\mathbf{F}_q$, ainsi que d'un point de base P qui génère un groupe d'ordre premier suffisamment grand.
2. Alice choisit un entier secret a et calcule $A = [a]P$;
3. Bob fait de même avec un entier secret b et calcule $B = [b]P$;
4. Alice et Bob s'échangent les points A et B sur un canal public ;
5. Alice calcule $S = [a]B = [ab]P$;
6. Bob calcule $S = [b]A = [ba]P$;
7. Alice et Bob se mettent d'accord sur la façon d'extraire une clé à partir du secret partagé S pour le chiffrement des communications (généralement l'empreinte de la coordonnée affine x par une fonction de hachage).

Les clés publiques d'Alice et Bob peuvent être soit statiques, soit éphémères. Par exemple, TLS 1.3 a supprimé les versions statiques par rapport à TLS 1.2, ce qui permet d'obtenir une *confidentialité persistante* (la divulgation d'une clé privée ne compromet pas les communications passées).

Au partage de clé Diffie-Hellman sont associés deux problèmes connexes à celui du logarithme discret. Tout d'abord il y a le *problème Diffie-Hellman calculatoire* (CDH) qui consiste à calculer le secret partagé $[ab]P$ à partir des valeurs P, $[a]P$ et $[b]P$. Le second est le *problème Diffie-Hellman décisionnel* (DDH) qui consiste à déterminer $Q = [ab]P$ à partir des valeurs P, Q, $[a]P$ et $[b]P$.

Il est évident que si le problème du logarithme discret peut être résolu, alors les problèmes CDH et DDH peuvent l'être aussi. L'inverse n'est pas nécessairement vrai.

1.3.2 Elliptic Curve Integrated Encryption Scheme

Cette primitive permet à Alice de chiffrer un message à partir de la clé publique de Bob. Il est nécessaire que soit associé un algorithme de chiffrement symétrique authentifié. Une version de cette primitive est décrite dans le standard [Res09].

Le fonctionnement utilise un échange Diffie-Hellman à la fois statique et éphémère. Bob possède une paire $(b, B = [b]P)$ de clé privée et publique à long terme, tandis qu'Alice utilise une paire éphémère pour chaque chiffrement. Les étapes générales sont les suivantes :

1. Alice génère un entier éphémère r et calcule $R = [r]P$ et $S = [r]B = [rb]P$;
2. Une clé de chiffrement symétrique $\mathcal{K}$ est dérivée de S ;
3. Alice chiffre un message $M : C = \mathrm{Enc}_{\mathcal{K}}(M)$, avec un tag d'authentification D ;
4. Alice envoie à Bob (C, R, D).
5. Bob reçoit (C, R, D) ;
6. Bob calcule $S = [b]R = [br]P$;
7. La clé de chiffrement symétrique $\mathcal{K}$ est dérivée de S ;
8. Bob déchiffre le message $M = \mathrm{Dec}_{\mathcal{K}}(C)$ et vérifie le tag d'authentification.

1.3.3 Signature ECDSA

ECDSA est un schéma de signature basé sur des courbes elliptiques [Nat13]. Ses paramètres sont une courbe elliptique E et un point de base P d'ordre premier N appartenant à la courbe.

Algorithme 1 : Génération de signature ECDSA

Entrée : message M, clé privée α, point P d'ordre N
Sortie : signature (r, s) du message M signé avec la clé privée α

1: **répéter**
2: $k \leftarrow$ entier aléatoire dans $[1, N-1]$
3: $Q \leftarrow [k]P$
4: $r \leftarrow x_Q \bmod N$
5: $s \leftarrow k^{-1}(\mathrm{H}(M) + \alpha r) \bmod N$
6: **tant que** $r \neq 0$ et $s \neq 0$
7: **retourner** (r, s)

Étant données une clé privée α dans $[1, N-1]$ et une fonction de hachage H, signer un message M est effectué selon l'algorithme 1, et la signature est formée par le couple (r, s). Le processus de vérification consiste en le calcul du point

$$\widetilde{Q} = [\mathrm{H}(M)s^{-1}]P + [rs^{-1}]P_{\mathrm{pub}} \tag{1.3}$$

où $P_{\mathrm{pub}} = [\alpha]P$ est la clé publique du signataire. La signature est valide si r est égal à la coordonnée affine x du point $\widetilde{Q}$ (relevée en tant qu'entier, puis réduite modulo N).

Vulnérabilités. D'une signature (r, s) correspond une équation linéaire dont les deux inconnues sont le nonce et la clé privée. Chaque nouvelle signature générée par une même clé privée ajoute une équation et une inconnue supplémentaire, il y a donc toujours une inconnue de plus qu'il n'y a d'équations.

Dans le cas d'un nonce k réutilisé, on obtiendrait deux signatures (r, s_1) et (r, s_2), et donc un système de deux équations à deux inconnues :

$$\begin{cases} ks_1 \equiv \alpha r + \mathrm{H}(M_1) \pmod{N} \\ ks_2 \equiv \alpha r + \mathrm{H}(M_2) \pmod{N}, \end{cases}$$

La clé privée peut alors s'exprimer directement à partir des données publiques :

$$\alpha = (s_1 \mathrm{H}(M_2) - s_2 \mathrm{H}(M_1))(r(s_2 - s_1))^{-1} \bmod N.$$

Plus généralement, un biais sur le nonce peut être exploité pour reconstituer la clé privée et cela fait l'objet du chapitre 2.

2
Problème du nombre caché

Ce chapitre présente le *problème du nombre caché*, ou *Hidden Number Problem* (HNP). Il est introduit dans [BV96] pour montrer qu'obtenir les bits de poids fort d'un secret partagé dans un échange Diffie-Hellman classique à partir des clés publiques est aussi difficile que de calculer le secret entier.

Ce problème a ensuite été adapté aux schémas de signatures DSA et ECDSA pour montrer qu'une clé privée peut être reconstituée en temps polynomial lorsque le nonce est révélé partiellement pour un nombre suffisant de signatures [NS02, NS03]. Cela se montre efficace pour exploiter les fuites obtenues par canaux auxiliaires ou par injection de fautes, ce qui sera abordé dans les chapitres **4** et **6**.

Le problème en question et son application sur ECDSA sont présentés dans un premier temps, suivi de sa résolution à base de réseaux Euclidiens (la méthode par analyse de Fourier est aussi abordée).

Sommaire

2.1 Description du problème

Le problème tel qu'introduit à l'origine par Boneh et Venkatesan [BV96] est défini, puis est ensuite reformulé pour une meilleure adaptation pour son application à ECDSA et autres cas rencontrés dans les chapitres suivants.

Définition 3 (HNP). Soit q un nombre premier, k un entier. Soit un oracle $\mathcal{O}_\alpha(t)$ qui, à partir d'une entrée t, calcule les k bits de poids fort de $\alpha \cdot t \bmod q$, où α est inconnu.

Le problème HNP consiste à calculer le nombre caché α en temps polynomial (en $\log q$), étant donné un accès à l'oracle $\mathcal{O}_\alpha(t)$.

Dans un échange Diffie-Hellman, le nombre caché α est le secret partagé g^{ab}, où g est un générateur du groupe multiplicatif du corps $\mathbf{F}_q$, et a et b sont les secrets d'Alice et Bob respectivement. Un tiers qui récupère les clés publiques $A = g^a$ et $B = g^b$ peut engager un échange de clés avec Alice en envoyant comme clé publique $C' = BC$ (où $C = g^c$ et c est sa clé privée). De son côté, Alice calcule

$$\begin{aligned} C'^a &\equiv B^a \cdot C^a && \pmod q \\ &\equiv \alpha \cdot A^c && \pmod q. \end{aligned}$$

La personne tierce est en mesure de calculer $t = A^c$ et on retrouve la description du problème. L'accès à l'oracle est donc une fuite lors du calcul $C'^a \bmod q$ du côté d'Alice. Un exemple concret sur le protocole TLS est présentée dans l'attaque Raccoon [Mer+21].

Le problème peut être reformulé de la façon suivante :

Définition 4 (HNP, alternative). Soit n équations linéaires

$$u_i X + v_i \equiv Y_i \pmod q, \quad 1 \leq i \leq n, \tag{2.1}$$

à $(n+1)$ variables $X, Y_1, \ldots, Y_n$. Le problème du nombre caché consiste à trouver X lorsque les variables Y_i vues en tant qu'entiers sont bornées dans des intervalles de largeurs inférieures à q/L_i pour des entiers L_i positifs.

Il est clair qu'un système linéaire ayant plus de variables que d'équations ne peut aboutir à une solution unique. La connaissance d'information supplémentaire sur n de ces variables réduit le nombre de solutions possibles et c'est l'objectif des méthodes par réseaux Euclidiens ou analyse de Fourier d'exploiter ces biais pour résoudre ce système.

Le cas le plus simple est lorsque les bits de poids fort des variables Y_i sont connues, ce qui correspond à la formulation initiale, mais cela s'adapte également lorsque ce sont les bits de poids faible qui sont connus en ajustant l'équation : l'important est que la partie inconnue des variables Y_i puisse être encadrée dans un intervalle relativement petit. C'est le sujet de la partie ci-dessous dans le cadre de ECDSA.

2.1.1 Application à ECDSA

Rappelons qu'une signature ECDSA est le couple (r, s) qui satisfait la relation

$$k \equiv \alpha r/s + \mathrm{H}(M)/s \pmod q,$$

où k un est nonce généré exclusivement pour cette signature, et α est la clé privée du signataire. Cette dernière est le nombre caché car fixe pour plusieurs signatures réalisées avec la même clé. Tandis que la valeur éphémère k joue le rôle de la variable Y dont une partie seulement est inconnue.

On considère plusieurs cas selon la partie connue de k.

Cas 1 (bits de poids faible). Soit $k = am + b$ la division Euclidienne de k par un entier m. On suppose que m et b sont connus. Cela correspond notamment au cas des ℓ bits de poids faible connus lorsque $m = 2^\ell$, mais plus généralement pour des valeurs m quelconques (voir la section 6.4.2 dans le chapitre sur l'attaque en faute différentielle pour un exemple). Alors on a

$$\begin{cases} Y = a \\ u = r/(sm) \mod q, \\ v = \mathrm{H}(M)/(sm) - b/m \mod q. \end{cases}$$

Si la valeur k est issue de la méthode de bourrage donnée en section 3.2.2, alors l'encadrement de la partie inconnue est

$$\frac{2^t - b}{m} \leq a < \frac{2^t + q - b}{m},$$

qui est un intervalle de largeur $L = q/m$.

Cette situation inclut également le cas où la méthode de randomisation par l'ordre du groupe est utilisée. Dans ce cas le scalaire k est bien plus large que q. Si la valeur aléatoire utilisée est un entier de λ bits, alors k est dans l'intervalle $[2^{\lambda-1}q,\ (2^\lambda + 1)q]$ donc la partie inconnue de k est encadrée par

$$\frac{q - b}{m} \leq a < \frac{q + 2^\lambda q - b}{m},$$

qui est un intervalle de largeur $L = 2^\lambda q/m$. Il est donc nécessaire que m soit plus grand que 2^λ pour être exploité dans l'attaque par réseau Euclidien. En particulier si $m = 2^\ell$, cela signifie que le nombre ℓ de bits de poids faible révélés doit être supérieur à λ.

Cas 2 (bits de poids fort). Dans le cas où il s'agit de a et m qui sont connus dans la division Euclidienne de k par m, alors on a

$$\begin{cases} Y = b, \\ u = r/s \mod q, \\ v = \mathrm{H}(M)/s - am \mod q. \end{cases}$$

La borne sur l'inconnue est simplement $0 \leq b < m$. Supposons que la méthode de bourrage est appliquée et que les $(\ell+1)$ bits de poids fort sont connus (bit lié au bourrage inclus), alors $m = 2^{t-\ell}$ (pour certaines courbes, cela peut être approximée par $q/2^\ell$ lorsque q est très proche de 2^t).

Il y a un cas particulier abordé dans la section 6.3.5 (chapitre sur l'injection de faute différentielle) où les $(\ell+1)$ bits de poids fort a sont *presque* connus : une valeur μ est obtenue qui peut valoir soit a soit $a+1$. Dans ce cas on a

$$\begin{cases} Y = k - \mu 2^{t-\ell}, \\ u = r/s \mod q, \\ v = \mathrm{H}(M)/s - \mu 2^{t-\ell} \mod q. \end{cases}$$

La borne sur l'inconnue est $-2^{t-\ell} \leq Y < 2^{t-\ell}$ de largeur $2^{t-\ell+1}$ (qui peut être approximé par $q/2^{\ell-1}$).

2.2 Résolution par réseaux Euclidiens

La connaissance partielle d'inconnues dans les systèmes d'équations linéaires peut être traduit sous la forme de recherche d'un vecteur court (par rapport à une norme) dans un réseau Euclidien. L'objet de cette section est d'introduire les notions afin de comprendre son utilisation et le nombre de signatures nécessaires en fonction du nombre de bits connus par nonce.

2.2.1 Réseaux Euclidiens

Un réseau Euclidien de dimension n est un sous-groupe additif de $\mathbf{R}^n$. Il peut être représenté par une base $\mathbf{B}(b_1, \ldots, b_d)$, où $b_i \in \mathbf{R}^n$ pour $1 \leq i \leq d$, et le réseau comporte toutes les combinaisons linéaires entières :

$$\mathcal{L}(\mathbf{B}) = \Big\{ \sum_{i=1}^{d} z_i \cdot b_i \mid z_i \in \mathbf{Z} \Big\}.$$

Le rang du réseau est d et généralement on a $d = n$, c'est-à-dire un réseau de rang plein. Par simplicité on ne considère que ce cas dans la suite.

À un réseau Euclidien est associé son déterminant (ou volume) dont le calcul est indépendant de la base :

$$\det(\mathcal{L}(\mathbf{B})) = \det(B \cdot B^t)^{1/2} = |\det \mathbf{B}|,$$

où la base $\mathbf{B}$ est vue comme une matrice dont chaque ligne est un vecteur de la base.

On note $\lambda(\mathcal{L})$ la longueur du plus court vecteur non nul d'un réseau Euclidien $\mathcal{L}$ et elle satisfait l'inégalité $\lambda(\mathcal{L}) \leq \sqrt{n}(\det \mathcal{L})^{1/n}$. À cette valeur est associée le problème du vecteur le plus court, *Shortest Vector Problem* (SVP) qui consiste à trouver un vecteur v de norme exactement égale à $\lambda(\mathcal{L})$.

L'algorithme de Gauss permet de résoudre SVP efficacement en dimension 2. Pour les dimensions supérieures, l'algorithme LLL [LLL82] permet de construire en temps polynomial une base dont le plus petit vecteur a une norme majorée par $2^{(n-1)/4} \det(\mathcal{L})^{1/n}$. En pratique le plus petit vecteur obtenu a une norme bien inférieure à cette borne, mais il n'est pas pour autant assuré que ce soit le plus petit vecteur du réseau Euclidien. Un autre algorithme plus performant pour obtenir une base réduite, mais plus coûteux en exécution est BKZ [SE94].

2.2.2 Construction pour résolution de HNP

Tout d'abord on introduit la notation $|\cdot|_q$ qui est définie par

$$|z|_q = \min_{y \in \mathbf{Z}} |z - yq|,$$

pour tout nombre réel z. Il s'agit d'une réduction modulo q dans l'intervalle $[-q/2, q/2]$ suivi d'une valeur absolue.

Soit $uX + v \equiv Y \bmod q$ une équation linéaire en les variables X and Y, où une approximation de Y vue en tant qu'entier est connue :

$$B_1 \leq Y < B_2,$$

avec B_1 et B_2 deux entiers qui satisfont l'inégalité $(B_2 - B_1) < q/L$ pour un entier positif L. La largeur de l'intervalle peut être réduite davantage en le centrant sur 0 : posons $C = (B_1 + B_2)/2$ le centre de l'intervalle, et on obtient l'inégalité

$$|Y - C| < q/(2L).$$

Ainsi en posant $v' = C - v$, on a

$$|uX - v'|_q = |Y - C|_q = |Y - C|,$$

car on sait que $(Y - C)$ est situé dans l'intervalle $[-q/2, q/2]$. En utilisant les bornes on arrive à l'inégalité

$$|uX - v'|_q < q/(2L), \tag{2.2}$$

qui signifie que lorsqu'elle est vue modulo q, la valeur v' est proche d'un multiple de X (le nombre caché).

Étant données n inégalités de la forme

$$|u_i X - v'_i|_q < q/(2L_i),$$

où X est une variable commune, on peut construire le réseau Euclidien généré par cette matrice entière :

$$\begin{bmatrix} 2L_1 q & & & & & \\ & 2L_2 q & & & & \\ & & \ddots & & & \\ & & & 2L_n q & & \\ 2L_1 u_1 & 2L_2 u_2 & \cdots & 2L_n u_n & 1 & 0 \\ 2L_1 v'_1 & 2L_2 v'_2 & \cdots & 2L_n v'_n & 0 & q \end{bmatrix}.$$

En notant $\mathbf{m}_i$ la ligne en position i de la matrice et

$$\begin{cases} \mathbf{u} = (2L_1 u_1, \ldots, 2L_n u_n, 1, 0) \\ \mathbf{v} = (2L_1 v'_1, \ldots, 2L_n v'_n, 0, q), \end{cases}$$

les vecteurs qui forment les deux dernières lignes, il existe des entiers λ_i tels que le vecteur $X\mathbf{u} - \mathbf{v} + \sum_{i=1}^{n} \lambda_i \mathbf{m}_i$ est un vecteur court du réseau Euclidien. En effet, chacune de ses coordonnées sont bornées par q d'après les inégalités.

Après l'application d'un algorithme de réduction de base comme LLL ou BKZ, il est possible de retrouver parmi les vecteurs de la base réduite celui que l'on recherche. Par construction, ce vecteur court contient le nombre caché X dans sa pénultième coordonnée.

2.2.3 Nombre de signatures

Dans cette partie on montre l'efficacité de la méthode en fonction du nombre de signatures pour plusieurs tailles de courbes elliptiques.

L'algorithme BKZ avec blocs de taille 30 a été utilisé pour la réduction de base avec FPLLL au moyen de son interface Python [FPLLL21], excepté pour la courbe `secp521r1` où il a été remplacé par LLL. Cela a été répété 100 fois et le nombre minimal, médian et maximal de signatures en fonction du nombre de bits ℓ de poids faible connus du nonce sont indiqués dans le tableau **2.1**.

Le ratio t/ℓ (où t est la taille en bits du cardinal de la courbe) donne une approximation du nombre de signatures nécessaires pour résoudre le système, d'où on peut dire que ℓ bits d'un nonce apporte environ ℓ bits d'information sur la clé privée. Cependant cela est moins vrai à mesure que le nombre de bits connus par nonce est faible. En effet, le vecteur cherché n'est plus significativement plus court que les autres dans le réseau Euclidien. Des propositions d'améliorations dans ces circonstances sont données dans [AH21, Sun+21].

Table 2.1 : Nombre de signatures en fonction du nombre de bits ℓ de poids faible connus du nonce dans ECDSA pour reconstruire la clé privée.

ℓ	`secp224r1`			`secp256r1`			`secp384r1`			`secp521r1`		
	min	med	max	min	med	max	min	med	max	min	med	max
4	57	60	65	64	70	77						
5	45	47	50	51	54	57	80	85	90			
6	37	39	41	42	45	47	64	68	71			
7	31	33	35	36	38	40	55	57	60			
8	28	29	30	31	33	34	47	50	52	75	85	94
9	25	26	27	28	29	30	42	44	45	65	69	75
10	22	23	24	25	26	27	38	40	41	56	59	62
11	20	21	22	23	24	25	35	36	37	49	53	55
12	19	19	20	21	22	23	31	33	34	44	47	49
13	17	18	18	20	20	21	29	30	31	41	43	45
14	16	17	17	18	19	19	27	28	29	38	39	41
15	15	15	16	17	18	18	26	26	27	36	37	38
16	14	14	15	16	16	17	24	25	25	33	34	36
17	13	14	14	15	16	16	23	23	24	31	32	33
18	13	13	13	14	15	15	21	22	22	29	30	31
19	12	12	13	14	14	14	20	21	21	28	28	29
20	11	12	12	13	13	14	19	20	20	26	27	27
21	11	11	11	12	13	13	18	19	19	25	26	26
22	10	11	11	12	12	12	18	18	18	24	24	25
23	10	10	10	11	12	12	17	17	18	23	23	24
24	10	10	10	11	11	11	16	16	17	22	22	23
25	9	9	10	10	11	11	16	16	16	21	21	22
26	9	9	9	10	10	11	15	15	16	20	21	21
27	9	9	9	10	10	10	14	15	15	20	20	20
28	8	8	9	9	10	10	14	14	14	19	19	19
29	8	8	8	9	9	9	13	14	14	18	18	19
30	8	8	8	9	9	9	13	13	13	18	18	18
31	8	8	8	9	9	9	13	13	13	17	17	18
32	7	7	8	8	8	9	12	12	13	17	17	17

2.3 Résolution par analyse de Fourier

Une alternative aux réseaux Euclidiens pour résoudre le problème HNP est la méthode de Bleichenbacher basée sur l'analyse de Fourier [Ble00]. Plusieurs travaux ont montré son application lorsqu'un bit par nonce est connue [Ara+14, Mul+14], voire lorsqu'un bit est connu avec une probabilité légèrement inférieure à 1 [Ara+20]. Bien que cette méthode n'est pas utilisée dans cette thèse, on donne ci-dessous les idées principales.

Dans une instance du problème HNP, on a un ensemble de couples (u_i, v_i) pour lesquels on recherche une valeur commune X telle que l'expression $u_i X + v_i$ réduite modulo q dans l'intervalle $[-q/2, q/2]$ soit proche de 0 (un biais d'un bit signifie que les valeurs sont toutes dans la moitié $[-q/4, q/4]$ de l'intervalle). Tandis que pour une valeur quelconque w, on s'attend à ce que les expressions $u_i w + v_i$ soient davantage réparties dans la totalité de l'intervalle, montrant un biais moins présent.

C'est ce principe qui est derrière cette méthode, en utilisant la fonction de biais ci-dessous :

$$B_q(w) = \frac{1}{n} \sum_{j=1}^{n} e^{2\pi i (u_i w + v_i)/q}.$$

Lorsque $w = X$, chaque terme de la somme $B_q(w)$ sont des nombres complexes proches de 1. Tandis que pour une valeur quelconque les termes correspondent à des points répartis dans le cercle unité dont la moyenne converge vers 0 pour un large échantillon.

Que ce biais soit observable pour la bonne valeur w n'est bien sûr pas suffisant, car on ne serait pas plus avancé qu'une recherche exhaustive. Une étape intermédiaire est nécessaire pour élargir l'apparition du biais pour des valeurs proches de celle recherchée : ainsi en parcourant un nombre restreint de valeurs w_i dans $[0, q-1]$ équitablement réparties, la valeur $B_q(w_i)$ maximale révèle une approximation de la valeur secrète X. La transformée de Fourier inverse est utilisée pour effectuer les calculs efficacement.

L'approximation obtenue peut être réinjectée dans les équations afin de réitérer le processus pour améliorer l'approximation jusqu'à obtenir la valeur entière ou en terminant éventuellement par une autre méthode.

Pour de plus amples détails, on peut se référer aux articles référencés plus haut.

3

Multiplication scalaire et exceptions

Dans ce chapitre on aborde en détails les formules d'addition de points selon plusieurs systèmes de coordonnées et les cas exceptionnels à gérer. Ensuite on présente plusieurs algorithmes de multiplication scalaire ainsi que l'analyse des exceptions des formules lors de leur usage dans ces contextes. Quelques exemples issus de bibliothèques cryptographiques sont donnés en fin de chapitre.

Sommaire

3.1 Exceptions dans les formules d'addition de points

Il existe de nombreuses formules pour les courbes elliptiques de chaque forme et selon plusieurs systèmes de coordonnées. La plupart sont disponible dans la base de données *Explicit-Formulas Database* (EFD).[1] On y trouve les décompositions en opérations élémentaires (addition, soustraction, multiplication), accompagnés des préconditions et les références.

Dans cette section on analyse plusieurs formules souvent utilisées en pratique. Celles-ci sont les formules d'addition ou de doublement de points, mais aussi celles qui reconstruisent une coordonnée non utilisée lors des calculs comme les formules `XYcoZ` ou `XZ`. Les situations où elles ne peuvent s'appliquer sont mises en évidence et une vue d'ensemble est donnée dans le tableau **3.1**.

Table 3.1 : Cas exceptionnels dans les formules pour les courbes elliptiques en forme de Weierstraß courte.

Formules		Exceptions
Affines	Add	$P_1 = \pm P_2$, $P_i = \mathcal{O}$
	Dbl	$P_1 = \mathcal{O}$, $y(P_1) = 0$
Jacobiennes `XYZ`	Add	$P_1 = P_2$, $P_i = \mathcal{O}$
	Dbl	Aucune
`XYcoZ`	`XYcoZ-ADDC`, `XYcoZ-ADD`	$P_1 = \pm P_2$, $P_i = \mathcal{O}$
	`XYcoZ-DBL`	$y(P_1) = 0$
	Récupération de Z (†)	$x(P) = 0$, $y(P) = 0$
`XZ`	Add diff.	$P_1 = P_2$
	Dbl	Aucune
	Récupération de y (†)	$P_i = \mathcal{O}$, $y(P_2 - P_1) = 0$

(†) Utilisées avec l'algorithme Montgomery ladder.

3.1.1 Formules pour représentation affine

La représentation affine de points sur une courbe elliptique sous forme de Weierstraß exclue le point à l'infini de toute opération dans les formules, que ce soit en entrée ou en sortie. De plus, l'addition ne peut être effectuée de la même façon lorsque les deux points sont égaux et nécessite un traitement à part.

Ainsi, les exceptions sont :

- Addition de points : P_1 ou P_2 correspond au point à l'infini ; ou alors ils sont égaux ou opposés ;
- Doublement de point : P_1 est le point à l'infini, ou est un point d'ordre 2 (si un tel point existe sur la courbe).

1. http://hyperelliptic.org/EFD/.

3.1.2 Formules pour coordonnées Jacobiennes XYZ

Avec le système de coordonnées Jacobiennes, un point affine (x, y) est représenté par $(X : Y : Z)$ où $x = X/Z^2$ et $y = Y/Z^3$ pour une valeur non nulle Z. Quant au point à l'infini, il est représenté par $(1 : 1 : 0)$, ou en général par $(\lambda^2 : \lambda^3 : 0)$ pour toute valeur λ non nulle.

Par conséquent, le point à l'infini possède une représentation, et ses coordonnées peuvent être soumises dans les formules. Cela n'assure pas pour autant que le résultat soit correct.

Un exemple de formules pour ce système de coordonnées dans la base EFD est donné par `add-2007-bl` pour l'addition et par `dbl-2007-dbl` pour le doublement.

Addition de points. L'addition de points entre $P_1 = (X_1 : Y_1 : Z_1)$ et $P_2 = (X_2 : Y_2 : Z_2)$ est donnée par les formules suivantes :

$$\begin{cases} X_3 = (Y_2Z_1^3 - Y_1Z_2^3)^2 - (X_2Z_1^2 - X_1Z_2^2)^2(X_2Z_1^2 + X_1Z_2^2) \\ Y_3 = (Y_2Z_1^3 - Y_1Z_2^3)(X_1Z_2^2(X_2Z_1^2 - X_1Z_2^2)^2 - X_3) - Y_1Z_2^3(X_2Z_1^2 - X_1Z_2^2)^3 \\ Z_3 = (X_2Z_1^2 - X_1Z_2^2)Z_1Z_2. \end{cases}$$

Lorsque les deux points ont une représentation affine ($Z \neq 0$), on retrouve les formules classiques du cas affine et par conséquent le cas de l'égalité est une exception. Il convient alors de regarder les situations où le point à l'infini apparaît en entrée ou en sortie. Le seul cas compatible est lorsque $P_1 = -P_2$ (et $P_1 \neq P_2$). Dans ce cas on obtient

$$(X_3 : Y_3 : 0) = (\mu^2 : \mu^3 : 0), \quad \mu = 2Y_1Z_1^3$$

où X_3 et Y_3 sont non nuls. Une représentation valide du point à l'infini est obtenue et cela est bien le résultat attendu de l'addition.

En résumé, les exceptions sont : P_1 ou P_2 est le point à l'infini ; P_1 et P_2 sont égaux.

Doublement de point. Le doublement du point $P_1 = (X_1 : Y_1 : Z_1)$ est donné par les formules suivantes :

$$\begin{cases} X_3 = (3X_1^2 + AZ_1^4)^2 - 8X_1Y_1^2 \\ Y_3 = (3X_1^2 + AZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ Z_3 = 2Y_1Z_1. \end{cases}$$

À nouveau, il faut regarder lorsque le point à l'infini apparaît en entrée ou en sortie. Si $P_1 = (\lambda^2 : \lambda^3 : 0)$, alors

$$(X_3 : Y_3 : Z_3) = (\lambda^8 : \lambda^{12} : 0),$$

ce qui est une représentation valide du point à l'infini. Prenons le cas d'un point d'ordre 2, alors $Y_1 = 0$ et on obtient

$$(X_3 : Y_3 : Z_3) = (\mu^2 : \mu^3 : 0), \quad \mu = -(3X_1^2 + AZ_1^4) \neq 0,$$

qui est lui aussi une représentation valide. En effet, l'expression $3X_1^2 + AZ_1^4$ est non nulle, sinon l'existence d'un tel point impliquerait que le discriminant de la courbe $4A^3 + 27B^2$ vaut zéro.

Il n'y a donc pas d'exception.

3.1.3 Formules XYcoZ pour coordonnées Jacobiennes

Ces formules sont basées sur la même représentation que précédemment, mais avec la particularité que les entrées (et les sorties) se partagent la même troisième coordonnée projective Z.

La variante de ces formules considérée pour l'analyse est celle qui n'utilise pas la coordonnée Z [Gou+11]. En conséquence il y a nécessité d'une formule de reconstruction de cette coordonnée afin de remettre sous forme affine. La correspondance avec la base EFD pour la formule d'addition est `zadd-2007-m`.

Addition de points avec mise à jour. Ces formules effectuent l'addition entre $P_1 = (X_1 : Y_1 : Z_1)$ et $P_2 = (X_2 : Y_2 : Z_2)$ avec $Z_1 = Z_2$ et met également à jour les coordonnées projectives de P_1 vers $(X_1' : Y_1' : Z_1')$ de telle sorte que la coordonnée Z corresponde à celle du résultat de l'addition. Les formules, notées `XYcoZ-ADD`, sont données ci-dessous :

$$\begin{cases} X_3 = (Y_2 - Y_1)^2 - (X_2 - X_1)^2(X_1 + X_2) \\ Y_3 = (Y_2 - Y_1)(X_1(X_2 - X_1)^2 - X_3) - Y_1(X_2 - X_1)^3 \\ X_1' = X_1(X_2 - X_1)^2 \\ Y_1' = Y_1(X_2 - X_1)^3 \\ Z_3 = Z_1' = (X_2 - X_1)Z_1. \end{cases}$$

Il y a davantage de contraintes par rapport aux formules classiques avec les coordonnées Jacobiennes à cause de la valeur commune pour la coordonnée Z. En effet, le cas $P_1 = -P_2$ implique que la mise à jour de P_1 sera faite par des coordonnées nulles.

Les exceptions sont donc les mêmes que pour le cas des formules affines : P_1 ou P_2 est le point à l'infini, P_1 et P_2 sont égaux ou opposés.

Addition conjuguée de points. Ces formules pour l'addition calculent en même temps la différence, notamment car une partie des opérations intermédiaires sont communes. Les sorties $P_1 + P_2 = (X_3 : Y_3 : Z_3)$ et $P_1 - P_2 = (X_4 : Y_4 : Z_4)$ sont données par les formules `XYcoZ-ADDC` ci-dessous :

$$\begin{cases} X_3 = (Y_2 - Y_1)^2 - (X_2 - X_1)^2(X_1 + X_2) \\ Y_3 = (Y_2 - Y_1)(X_1(X_2 - X_1)^2 - X_3) - Y_1(X_2 - X_1)^3 \\ X_4 = (Y_2 + Y_1)^2 - (X_2 - X_1)^2(X_1 + X_2) \\ Y_4 = -(Y_2 + Y_1)(X_1(X_2 - X_1)^2 - X_4) - Y_1(X_2 - X_1)^3 \\ Z_3 = Z_4 = (X_2 - X_1)Z_1. \end{cases}$$

Comme avec l'addition de points avec mise à jour, ces formules ont les mêmes exceptions que le cas affine.

Doublement de point et mise à jour. Il s'agit des mêmes formules qu'en cas classique des coordonnées Jacobiennes, mais avec la mise à jour de l'entrée pour partager la même

valeur pour Z que le résultat du doublement. Les formules, notées `XYcoZ-DBL`, sont données ci-dessous :

$$\begin{cases} X_3 = (3X_1^2 + AZ_1^4)^2 - 8X_1Y_1^2 \\ Y_3 = (3X_1^2 + AZ_1^4)(4X_1Y_1^2 - X_3) - 8Y_1^4 \\ X_1' = 4X_1Y_1^2 \\ Y_1' = 8Y_1^4 \\ Z_3 = Z_1' = 2Y_1Z_1. \end{cases}$$

Le doublement du point à l'infini donne un résultat correct pour $P_1 + P_2$ comme pour le cas des coordonnées complètes, mais aussi pour la mise à jour de P_1. En effet, si $P_1 = (\lambda^2 : \lambda^3 : 0)$, alors ses coordonnées mises à jour sont $X_1' = (2\lambda^4)^2$, $Y_1' = (2\lambda^4)^3$ et $Z_1' = 0$, ce qui est une représentation valide du point à l'infini. Cependant, ce n'est pas le cas pour un point d'ordre 2, où la mise à jour entraînerait $X_1' = Y_1' = Z_1' = 0$.

L'unique exception est : P_1 est un point d'ordre 2.

Remarque 1. Notons tout de même que le cas du point à l'infini en entrée donne des résultats incompatibles avec les autres formules et est un cas un exceptionnel à l'usage dans un algorithme de multiplication scalaire.

Récupération de la coordonnée Z. Finalement, il y a la nécessité de récupérer la coordonnée Z en vue d'obtenir une représentation affine. Étant donné un point $P = (X : Y : Z)$ avec la coordonnée Z inconnue, et (x, y) la représentation affine connue de ce même point, alors comme $x = X/Z^2$ et $y = Y/Z^3$, on a la relation suivante :

$$Z = \frac{x \cdot Y}{y \cdot X}. \tag{3.1}$$

Si x ou y est nul, alors la coordonnée Z ne peut être reconstruite, ce qui entraîne des cas exceptionnels pour les courbes elliptiques ayant de tels points.

3.1.4 Formules XZ pour coordonnées projectives homogènes

Ces formules sont basées sur les coordonnées projectives homogènes, avec la particularité que la coordonnée y n'est pas utilisée lors des calculs. Un point (x, y) est représenté par $[X : Z]$ avec $x = X/Z$ et le point à l'infini par $[1 : 0]$ (ou plus généralement $[\lambda : 0]$ pour tout λ non nul). Un point quelconque n'est donc pas différencié de son opposé et l'utilisation des crochets est pour distinguer et éviter des ambiguïtés avec les coordonnées projectives complètes.

Une conséquence de ne pas utiliser la seconde coordonnée est qu'il est nécessaire d'avoir une donnée supplémentaire pour réaliser l'addition. Il existe aussi la possibilité de reconstruire la coordonnée manquante sous certaines conditions. Les références dans la base EFD est `mdadd-2002-it-4` pour l'addition différentielle et `dbl-2002-bj` pour le doublement de point, ainsi que les articles [BJ02, IT02].

Addition différentielle de points. Les points n'étant pas différenciés de leurs opposés, l'addition ne peut être calculée sans une donnée auxiliaire. En effet, l'abscisse du résultat de l'addition peut être soit $x(P_1 + P_2)$ ou soit $x(P_1 - P_2)$. Ainsi si l'une de ces deux valeurs est connue à l'avance il ne reste plus qu'une possibilité.

Étant donnés P_1 et P_2 par leur représentation $[X_1 : Z_1]$ et $[X_2 : Z_2]$, et la coordonnée affine x de $P_1 - P_2$, alors l'addition de $P_1 + P_2$ est donnée par les formules suivantes :

$$\begin{cases} X_3 = 2(X_1Z_2 + X_2Z_1)(X_1X_2 + AZ_1Z_2) + 4BZ_1^2Z_2^2 - x(X_1Z_2 - X_2Z_1)^2, \\ Z_3 = (X_1Z_2 - X_2Z_1)^2. \end{cases}$$

Ce qui rend cette addition différentielle intéressante est qu'elle donne des résultats corrects avec le point à l'infini. Premièrement il y a le cas où l'un des points est le point à l'infini. Soit $P_1 \neq \mathcal{O}$ et $P_2 = \mathcal{O}$, donc on a $Z_1 \neq 0$, $X_2 \neq 0$ et $Z_2 = 0$. Alors $X_3 = 2X_1Z_1X_2^2 - xX_2^2Z_1^2$, $Z_3 = X_2^2Z_1^2 \neq 0$ et

$$\frac{X_3}{Z_3} = \frac{2X_1Z_1X_2^2}{X_2^2Z_1^2} - x\frac{X_2^2Z_1^2}{X_2^2Z_1^2} = 2x - x = x,$$

donc $[X_3 : Z_3]$ est une représentation valide de P_1 qui est le résultat attendu de $P_1 + P_2$.

Ensuite, il y a le cas où le point à l'infini est le résultat de l'addition, c'est-à-dire que P_1 et P_2 sont opposés. En effet, si $P_1 = -P_2$ (et $P_1 \neq P_2$), alors $Z_3 = 0$ et on a la relation

$$\frac{X_3}{Z_1^2Z_2^2} = 4\left(\left(\frac{X_1}{Z_1}\right)^3 + A\frac{X_1}{Z_1} + B\right) \neq 0,$$

ce qui signifie que le résultat est une représentation valide du point à l'infini.

La seule exception est lorsque les deux points sont égaux, mais ce cas ne peut arriver dans son utilisation avec l'algorithme Montgomery ladder comme il est expliqué dans la section 3.2.3.

Remarque 2. Il y a une autre variante de formules pour l'addition différentielle donnée à l'entrée `mdadd-2002-bj` de la base EFD. Dans ces formules, il y a la coordonnée x en facteur pour le calcul de Z_3. En conséquence le calcul de $P_1 + P_2$ est erroné lorsque x est nul. Un point ayant cette coordonnée nulle utilisé en entrée de l'algorithme de multiplication scalaire Montgomery ladder n'est pas compatible.

Doublement de point. Les formules sont données ci-dessous :

$$\begin{cases} X_3 = (X_1^2 - AZ_1^2)^2 - 8BX_1Z_1^3 \\ Z_3 = 4Z_1(X_1^3 + AX_1Z_1^2 + BZ_1^3). \end{cases}$$

Comme pour les coordonnées Jacobiennes, il n'y a pas de cas exceptionnel pour le doublement de points.

En effet, si $P_1 = [\lambda : 0]$ pour une valeur λ non nulle, alors $X_3 = \lambda^2$ et $Z_3 = 0$, ce qui est une représentation valide du point à l'infini. Dans le cas où P_1 est un point d'ordre 2, alors on a $X_1^3 + AX_1Z_1^2 + BZ_1^3 = 0$. Donc Z_3 est nul, et $X_3 \neq 0$ sinon cela signifierait que le discriminant de la courbe est nulle.

Récupération de la coordonnée y. Ces dernières formules concernent la reconstruction de la coordonnée y. Étant donnés $[X_1 : Z_1]$ et $[X_2 : Z_2]$ les représentations de P_1 et P_2, ainsi que $P = (x, y)$ tel que $P_2 - P_1 = P$, alors les formules ci-dessous donnent une représentation complète du point P_1 :

$$\begin{cases} X_1' = 2yX_1Z_1Z_2 \\ Y_1' = 2BZ_1^2Z_2 + Z_2(AZ_1 + xX_1)(xZ_1 + X_1) - X_2(xZ_1 - X_1)^2 \\ Z_1' = 2yZ_1^2Z_2 \end{cases} \tag{3.2}$$

Cette reconstruction est erronée si P_1 ou P_2 est le point à l'infini ou si y est nul (également dans le cas où P_1 et P_2 sont égaux, mais ce cas ne peut arriver au cours de l'utilisation, voir section 3.2.3).

3.2 Montgomery Ladder

Le premier algorithme analysé dans ce chapitre est la multiplication scalaire nommée Montgomery ladder [Mon87]. Un avantage de cet algorithme pour calculer $Q = [k]P$ est que les mêmes opérations sont exécutées pour chaque bit traité. Ce comportement régulier le rend attrayant pour un usage en cryptographie.

Une présentation de l'algorithme est donnée avec quelques variantes, suivi de plusieurs techniques pour que le nombre d'itérations soit fixe. On termine avec l'analyse des éventuels cas exceptionnels selon les situations.

3.2.1 Description de l'algorithme

L'algorithme utilise deux points R_0 et R_1 qui satisfont toujours l'invariant de boucle $R_1 - R_0 = P$. Soit $k = (k_{n-1}, \ldots, k_0)_2$ la représentation binaire du scalaire k, et supposons que les bits de poids fort $\widehat{k} = (k_{n-1}, \ldots, k_{j+1})_2$ ont déjà été traités : on a $R_0 = [\widehat{k}]P$ et $R_1 = [\widehat{k}+1]P$. Le couple (R_0, R_1) est mis à jour en fonction du bit courant k_j comme suit :

$$(R_0, R_1) \leftarrow \begin{cases} ([2]R_0, R_0 + R_1) & \text{si } k_j = 0, \\ (R_0 + R_1, [2]R_1) & \text{si } k_j = 1. \end{cases}$$

En conséquence, à la fin de cette étape on a $R_0 = [2\widehat{k} + k_j]P$, et la relation $R_1 - R_0 = P$ est encore vraie. Ce procédé continue jusqu'au dernier bit, et l'état final donne $R_0 = [k]P$.

Une description complète est donnée dans l'algorithme 2.

Échange conditionnel. Pour éviter des branchements conditionnels pour que les résultats des opérations soit affectés aux bons points R_0 ou R_1, un échange conditionnel est généralement utilisé (avec des masques binaires par exemple). Une première fois pour échanger les points avant les opérations d'addition et de doublement, puis une seconde fois après pour rétablir l'ordre des points. Cette variante est donnée dans l'algorithme 3.

Une autre variante est souvent utilisée dans la plupart des implémentations de bibliothèques cryptographiques, où le second échange est fusionné avec le premier de l'itération suivante. Cette variante est donnée dans l'algorithme 4. Pour cela une variable pbit est introduite pour garder trace du bit précédent : lorsqu'elle vaut 0, alors l'état

actuel du couple (R_0, R_1) est « normal », tandis que si elle vaut 1, alors R_0 et R_1 sont inversés (notamment, on a la relation $R_1 - R_0 = -P$). Un dernier échange conditionnel est nécessaire après la dernière boucle pour s'assurer que l'état soit redevenu normal avant de fournir la sortie de l'algorithme.

Les échanges des points dans le couple (R_0, R_1) jouent un rôle essentiel dans l'attaque en faute différentielle du chapitre 6 ; notamment car l'invariant de boucle devient

$$R_{1-\text{pbit}} - R_{\text{pbit}} = P$$

dans la variante de l'algorithme 4.

Variante XYcoZ. Les formules `XYcoZ` sont adaptées pour l'algorithme Montgomery ladder, qui doit être légèrement modifié en remplaçant l'addition de points et le doublement par les formules `XYcoZ-ADDC` et `XYcoZ-ADD` :

$$R_{k_i}, R_{1-k_i} \leftarrow \texttt{XYcoZ-ADDC}(R_{k_i}, R_{1-k_i})$$
$$R_{k_i}, R_{1-k_i} \leftarrow \texttt{XYcoZ-ADD}(R_{k_i}, R_{1-k_i})$$

Cela reste équivalent à la variante classique tel qu'on peut le voir dans la figure 3.1 où une étape aboutit au même résultat.

On peut notamment remarquer que l'invariant apparaît entre les deux opérations. C'est à partir de cette observation qu'il est possible d'appliquer la récupération de la coordonnée Z manquante lors du traitement du dernier bit du scalaire en appliquant la formule de l'équation (3.1) entre les opérations `XYcoZ-ADDC` et `XYcoZ-ADD`. Une alternative est proposée en annexe A par l'introduction d'une fonction `XYcoZ-SUB` après la dernière itération.

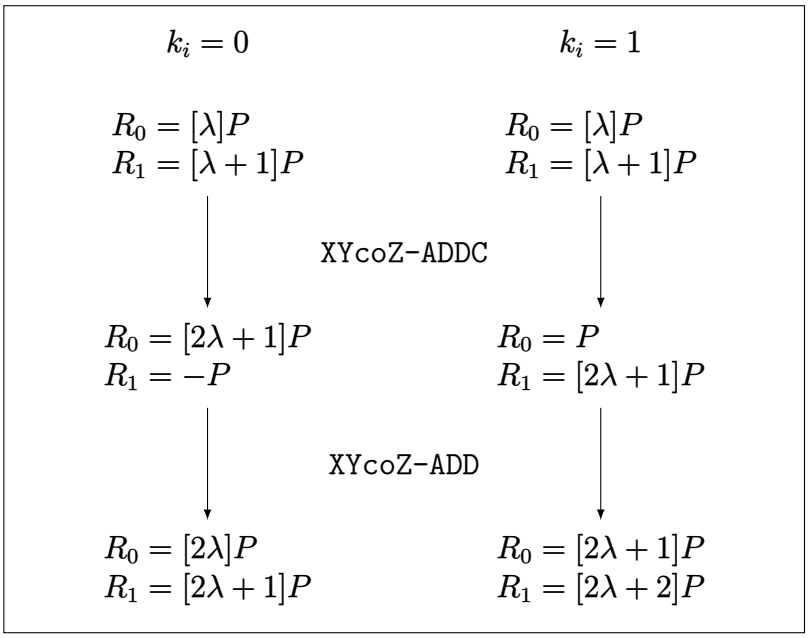


Figure 3.1 : Étape de Montgomery ladder avec les formules XYcoZ.

Algorithme 2 : Montgomery ladder

Entrée : $k = (k_{n-1}, \ldots, k_0)_2$, P, $k_{n-1} = 1$
Sortie : $Q = [k]P$

$R_0 \leftarrow P$
$R_1 \leftarrow [2]P$
pour $i = n-2$ **à** 0 **faire**
    $R_{1-k_i} \leftarrow R_0 + R_1$
    $R_{k_i} \leftarrow [2]R_{k_i}$
retourner R_0

Algorithme 3 : Montgomery ladder avec échanges conditionnels

Entrée : $k = (k_{n-1}, \ldots, k_0)_2$, P, $k_{n-1} = 1$
Sortie : $Q = [k]P$

$R_0 \leftarrow P$
$R_1 \leftarrow [2]P$
pour $i = n-2$ **à** 0 **faire**
    conditional_swap(k_i, R_0, R_1)
    $R_1 \leftarrow R_0 + R_1$
    $R_0 \leftarrow [2]R_0$
    conditional_swap(k_i, R_0, R_1)
retourner R_0

Algorithme 4 : Montgomery ladder avec un unique échange conditionnel par boucle

Entrée : $k = (k_{n-1}, \ldots, k_0)_2$, P, $k_{n-1} = 1$
Sortie : $Q = [k]P$

$R_0 \leftarrow P$
$R_1 \leftarrow [2]P$
pbit $\leftarrow 0$
pour $i = n-2$ **à** 0 **faire**
    pbit $\leftarrow$ pbit $\oplus\, k_i$
    conditional_swap(pbit, R_0, R_1)
    $R_1 \leftarrow R_0 + R_1$
    $R_0 \leftarrow [2]R_0$
    pbit $\leftarrow k_i$
conditional_swap(k_0, R_0, R_1)
retourner R_0

3.2.2 Nombre d'itérations fixe

Bien que l'algorithme exécute toujours les mêmes opérations à chaque itération, cela ne le rend pas pour autant à temps constant. Un nombre fixe d'itérations est nécessaire.

Plusieurs méthodes sont présentées ci-dessous.

Bourrage du scalaire. La méthode du bourrage [BT11] utilise l'ordre q du point de base pour cacher la longueur en bits du scalaire en entrée, qui vérifie l'inégalité $0 < k < q$. Soit $t = \lceil \log_2(q) \rceil$ la longueur en bits de q, alors le scalaire k est remplacé par k_{pad} défini ci-dessous :

$$k_{\text{pad}} = \begin{cases} k + q & \text{si } \lceil \log_2(k+q) \rceil = t + 1, \\ k + 2q & \text{sinon.} \end{cases}$$

Le résultat est un scalaire qui est toujours de longueur $(t+1)$ bits, donc il y a toujours t itérations de la boucle.

On peut encadrer le scalaire k_{pad} plus précisément. Si $k < 2^t - q$, alors $k_{\text{pad}} = k + 2q$ et $2q \leq k_{\text{pad}} < 2^t + q$. Tandis que si $k \geq 2^t - q$, alors $k_{\text{pad}} = k + q$ et $2^t \leq k_{\text{pad}} < 2q$. Globalement, le nouveau scalaire satisfait l'inégalité

$$2^t \leq k_{\text{pad}} < 2^t + q. \tag{3.3}$$

Vue que q est l'ordre du point de base, le résultat final de la multiplication scalaire est inchangé.

Clamping. Cette technique est spécifique pour les courbes sous forme de Montgomery. Prenons la courbe `Curve25519` comme exemple : le bit 2^{254} est fixé à 1 et tout bit supérieur est mis à zéro, de telle sorte que tout scalaire est un entier d'exactement 255 bits.

Cette méthode ne devrait pas être utilisée pour générée des signatures, car cela introduirait un biais exploitable pour résoudre le problème HNP (voir le chapitre 2).

La proposition d'Apple. Dans les versions les plus récentes de la bibliothèque cryptographique d'Apple (à partir de iOS 14.5 et macOS 11.3) l'implémentation de l'algorithme Montgomery ladder pour les courbes elliptiques a été modifiée pour gérer des scalaires de longueur variable (notamment pour corriger la vulnérabilité décrite dans le chapitre 5). Les idées générales sont décrites ci-dessous.

Dans la section 3.2.1 il a été décrit que l'état est initialisé par $(P, [2]P)$ sous l'hypothèse que le bit k_{n-1} est 1. Dans la situation où ce bit vaut 0, une négation conditionnelle peut être effectuée pour transformer l'état en $(-P, [2]P)$. Il devient $(-P, P)$ après l'addition de points, puis $([-2]P, P)$ après le doublement. Avec une autre négation et un échange, le couple (R_0, R_1) revient à son état originel. Ensuite, dès qu'un bit du scalaire vaut 1 alors la version classique de l'algorithme peut être exécutée.

Dans le code d'Apple, cela est réalisé avec plusieurs variables booléennes et est entièrement décrit dans l'algorithme 5, basé sur les formules `XYcoZ`. Cependant, il est assez difficile de vérifier la validité sous cette forme. Pour voir le comportement, on note $(R_0, R_1), (\text{nbit}, \text{zbit}, \text{pbit})$ l'état des variables à chaque itération. L'évolution de cet état en fonction des bits du scalaire est donnée dans la figure 3.2.

Algorithme 5 : Montgomery ladder avec les formules `XYcoZ`

Entrée : $P, k = (k_{n-1}, \ldots, k_0)_2, 1 < k < q - 1$
Sortie : $[k]P$

1: $(R_0, R_1) \leftarrow$ `XYcoZ-DBL`(P) ▷ $(R_0, R_1) = ([2]P, P)$
2: $(\text{nbit}, \text{pbit}, \text{zbit}) \leftarrow (k_{n-1} \oplus 1, k_{n-1}, k_{n-1})$
3: **pour** $i = n - 2$ **à** 1 **faire**
4: conditional_neg(nbit, R_1) ▷ $R_1 \leftarrow -R_1$ si nbit $= 1$
5: conditional_swap((pbit $\oplus$ k_i) $\vee$ (zbit $\oplus$ 1), R_0, R_1)
6: $(R_0, R_1) \leftarrow$ `XYcoZ-ADDC`(R_0, R_1) ▷ $(R_0, R_1) \leftarrow (R_0 + R_1, R_0 - R_1)$
7: $(R_0, R_1) \leftarrow$ `XYcoZ-ADD`(R_0, R_1) ▷ $(R_0, R_1) \leftarrow (R_0 + R_1, R_0')$
8: nbit $\leftarrow$ (zbit $\oplus$ 1) $\wedge$ k_i
9: zbit $\leftarrow$ zbit $\vee$ k_i
10: pbit $\leftarrow$ k_i
11: conditional_neg(nbit, R_1)
12: conditional_swap(pbit $\oplus$ k_0, R_0, R_1)
13: $(R_0, R_1) \leftarrow$ `XYcoZ-ADDC`(R_0, R_1)
14: $R_b \leftarrow R_1$
15: conditional_swap(k_0, R_0, R_1)
16: $Z, \lambda_x, \lambda_y \leftarrow$ `XYcoZ-recoverZ`(R_0, R_1, R_b)
17: conditional_swap(k_0, R_0, R_1)
18: $(R_0, R_1) \leftarrow$ `XYcoZ-ADD`(R_0, R_1)
19: conditional_swap(k_0, R_0, R_1)
20: **retourner** $(\lambda_x \cdot X(R_0), \lambda_y \cdot Y(R_0), Z)$

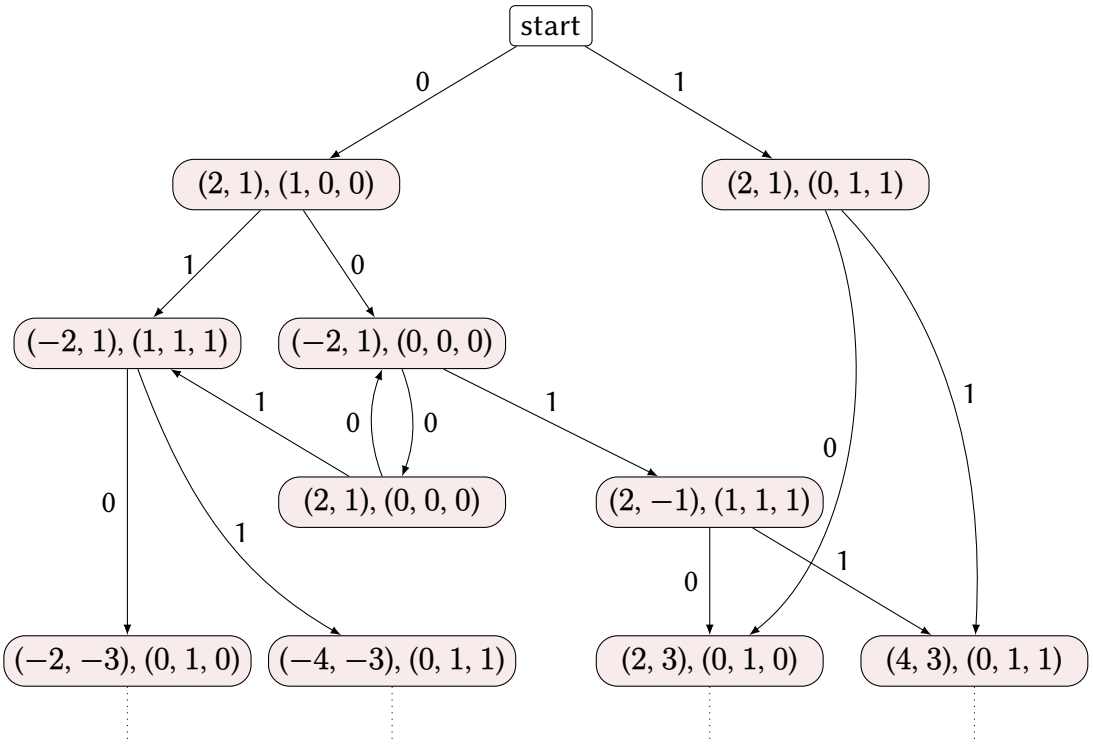


Figure 3.2 : État interne des variables (R_0, R_1) et (nbit, zbit, pbit) dans la variante de Montgomery ladder de l'algorithme 5.

On aperçoit clairement la boucle tant que que les bits du scalaires rencontrés sont 0. Cependant, dès qu'un bit vaut 1, et que le bit qui suit a été traité alors les variables nbit et zbit prennent une valeur fixe jusqu'à la fin de l'exécution. On arrive dans une des deux situations suivantes :

- Le nombre de bits nuls avant le premier bit 1 est pair, et on retrouve la situation classique de l'algorithme ;
- Le nombre de bits nuls avant le premier bit 1 est impair, alors la situation est similaire, à ceci près de la présence d'un facteur -1 pour les points R_0 et R_1 : par exemple, si les trois premiers bits sont $(0, 1, 0)$, on obtient $R_0 = [-2]P$ et $R_1 = [-3]P$.

Dans le second cas, continuer l'exécution de l'algorithme aboutit à $R_0 = [-k]P$ qui n'est pas le résultat attendu. Cependant, un effet est produit lors de la reconstruction de la coordonnée Z. En effet, la fonction `XYcoZ-recoverZ` reconstruit la coordonnée partagée des points R_0 et R_1, en utilisant l'invariant qui apparaît après le calcul de $R_0 - R_1$ dans la fonction `XYcoZ-ADDC`. Ce point est utilisé dans l'équation (3.1) sous l'hypothèse qu'il s'agisse de l'invariant attendu qui est P. Dans la situation où le nombre de bits nuls est impair avant le premier bit à 1, les deux points R_0 et R_1 sont tous deux changés en leur opposé, donc la valeur utilisée à la place de l'invariant est $-P$. Par conséquent, la coordonnée reconstruite de $[-k]P$ en fin d'algorithme est $-Z$. En notant (x, y) la représentation affine de ce point et $(X : Y : Z)$ celle en coordonnée projective Jacobienne complète, alors sa forme affine sera calculée comme

$$\left(\frac{X}{(-Z)^2}, \frac{Y}{(-Z)^3}\right) = (x, -y) = [k]P.$$

Ainsi le facteur -1 est corrigé automatiquement et la multiplication scalaire est correcte.

3.2.3 Les situations exceptionnelles

On analyse dans cette partie l'apparition ou non des cas exceptionnels dans les formules décrites dans la section précédente. En premier lieu, on présente des exceptions communes.

Comme les points R_0 et R_1 satisfont l'invariant $R_1 - R_0 = P$ où P est le point d'entrée de l'algorithme, alors l'exception $R_0 = R_1$ n'est possible que si P est le point à l'infini $\mathcal{O}$.

Une autre exception est le cas $R_0 = -R_1$. Supposons que l'exception se produise lors du traitement du bit k_j. Il y a deux cas :

- k_j vaut 0, donc le résultat de l'étape est $(R_0, R_1) = (-P, \mathcal{O})$; ainsi les bits de poids fort valent $\lambda q - 1$ pour un entier $\lambda > 0$ impair (pour des raisons de parité), donc les scalaires exceptionnels sont

$$k \in \{\, (\lambda q - 1)2^j + \mu \mid \lambda > 0 \text{ impair},\ 0 \leq \mu < 2^j \,\}. \tag{3.4}$$

- k_j vaut 1, donc le résultat de l'étape est $(R_0, R_1) = (\mathcal{O}, P)$; ainsi les bits de poids fort valent λq pour un entier $\lambda > 0$ impair (pour des raisons de parité), donc

$$k \in \{\, \lambda q 2^j + \mu \mid \lambda > 0 \text{ impair},\ 0 \leq \mu < 2^j \,\}. \tag{3.5}$$

La position j où l'exception intervient dépend directement de la taille du scalaire. Ainsi, lorsque k est plus petit que l'ordre q du point de base l'exception ne peut se produire que lors de la dernière itération et pour le scalaire $q-1$. Si la méthode de bourrage donnée en section 3.2.2 est utilisée, l'exception va se produire dans la pénultième itération pour les scalaires $2q-2$, $2q-1$, $2q$ et $2q+1$, dont les valeurs avant le bourrage sont respectivement $q-2$, $q-1$, 0 et 1.

Un résumé de l'analyse pour les cas spécifiques est donné dans le tableau 3.2, et dans les paragraphes ci-dessous.

Table 3.2 : Entrées exceptionnelles pour l'algorithme Montgomery ladder.

Formules		Scalaires except.	Points except.
Jac XYZ	$0 \leq k \leq q$	0	$\mathcal{O}$
	Bourrage	$0, 1, q-2, q-1$ (†)	–
	$k > q$	Eq. (3.4) et (3.5)	–
XYcoZ	$0 \leq k \leq q$	$0, q-1, q$	$\mathcal{O}$, $(0, \pm\sqrt{B})$
	Bourrage	$0, 1, q-2, q-1$ (†)	–
	$k > q$	Eq. (3.4) et (3.5)	–
XZ	$k \geq 0$	$0, q-1$ (†) (‡)	$\mathcal{O}$

(†) : valeur du scalaire après réduction dans $[0, q-1]$

(‡) : uniquement s'il y a reconstruction de la coordonnée y

Exceptions avec les formules Jacobiennes XYZ. La situation $R_0 = -R_1$ n'est pas une exception pour ces formules, mais son résultat est le point à l'infini $\mathcal{O}$ qui ne peut être utilisé dans les additions de points suivantes. On a vu que cela se produit à l'étape j pour les scalaires donnés dans les équations (3.4) et (3.5), ce qui provoque une exception pendant le traitement du bit k_{j-1}. Ces scalaires ne sont donc pas nécessairement exceptionnels pour $j = 0$ comme par exemple les scalaires $q-1$ et q.

Exceptions avec les formules Jacobiennes XYcoZ. Par rapport aux formules utilisant les coordonnées complètes, les formules XYcoZ ont davantage d'exceptions, notamment toutes celles des équations (3.4) et (3.5).

Il y a également d'autres points exceptionnels en plus du point à l'infini en entrée d'algorithme. Cela est dû à la récupération de la coordonnée manquante Z à la fin de la multiplication scalaire. En effet, la formule de l'équation (3.1) requiert que le point P n'ait pas de coordonnée nulle : donc les points $(0, \pm\sqrt{b})$ sont exceptionnels pour les courbes elliptiques ayant de tels points.

Exceptions avec les formules projectives XZ. Les seules exceptions avec ces formules ont lieu lors de la récupération de la coordonnée y de la sortie (si celle-ci est nécessaire). Cela se produit alors lorsque R_0 ou R_1 est le point à l'infini après la dernière itération, donc lorsque $k \bmod q$ vaut 0 ou $q-1$.

3.3 Algorithmes à base de fenêtres

Cette section introduit les multiplications scalaires utilisant des précalculs. En particulier, deux classes d'entre elles :

- Les méthodes fenêtrées qui traitent le scalaire par morceaux composés d'un nombre fixe de bits consécutifs ;
- Les méthodes *comb* (« peigne ») qui prennent des bits non-consecutifs séparés d'une distance fixe.

Le fonctionnement repose sur un accumulateur additionné avec des points précalculés qui dépendent des fenêtres extraites du scalaire, jusqu'à ce qu'il soit entièrement traité.

Dans la suite on suppose que les formules avec coordonnées Jacobiennes sont utilisées, dont les exceptions sont le cas d'égalité et du point à l'infini au cours de l'addition de points.

Remarque 3. Il existe d'autres algorithmes dont les fenêtres ne sont pas de tailles fixes. Un exemple est l'algorithme la représentation NAF (*Non-Adjacent Form*) sous sa forme fenêtrée. L'inconvénient de cette méthode est que les positions des fenêtres non nulles avec cet encodage sont variables. Son utilisation est alors vulnérable aux attaques par canaux auxiliaires comme il le fut montré par exemple dans [MPP20].

3.3.1 Versions classiques

La version simple de la multiplication scalaire avec des fenêtres de tailles fixes est donnée dans l'algorithme 6. L'inconvénient majeur de ces méthodes est le traitement des fenêtres nulles, c'est-à-dire lorsque tous les bits qui constituent la fenêtre valent zéro. Ces situations correspondent à l'addition entre l'accumulateur et le point à l'infini, ainsi cette exception apparaît fréquemment pour les formules affines ou pour coordonnées Jacobiennes. Il est alors impératif de gérer cette situation ce qui peut être fait notamment par l'introduction d'additions factices. Les conséquences de ce choix sont explorées dans le chapitre 4.

Cependant, lorsque le scalaire est plus petit que q, alors les exceptions d'égalité de ou de points opposés ne peuvent se produire.

Algorithme 6 : Multiplication scalaire fenêtrée simple

Entrée : $P, k = (k_{n-1}, \dots k_0)_2, w, P_i = [i]P$ avec $0 \leq i < 2^w$
Sortie : $[k]P$

1: $R \leftarrow \mathcal{O}$
2: **pour** $i \leftarrow n-1$ **à** 0 **faire**
3: $\quad R \leftarrow [2]R$
4: $\quad$ **si** $i \bmod w = 0$ **alors**
5: $\quad\quad d = (k_{i+w-1}, \dots, k_i)$
6: $\quad\quad R \leftarrow R + P_d$
7: **retourner** R

3.3.2 Versions avec réencodage du scalaire

Plusieurs variantes existent et consistent à réencoder le scalaire. Dans certains cas pour des raisons d'efficacité (en s'appuyant sur la négation de point presque gratuite en coût d'exécution), tandis que d'autres ont pour but d'éviter à devoir gérer le point à l'infini [Möl01, HPB04, OT03].

Notons en passant que l'utilisation de scalaires plus petits que l'ordre du point de base permet aussi de gérer certaines des autres exceptions.

Réencodage de Booth. Cet algorithme est une variante qui utilise la négation de point en passant par le réencodage des fenêtres en une valeur relative : chaque fenêtre est formée à partir de $w+1$ bits consécutifs et convertie en entier dans l'intervalle $[-2^{w-1}, 2^{w-1}]$. Cette méthode est basée sur le réencodage de Booth [Boo51] et est donnée dans l'algorithme 7.

Algorithme 7 : Réencodage de Booth

Entrée : d, w, avec $0 \leq d < 2^{w+1}$
Sortie : Codage de d en (s, d') avec $sd' \in [-2^{w-1}, 2^{w-1}]$

1: **si** $d \geq 2^w$ **alors**
2: $\quad d \leftarrow 2^{w+1} - 1 - d$
3: $\quad s \leftarrow 1$
4: **sinon**
5: $\quad s \leftarrow 0$
6: **retourner** s, $\lfloor (d+1)/2 \rfloor$

Un scalaire $k = (k_{n-1}, \ldots, k_0)_2$ est divisé en $m = \lceil n/w \rceil$ fenêtres et réécrit

$$k = \sum_{i=0}^{m-1} (-1)^{s_i} d_i 2^{iw}, \quad (s_i, d_i) = \mathsf{Booth}((k_{iw-(w-1)}, \ldots, k_{iw}, k_{iw-1})_2),$$

où les bits k_i valent 0 pour $i = -1$ ou $i \geq n$. On peut remarquer que chaque fenêtre est composée d'un bit de la fenêtre qui précède et celle qui suit. L'algorithme de multiplication scalaire traite chaque fenêtre l'une après l'autre, dont l'ordre est généralement de la gauche vers la droite comme dans l'algorithme 8 (un exemple de la droite vers la gauche est donné dans la section 4.1.4).

Algorithme 8 : Multiplication scalaire fenêtrée avec le réencodage de Booth

Entrée : $k = (k_{n-1}, \ldots, k_0)_2$, P, w, $P_i = [i]P$ pour $0 \leq i \leq 2^{w-1}$
Sortie : $[k]P$

1: $R \leftarrow \mathcal{O}$
2: **pour** $i \leftarrow \lceil n/w \rceil - 1$ **à** 0 **faire**
3: $\quad R \leftarrow [2^w]R$
4: $\quad s_i, d_i \leftarrow \mathsf{Booth}((k_{iw+(w-1)}, \ldots, k_{iw}, k_{iw-1})_2)$
5: $\quad R \leftarrow R + (-1)^{s_i} P_{d_i}$
6: **retourner** R

Cependant, cet encodage n'empêche pas les fenêtres nulles. En effet, les valeurs 0 et $2^{w+1}-1$ sont toutes les deux converties vers la valeur 0, donc il y a un nombre conséquent de scalaires provoquant l'exception de l'addition avec le point à l'infini.

Il est également possible que l'exception d'égalité se produise au cours de l'exécution, nécessitant l'utilisation des formules de doublement.

Proposition 1. *Soit k un scalaire tel que $0 < k < q$. Le cas de l'égalité dans l'addition de la ligne 5 de l'algorithme 8 (en dehors de l'égalité avec le point à l'infini) ne peut se produire qu'au cours de la dernière itération pour le scalaire $k = q - 2(q \bmod 2^w)$ lorsque $q \bmod 2^w \leq 2^{w-1}$.*

Démonstration. Supposons que l'égalité $R = [(-1)^{s_0} d_0]P$ est vraie lors de la dernière itération, nécessitant un doublement de point en place de l'addition. Cela signifie que $k \equiv (-1)^{s_0} 2d_0 \bmod q$. Si s_0 vaut 0, alors cela devient l'égalité $k = 2d_0$, mais d'après l'encodage on obtiendrait $d_0 = k \bmod 2^w$, et on en déduit que $k = d_0 = 0$. Maintenant, si s_0 vaut 1, alors on obtiendrait l'égalité $k = q - 2d_0$. Il faut alors que la dernière fenêtre réencodée de $q - 2d_0$ soit $-d_0$. Cela se produit si $q \bmod 2^w$ est un entier inférieur ou égal à 2^{w-1} et le scalaire correspond à $k = q - 2(q \bmod 2^w)$. En effet, la fenêtre avant l'encodage est $2^{w+1} - 2(q \bmod 2^w)$, et est plus grand que 2^w, donc l'encodage donne $d_0 = q \bmod 2^w$ et $s_0 = 1$. Par conséquent, on a $k = q - 2d_0$ et l'exception d'égalité apparaît lors de la dernière addition de points.

Il reste à montrer que l'égalité ne peut arriver dans un tour précédent. Soit j tel que $1 \leq j \leq m-2$ et $R = [(-1)^{s_j} d_j]P$, alors on a

$$(-1)^{s_j} d_j \equiv \sum_{i=j+1}^{m-1} (-1)^{s_i} d_i 2^{w(i-j)} \mod q,$$

d'où la relation

$$k \equiv \sum_{i=0}^{j-1} (-1)^{s_i} d_i 2^{iw} + (-1)^{s_j} d_j 2^{iw+1} \mod q.$$

L'expression de droite peut être prouvée inférieure à q en valeur absolue. Si elle est négative alors on aurait

$$k = q + \sum_{i=0}^{j-1} (-1)^{s_i} d_i 2^{wl} + (-1)^{s_j} d_j 2^{wj+1},$$

donc $k \equiv 1 + d_0 \bmod 2$, mais on a aussi $k \equiv d_0 \bmod 2$, donc c'est impossible. Au contraire, si l'expression est positive alors

$$k = \sum_{i=0}^{j-1} 2^{iw} (-1)^{s_i} d_i + (-1)^{s_j} 2^{jw+1},$$

donc $(-1)^{s_j} d_j = \sum_{i=j+1}^{m-1} (-1)^{s_i} d_i 2^{w(i-j)}$. La partie droite est un multiple de 2^w donc l'égalité est vraie seulement si $d_i = 0$ pour i dans l'intervalle $[j, m-1]$. Il s'agit donc de l'exception avec le point à l'infini dans toutes les additions précédentes. □

Des exemples de scalaires exceptionnels pour le cas de l'égalité dans la dernière itération sont donnés dans le tableau 3.3.

Table 3.3 : Scalaires exceptionnels pour la multiplication scalaire fenêtrée avec le réencodage de Booth.

Courbe	3	4	5	6	7
secp224r1					$q-122$
secp256r1	$q-1$	$q-2$		$q-34$	
secp521r1	$q-2$		$q-18$	$q-18$	$q-18$

Méthode comb avec réencodage relatif impair. Cet algorithme est une variante de la méthode *comb*, où chaque fenêtre est codée telle quelle est toujours impaire et peut être négative. L'encodage est une variante de celui proposée dans [HPB04] et est notamment utilisé par Mbed TLS (voir section 3.4.5).

Un inconvénient est qu'il est nécessaire que le scalaire en entrée k soit impair, car la première fenêtre *comb* doit être impaire et contient le bit de poids faible de k. Mais si k est pair, alors $q-k$ est impair, donc cela peut être géré avec précaution avec une sélection conditionnelle sécurisée au début de l'algorithme. Pour la suite de cette partie on suppose que k est impair et dans l'intervalle $[1,q]$.

Le découpage du scalaire k en fenêtres *comb* peut être vu ci-dessous :

$$\begin{array}{rccccccccc} k = & & k_0 2^0 & + & k_1 2^1 & + & \cdots & + & k_{m-1}2^{m-1} \\ & + & k_m 2^m & + & k_{m+1}2^{m+1} & + & \cdots & + & k_{2m-1}2^{2m-1} \\ & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ & + & k_{m(w-1)}2^{m(w-1)} & + & k_{m(w-1)+1}2^{m(w-1)+1} & + & \cdots & + & k_{mw-1}2^{mw-1} \end{array} \tag{3.6}$$

Chaque colonne forme une fenêtre de w bits, tous séparés par la même distance $m = \lceil n/w \rceil$. On note $K[i]$ la colonne en position i (celle dont le bit de poids faible est k_i), et le scalaire est réécrit :

$$k = \sum_{i=0}^{m-1} K[i]2^i.$$

La première fenêtre $K[0]$ est impaire, et la méthode suivante encode les suivantes en valeur impaire. Supposons que les fenêtres $K[j]$ sont impaires pour $0 \leq j \leq i-1$. Si la fenêtre $K[i]$ est paire, alors on ajoute les bits représentant $K[i-1]$ à ceux représentant $K[i]$ bit par bit. Les retenues sont sauvegardées pour chacune des w positions, et la fenêtre $K[i-1]$ est changée en $-K[i-1]$. Les retenues sont ajoutées à la fenêtre $K[i+1]$. Cette opération signifie que $K[i-1]+2K[i]$ est changé en $-K[i-1]+2(K[i-1]+K[i])$ dans l'équation (3.3.2) ce qui n'en change pas la valeur. Les retenues se propagent jusqu'à atteindre une nouvelle fenêtre $K[m]$ qui est positive.

En notant $K'[i]$ les valeurs des nouvelles fenêtres *comb* et s_i un indicateur du signe (avec 0 pour positif et 1 pour négatif), le scalaire réencodé est

$$k = \sum_{i=0}^{m} (-1)^{s_i} K'[i]2^i,$$

où $K'[i]$ est impair pour $0 \leq i \leq m$.

La multiplication scalaire complète est décrite dans l'algorithme 9.

Algorithme 9 : Multiplication scalaire *comb* et réencodage signé impair

Entrée : $k = (k_{n-1}, \ldots, k_0)_2$, P, w
Sortie : $[k]P$

Phase de précalculs
pour $d \leftarrow 1$ **à** $2^w - 1$ impair **faire**
  $P_d \leftarrow [\sum_{i=0}^{w-1} d_i 2^{im}]P$ ▷ $d = (d_{w-1}, \ldots, d_0)_2$

Phase d'encodage
si k est pair **alors**
  $k' \leftarrow q - k$
sinon
  $k' \leftarrow k$
$(s_0, d_0), \ldots, (s_m, d_m) \leftarrow \mathsf{Encodage}(k')$ ▷ $d_i = (d_{i,w-1}, \ldots, d_{i,0})_2$ et $K'[i] = \sum_{j=0}^{w-1} d_{i,j} 2^{jm}$

Phase d'évaluation
$R \leftarrow P_{d_m}$
pour $i \leftarrow m - 1$ **à** 0 **faire**
  $R \leftarrow [2]R$
  $R \leftarrow R + (-1)^{s_i} P_{d_i}$
si k est pair **alors**
  $y(R) \leftarrow -y(R)$
retourner R

La suite est dédiée à la preuve que les exceptions des formules pour coordonnées Jacobiennes ne peuvent se produire que sous certaines conditions assez restrictives.

Proposition 2. *Si l'ordre q du point de base satisfait $q \equiv 1 \bmod 4$ et $m \geq 2w + 5$, alors il n'y a aucune exception avec les formules pour coordonnées Jacobiennes.*

Démonstration. Pour simplifier les notations, le signe des fenêtres est inclus directement dans la notation $C_i = (-1)^{s_i} K'[i]$. On note M_0 et M_1 les valeurs telles que $k = M_0 + 2^j M_1$ où $M_0 = \sum_{i=0}^{j-1} C_i 2^i$ et $M_1 = \sum_{i=j}^{m} C_i 2^{i-j}$. Les deux bornes suivantes sont utiles pour la preuve :

$$1 \leq |C_i| < 2^{(w-1)m+1} \quad \text{et} \quad 2^{(m-1)w} < q < 2^{mw}.$$

Apparition du point à l'infini dans une itération. Supposons que $j \geq 1$ et que le résultat de l'addition au cours de cette itération est le point à l'infini. Cela signifie qu'on a l'égalité $R = -[C_j]P$, ce qui donne la relation $M_1 \equiv 0 \bmod q$, ainsi que la majoration $|M_1| < 2^{mw-j+2}$.

Si $j \geq w+2$, cette majoration devient $|M_1| < q$, donc M_1 est nul, ce qui est impossible car M_1 est impair. Maintenant supposons que $j < w + 2$. On a $k \equiv M_0 \bmod q$ et la majoration $|M_0| < 2^{(w-1)m+w+3}$. Vu qu'on a supposé que $m \geq 2w + 5$, alors on obtient $|M_0| < q$. Alors soit $k = M_0$ ce qui implique que M_1 est nul, soit $k = M_0 + q$ ce qui est impossible pour des raisons de parité.

Le résultat de l'addition ne peut donc pas être le point à l'infini, excepté lors de la dernière itération, ce qui ne peut arriver que si le scalaire en entrée est q, ce qui serait le résultat attendu.

Cas de l'égalité dans la dernière itération. Avant l'addition dans la dernière itération de la boucle, on a $R = [\sum_{i=1}^{m} C_i 2^i]P$ et ne peut être le point à l'infini comme il vient d'être montré ci-dessus. La seule exception serait le cas où $R = [C_0]P$, et donc $k \equiv 2C_0 \bmod q$.

Comme m est suffisamment large, on a $|2C_0| < q$, alors soit $k = 2C_0$ ou soit $k = q + 2C_0$. Le premier cas est impossible puisque k est impair. Dans le second, C_0 est négatif, donc la fenêtre non encodée $K[1]$ est paire selon le procédé d'encodage : le second bit de poids faible k_1 vaut 0. Ainsi on a $k \equiv 1 \bmod 4$, et on en déduit que $q \equiv 3 \bmod 4$.

Si l'ordre q satisfait cette condition, il est possible qu'un scalaire produise l'exception au cours de la dernière itération.

Cas de l'égalité dans une précédente itération. Supposons que l'égalité se produise dans une boucle précédente, c'est-à-dire que $R = [C_j]P$ pour un valeur $j \geq 1$, et donne la relation $M_1 \equiv 2C_j \bmod q$. On obtient la borne $|M_1 - 2C_j| < 2^{mw-j+2}$ à partir de celles sur C_i et q.

Si $j \geq w + 2$, alors on a $|M_1 - 2C_j| < q$, et donc $M_1 = 2C_j$ ce qui est impossible pour des raisons de parité. Maintenant supposons $j < w + 2$. On a $k \equiv M_0 + 2^{j+1}C_j \bmod q$, et la borne $|M_0 + 2^{j+1}C_j| < 2^{(w-1)m+w+5}$. Comme il a été supposé que $m \geq 2w + 5$, on a donc $|M_0 + 2^{j+1}C_j| < q$. Alors soit $k = M_0 + 2^{j+1}C_j$ qui implique l'égalité $M_1 = 2C_j$, ou soit $k = M_0 + 2^{j+1}C_j + q$, ce qui est impossible dans les deux cas pour cause de parité. □

3.4 Vue d'ensemble des bibliothèques

Dans cette section, plusieurs bibliothèques cryptographiques sont analysées sous l'angle de la gestion des exceptions dans les formules d'additions. Les algorithmes, formules et éventuellement autres techniques utilisées sont données pour chacune d'elles pour la réalisation d'une multiplication scalaire simple avec soit un point fixe en entrée (le point de base donné avec les paramètres), soit un point variable.

Pour rester concis, seules les courbes sous forme de Weierstraß courtes sur un corps premier sont prises en compte. Une vue d'ensemble des bibliothèques est donnée dans le tableau 3.4 avec la nomenclature suivante :

- Algorithmes : ladder (Montgomery ladder) ; wBooth (méthode fenêtrée avec rééencodage de Booth) wSimple (méthode fenêtrée simple) ; combOddSigned (méthode *comb* avec réencodage impair relatif) ;
- Courbes : NIST optm (implémentation optimisée des courbes du NIST) ; `secp256r1` asm (implémentation optimisée partiellement en assembleur de la courbe `secp256r1`).

3.4.1 Multiplication scalaire dans OpenSSL

La version analysée est OpenSSL 1.1.1k.[2] Plusieurs primitives cryptographiques sont présentes dans OpenSSL nécessitant une multiplication scalaire simple où le scalaire est secret : ECDH, ECDSA, SM2. Les courbes elliptiques présentes sont elles aussi nombreuses : NIST, SECG, X9_62, WPA/WTLS, Brainpool et SM2.

2. https://www.openssl.org/source/.

Table 3.4 : Vue d'ensemble des algorithmes et formules pour la multiplication scalaire dans les bibliothèques cryptographiques.

Bibliothèque	Courbe	Point fixe		Point variable	
OpenSSL	défaut	ladder	XZ	ladder	XZ
(1.1.1k)	NIST optm	comb	Jac	wBooth	Jac
	secp256r1 asm	wBooth	Jac	wBooth	Jac
BoringSSL	défaut	wSimple	Jac	wSimple	Jac
	NIST optm	comb	Jac	wBooth	Jac
	secp256r1 asm	wBooth	Jac	wBooth	Jac
LibreSSL	défaut	ladder	Jac	ladder	Jac
CoreCrypto	défaut	ladder	XYcoZ	ladder	XYcoZ
Mbed TLS	défaut	combOddSigned	Jac	combOddSigned	Jac

Par défaut, l'algorithme est Montgomery ladder avec les formules projectives homogènes vues en section 3.1.4, ainsi que la méthode de bourrage. D'après le tableau 3.2 les scalaires 0 et $q-1$ produisent une exception pour la reconstruction de la coordonnée affine y et ce cas est géré par des branchements conditionnels.

Les courbes du NIST secp224r1, secp256r1 et secp521r1 possèdent une implémentation spécifique et optimisée pour une architecture 64 bits. Celles-ci utilisent la méthode *comb* avec $w = 4$ lorsque le point en entrée est fixe, et la méthode fenêtrée avec rééencodage de Booth et $w = 5$ lorsque le point d'entrée est variable. Dans les deux cas, l'exception liée aux fenêtres nulles est gérée avec une addition factice de points, dont les conséquences pour la sécurité est abordée dans le chapitre 4. Quant à l'exception liée au cas d'égalité qui ne peut intervenir que dans la dernière itération de la boucle avec le réencodage de Booth, on peut noter que le choix de fenêtre implique que cela ne peut jamais se produire pour les courbes secp224r1 et secp256r1. Par contre le scalaire $q-18$ produit l'exception pour la courbe secp521r1 et cela est attesté par les commentaires du code source [3] (voir listing 3.1).

La courbe secp256r1 possède également une implémentation spécifique supplémentaire utilisée par défaut (nécessite une option à la compilation pour désactiver). Elle est issue des travaux présentés dans [GK15] où l'arithmétique modulaire est implémentée en assembleur et optimisée en tirant profit du nombre premier spécifique qui définit le corps fini pour cette courbe. La différence principale est lorsque le point d'entrée est fixe, l'algorithme avec réencodage de Booth est utilisé avec une taille de fenêtre $w = 7$ et en traitant les fenêtres dans l'autre sens.

3.4.2 Multiplication scalaire dans BoringSSL

Les implémentations spécifiques des courbes du NIST sont les mêmes que celles présentes dans OpenSSL, avec une partie du code réécrit pour séparer en plusieurs fonctions la multiplication scalaire simple des multiplications scalaires à plusieurs points.

3. https://github.com/openssl/openssl/blob/master/crypto/ec/ecp_nistp521.c.

```
if (points_equal) {
    /*
     * This is obviously not constant-time but it will almost-never happen
     * for ECDH / ECDSA. The case where it can happen is during scalar-mult
     * where the intermediate value gets very close to the group order.
     * Since |ossl_ec_GFp_nistp_recode_scalar_bits| produces signed digits
     * for the scalar, it's possible for the intermediate value to be a small
     * negative multiple of the base point, and for the final signed digit
     * to be the same value. We believe that this only occurs for the scalar
     * 1ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
     * ffffffa51868783bf2f966b7fcc0148f709a5d03bb5c9b8899c47aebb6fb
     * 71e913863f7, in that case the penultimate intermediate is -9G and
     * the final digit is also -9G. Since this only happens for a single
     * scalar, the timing leak is irrelevant. (Any attacker who wanted to
     * check whether a secret scalar was that exact value, can already do
     * so.)
     */
    point_double(x3, y3, z3, x1, y1, z1);
    return;
}
```

Listing 3.1 : Cas exceptionel d'égalité dans OpenSSL pour la courbe elliptique secp521r1.

En conséquence, la gestion de l'exception des fenêtres nulles est identique à OpenSSL, et cela en est de même pour l'algorithme par défaut qui est une méthode fenêtrée simple (avec taille $w = 5$ des fenêtres).

Par rapport à OpenSSL, il est à noter que seules les implémentations spécifiques optimisées des courbes `secp224r1` et `secp256r1` sont présentes. Les autres courbes intégrées sont `secp384r1` et `secp521r1`.

3.4.3 Multiplication scalaire dans LibreSSL

Les courbes présentes sont essentiellement les mêmes que pour OpenSSL avec une addition (la courbe `frp256v1` de l'ANSSI).

L'algorithme par défaut est Montgomery ladder, mais contrairement à OpenSSL ce sont les formules pour coordonnées Jacobiennes qui sont utilisées. Les rares cas exceptionnels du tableau 3.2 sont gérés par des branchements conditionnels.

3.4.4 Multiplication scalaire dans CoreCrypto

Cette bibliothèque réalise les opérations cryptographiques dans les produits Apple, dont le code source est mis à disposition.[4] Cette partie décrit des propriétés de la bibliothèque avant l'application de correctifs introduits à partir de iOS 14.5 and macOS 11.3.

Les courbes présentes sont celles du NIST : `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1` et `secp521r1`.

L'algorithme Montgomery ladder est utilisé avec les formules `XYcoZ`. La méthode de bourrage est appliquée, puis le scalaire est scindé en trois morceaux par une division Euclidienne. Trois multiplications scalaires sont exécutées avec le quotient, le diviseur et le reste.

Les rares cas particuliers sont gérés directement par l'utilisation d'une méthode de randomisation qui scinde le scalaire en plusieurs morceaux dont aucun ne peut correspondre

4. https://developer.apple.com/security/ (lien en bas de page)

aux scalaires exceptionnels.

Le chapitre 6 est consacré à ce sujet, car un des scalaires créé par cette méthode a une taille en bits variable, ce qui casse la propriété nécessaire pour une exécution à temps constant.

Pour finir, il y a les points de coordonnée affine x nulle qui sont des points d'entrée exceptionnels. Les utiliser en entrée de la multiplication scalaire génère une erreur.

Il est conseillé de lire la section 5.3 avant de lire ce paragraphe « danger ». Il y a un cas exceptionnel supplémentaire qui peut se produire avec la méthode de randomisation. La méthode de bourrage créé le scalaire $k_{\text{pad}} = k + \varepsilon q - 2^{32}$ où ε vaut 1 ou 2, puis celui-ci est scindé par une division Euclidienne par un entier m aléatoire de 32 bits :

$$k_{\text{pad}} = am + b$$

Un premier point $R = [am]P$ est calculé et un second $S = [b + 2^{32}]$. Une addition finale $R + S$ est nécessaire pour obtenir le bon résultat. C'est lors de cette addition qu'une égalité est possible. Par exemple sur la courbe `secp224r1` avec

$$k = 8816999876 \quad \text{et} \quad m = 2267504508.$$

La division Euclidienne donne

$$\begin{cases} a = 237793985167685847834857006485118978563528463021908062479 45, \\ b = 113532642, \end{cases}$$

et on obtient l'égalité $am = b + 2^{32} + 2q$, ce qui provoque l'exception et nécessite un doublement de point.

De toute évidence il est peu probable que cette situation arrive : il faut que le scalaire k soit extrêmement petit et que la valeur m générée soit particulière.

3.4.5 Multiplication scalaire dans Mbed TLS

Les courbes elliptiques présentes sont celles du NIST, SECG et Brainpool. L'algorithme de multiplication scalaire est réalisé par la méthode *comb* avec un réencodage relatif impair donné dans la section 3.3.2.

D'après la proposition 2, le cas de l'égalité dans la dernière itération est possible sous certaines conditions, contrairement à ce qui est annoncé en commentaire du code source [5] (voir listing 3.2).

```
/*
 * Addition: R = P + Q, mixed affine-Jacobian coordinates (GECC 3.22)
 * (...)
 * Special cases: (1) P or Q is zero, (2) R is zero, (3) P == Q.
 * None of these cases can happen as intermediate step in ecp_mul_comb():
 * - at each step, P, Q and R are multiples of the base point, the factor
 *   being less than its order, so none of them is zero;
 * - Q is an odd multiple of the base point, P an even multiple,
 *   due to the choice of precomputed points in the modified comb method.
 * So branches for these cases do not leak secret information.
 * (...)
 */
```

Listing 3.2 : Mbed TLS sur l'addition de points : le cas (3) est possible malgré le commentaire.

5. https://github.com/ARMmbed/mbedtls/blob/development/library/ecp.c.

Un exemple est la courbe elliptique `secp384r1` dont le cardinal satisfait $q \equiv 3 \bmod 4$. La taille de fenêtre est $w = 6$ donc chaque fenêtre *comb* est composée de bits séparés d'une distance de 64 bits avant le réencodage. Alors le scalaire

$$k = q - 2(1 + 2^{128} + 2^{192} + 2^{256} + 2^{320})$$

provoque le cas d'égalité dans l'addition lors de la dernière itération.

En effet, les deux premières fenêtres *comb* avant le réencodage sont :

$$\begin{cases} K[0] = 1 + 2^{128} + 2^{192} + 2^{256} + 2^{320} \\ K[1] = 2^{64}. \end{cases}$$

La fenêtre $K[1]$ étant paire, elle est remplacée par $K[0] + K[1]$, tandis que $K[0]$ est remplacée par $-K[0]$. La dernière addition de points dans l'algorithme est donc

$$R + \Big[-K[0]\Big]P = [k]P,$$

or on a $k \equiv -2K[0] \bmod q$ donc $R = \Big[-K[0]\Big]P$ et la fonction de doublement est nécessaire.

4
Additions factices de points

Le contenu de ces chapitres est issu des travaux présentés dans [Rus20] et [Rus21a] à la conférence *Symposium sur la sécurité des technologies de l'information et des communications*, éditions 2020 et 2021.

Les additions factices de points sont une réponse à la problématique de la gestion de certains cas exceptionnels présentés dans le chapitre 3. Les conséquences de l'utilisation de cette méthode dans des implémentations présentes dans plusieurs bibliothèques cryptographiques sont abordées dans ce chapitre. Il y a deux contributions :

- D'un côté on présente une attaque par injection de faute ;
- De l'autre une attaque par analyse simple de consommation de courant dans une implémentation spécifique.

Dans les deux cas, les attaques sont combinées avec une attaque par réseau Euclidien afin de reconstituer une clé privée dans le contexte de signature ECDSA.

Sommaire

4.1 Additions factices et attaque *Safe-Error*

En général, une attaque par injection de faute provoque l'algorithme cryptographique à produire une sortie erronée, de laquelle peut ensuite être déduite la clé secrète. Il y a une autre classe d'attaques par injection de faute appelée *Safe-Error* introduite dans [YJ00] qui consiste à regarder si la faute a eu un effet ou non sur la sortie. En effet, une faute est induite sur une opération supposée factice, et cela est confirmé si le résultat reste intact. Par conséquent, les bits secrets liés à cette opération peuvent être devinés.

Cela montre qu'une mesure de protection peut mener à une vulnérabilité exploitable par un autre moyen. Par exemple les contremesures proposées dans [GV10, Ron13] ajoutent des opérations factices pour masquer la différence d'exécution entre l'addition de points et du doublement et l'attaque *Safe-Error* a été appliquée dans [Fou+16] lorsque ces mesures sont appliquées.

Ce type d'attaque fut originellement introduit avec une faute qui touche un registre non-utilisé en fonction du secret, appelée *M Safe-Error* (pour « mémoire »). Le principe a été étendu dans [Yen+01] pour des fautes calculatoires sur l'unité arithmétique et logique du processeur. Cette version est appelée *C Safe-Error* (pour « calculatoire »).

On introduit le principe général de l'attaque *Safe-Error* dans le cadre de signatures ECDSA, puis on montre comment elle peut s'appliquer sur les algorithmes fenêtrés ou sur Montgomery ladder. On termine par deux exemples sous la forme de simulations d'injections de fautes.

4.1.1 Principe général

Premier exemple : *double-and-add-always*. L'algorithme de multiplication scalaire le plus simple est *double-and-add*. Le scalaire est traité des bits de poids fort vers les plus faible, avec un doublement à chaque fois, et une addition seulement si le bit courant vaut 1. Il s'agit essentiellement d'une lecture du scalaire de la gauche vers la droite, avec un doublement de point pour ajouter un 0 et éventuellement une addition pour modifier ce bit en 1. De toute évidence il n'est pas nécessaire de modifier un bit 0 en 0, ce qui pour une courbe elliptique signifie de faire une addition avec le point à l'infini, donc il n'y a pas besoin d'effectuer une opération.

Il a déjà été établi qu'il s'agit d'une situation exceptionnelle pour la plupart des formules d'addition de points. L'algorithme *double-and-add-always* est une réponse pour obtenir une régularité de l'exécution en utilisant une addition factice de points quand le bit courant est 0.

Cependant, par définition, une addition factice n'a pas d'impact sur le calcul de la sortie d'algorithme, donc une faute pendant cette opération révèle si le bit courant est 0 lorsqu'il n'y a pas d'effet observé, ou bien 1 en cas contraire.

Un inconvénient de l'attaque sur *double-and-add-always* est qu'un seul bit est révélé par faute. Bien que cela peut être effectué itérativement en ciblant différents bits pour reconstruire un scalaire fixe, cela n'est pas efficace pour attaquer un schéma de signature comme ECDSA où le scalaire est un nonce aléatoire à chaque exécution.

Dans la suite, on s'intéresse à des algorithmes tels qu'une faute puisse révéler plusieurs bits du scalaire secret en vue d'attaquer ECDSA.

Modèle et étapes de l'attaque. Pour réaliser cette attaque, il est nécessaire de pouvoir produire une faute pendant une instruction parmi un ensemble d'opérations potentiellement factices.

La localisation de la faute est importante, mais n'a pas besoin d'être complètement précise quand les opérations potentiellement factices sont composées de nombreuses instructions (telles que des calculs sur de grands entiers). De plus, la nature exacte de la faute n'est pas pertinente tant qu'une erreur a été produite. Ainsi, toute faute aléatoire transitoire qui produit une erreur calculatoire peut suffire.

Pour cibler ECDSA, l'attaque doit être répétée plusieurs fois quand une même clé privée est utilisée. Finalement, les valeurs publiques doivent être récupérées par l'attaquant (clé publique, signatures et messages signés).

Les étapes principales sont résumées ci-dessous :

1. Produire une faute sur une des instructions supposées factices pendant la génération d'une signature ECDSA ;
2. Récupérer la signature et le message correspondant, vérifier avec la clé publique et conserver en cas de validité ;
3. Répéter les étapes 1 et 2 jusqu'à ce que le nombre de signatures valides atteint un quota minimal (quelques dizaines, variable selon les caractéristiques de l'algorithme et paramètres de la courbes, voir le tableau 2.1) ;
4. Appliquer l'attaque à base de réseau Euclidien pour tenter une reconstruction de la clé privée à partir des signatures valides ;
5. Si la clé privée n'est pas reconstruite, ajouter des signatures valides à partir de l'étape 1.

L'attaque par réseau Euclidien fonctionne bien si le nombre de signatures est suffisant, mais il est probable que cela échoue en cas d'une signature injustement incluse.

4.1.2 Algorithmes fenêtrés et additions factices

Plusieurs bibliothèques cryptographiques introduisent explicitement des additions factices de points pour rendre la multiplication scalaire à temps constant et à comportement régulier. C'est le cas notamment pour certains algorithmes à base de fenêtres qu'on retrouve dans les bibliothèques analysées en section 3.4.

Cible pour l'injection de faute. On rappelle que dans les méthodes fenêtrées, le scalaire k est séparé en plusieurs fenêtres $d_0, d_1, \ldots$, où chaque d_i provient de plusieurs bits du scalaire. Chaque addition se déroule entre un accumulateur et un point précalculé sélectionné dans un tableau selon la valeur de la fenêtre.

L'attaque *Safe-Error* est possible si les conditions suivantes sont remplies :

- Les valeurs d_i sont calculées à partir de plusieurs bits consécutifs du scalaire et peuvent être nulles (après encodage s'il y en a un) ;
- L'addition de points dans la boucle principale est factice quand l'un des points représente le point à l'infini.

La seconde condition est la conséquence de l'exception du point à l'infini dans les formules. Cela se passe lorsqu'une des fenêtre d_i vaut zéro, d'où la première condition.

L'attaque *Safe-Error* consiste à cibler la dernière addition de points dans la phase d'évaluation de l'algorithme. Il y a deux raisons de faire ce choix :

1. Elle est liée soit aux bits de poids fort, soit au bits de poids faible du scalaire, ce qui est important pour l'attaque par réseau Euclidien ;
2. Elle se déroule entre un point qui dépend de toutes les fenêtres précédentes et un point qui dépend uniquement de la dernière, donc une addition factice de points est hautement probable d'être la conséquence de la dernière fenêtre valant zéro.

Enfin, on note en particulier que les instructions à cibler dans la dernière addition doivent être liées au calcul de la coordonnée x de la sortie, vu que seule cette coordonnée est utilisée dans une génération de signature ECDSA.

Implémentations vulnérables. L'implémentation spécifique de la courbe `secp256r1`, dont une partie du code est écrite directement en assembleur, utilise un algorithme fenêtré et est concernée par l'attaque. Elle est présente dans quelques bibliothèques :

- OpenSSL : introduite dans la version 1.0.2 pour l'architecture `x86_64`, et par la suite pour `x86`, `ARMv4`, `ARMv8`, `PPC64` et `SPARCv9`. C'est l'implémentation par défaut pour cette courbe si l'option pour désactiver les optimisations assembleurs n'est pas spécifiée à la compilation ;
- BoringSSL : introduite dans le *commit* `1895493` (novembre 2015), seulement pour l'architecture `x86_64` ;
- LibreSSL : introduite en novembre 2016 dans OpenBSD, mais non présente dans la version portable de la bibliothèque.

On peut ajouter aussi l'algorithme par défaut dans BoringSSL, un algorithme fenêtré simple. Cela concerne les courbes `secp384r1`, `secp521r1`, ainsi que `secp224r1` (selon les options de compilation pour cette dernière).

Il existe d'autres situations dans ces bibliothèques où des additions factices de points sont utilisées, mais l'algorithme sous-jacent est une méthode fenêtrée de type *comb*. Cela nécessiterait plusieurs injections de fautes par exécution pour être exploité comme cela avait été réalisé dans [Fou+16].

4.1.3 Montgomery ladder et additions factices

Contrairement à ce qui est rapporté dans [JY02], l'addition de points dans l'algorithme Montgomery ladder peut devenir factice comme l'a été montré dans [DHB17] et qu'on explique ci-dessous.

On rappelle qu'une étape dans l'algorithme exécute toujours une addition entre les points R_0 et R_1 suivi d'un doublement de R_b lorsque le bit courant est b. Le résultat final est le point R_0 après la dernière étape. Si le dernier bit est 0, alors le résultat de l'addition est placé dans R_1, qui ne fait pas partie de la sortie. Ainsi la dernière addition de points est factice si le bit k_0 du scalaire est nul. Cela est aussi vrai pour les précédentes étapes tant que tous les bits de poids faible qui suivent sont également nuls. La figure **4.1** montre la situation où les trois dernières additions n'interviennent pas pour le calcul de la sortie.

Plus généralement, si $k_j = 1$ est le dernier bit non nul du scalaire, alors le doublement de point dans l'étape j ainsi que les additions dans les étapes suivantes sont des opérations inutiles pour l'obtention du résultat final. Une attaque *Safe-Error* sur Montgomery ladder se réalise ainsi selon la cible :

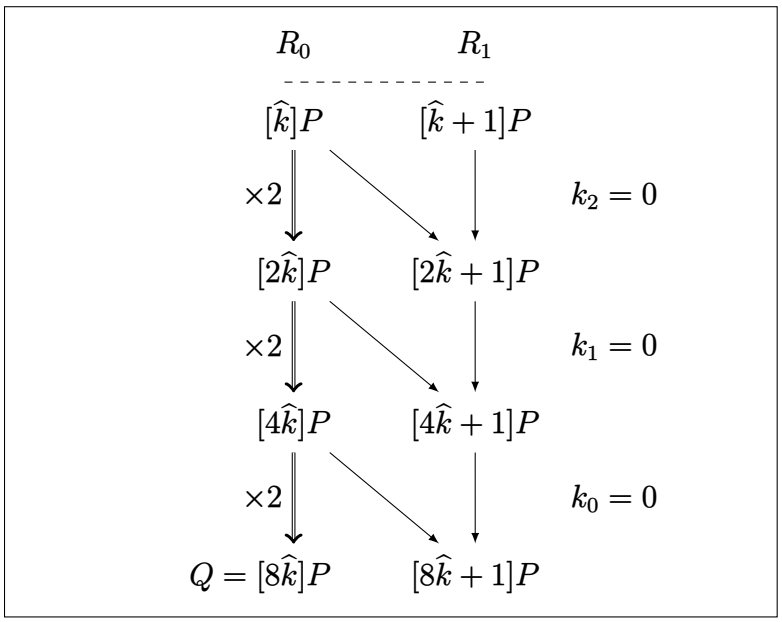


Figure 4.1 : Additions factices implicites dans Montgomery ladder (les quatre dernières valeurs de R_1 n'ont pas d'influence sur le résultat final).

- Si une faute sur le doublement de point pendant le traitement du bit k_i n'a pas d'effet, alors cela révèle que $(k_i, k_{i-1}, \ldots, k_0)_2 = (1, 0, \ldots, 0)_2$;
- Si une faute sur l'addition de point pendant le traitement du bit k_i n'a pas d'effet, alors cela révèle que $(k_i, k_{i-1}, \ldots, k_0)_2 = (0, 0, \ldots, 0)_2$.

Dans le cas où les formules projective `XZ` sont utilisées, le point R_1 est nécessaire dans la reconstruction de la coordonnée affine y du point de sortie, mais celle-ci n'est généralement pas utilisée dans les primitives cryptographiques telles que ECDSA. Une faute pourrait tout de même être détectée par une validation de point.

Pour les formules `XYcoZ`, une perturbation de n'importe quelle formule aura un impact à la fois sur R_0 et R_1, et empêche une reconstruction correcte de la coordonnée projective Z, qui est nécessaire pour obtenir la forme affine. Ces formules peuvent être considérées comme étant immunes face à une attaque *Safe-Error*.

4.1.4 Simulation d'attaque *Safe-Error* sur l'implémentation efficace de la courbe secp256r1

Le concept de l'attaque avec une faute artificiellement introduite dans le code source est donnée à l'adresse suivante :

https://github.com/orangecertcc/ecdummy.

Cependant, on présente ici une simulation mise à jour en intervenant directement dans l'exécution par l'utilisation de GDB, le débogueur du projet GNU. L'implémentation ciblée est la même, à savoir la version efficace de la courbe `secp256r1`. La simulation a été réalisée avec OpenSSL version 1.1.1k compilé sur un Raspberry Pi 4B et est disponible à l'adresse suivante :

https://github.com/orangecertcc/ecdummy-gdb.

Algorithme 10 : Opérations arithmétiques de l'addition de point pour l'implémentation efficace de la courbe `secp256r1` (en évidence : instructions qui peuvent être ciblées pour l'injection de faute dans la dernière addition)

Entrée : $P_1 = (X_1, Y_1, Z_1)$ et $P_2 = (x_2, y_2)$
Sortie : $P_1 + P_2 = (X_3, Y_3, Z_3)$

1: `in1infty` $\leftarrow (Z_1 == 0)$
2: `in2infty` $\leftarrow (x_2, y_2) == (0, 0)$
3: $\mathbf{t_0 \leftarrow Z_1^2}$
4: $\mathbf{t_1 \leftarrow x_2 \cdot t_0}$
5: $\mathbf{t_2 \leftarrow t_1 - X_1}$
6: $\mathbf{t_3 \leftarrow t_0 \cdot z_1}$
7: $\mathbf{z_3 \leftarrow t_2 \cdot z_1}$
8: $\mathbf{t_3 \leftarrow t_3 \cdot y_2}$
9: $\mathbf{t_4 \leftarrow t_3 - Y_1}$
10: $\mathbf{t_5 \leftarrow t_2^2}$
11: $\mathbf{t_6 \leftarrow t_4^2}$
12: $\mathbf{t_7 \leftarrow t_5 \cdot t_2}$
13: $\mathbf{t_1 \leftarrow X_1 \cdot t_5}$
14: $\mathbf{t_5 \leftarrow 2 \cdot t_1}$
15: $\mathbf{X_3 \leftarrow t_6 - t_5}$
16: $\mathbf{X_3 \leftarrow X_3 - t_7}$
17: $t_2 \leftarrow t_1 - X_3$
18: $t_3 \leftarrow y_1 \cdot t_7$
19: $t_2 \leftarrow t_2 \cdot t_4$
20: $y_3 \leftarrow t_2 - t_3$
21: **si** `in1infty` **alors**
22: $\quad (X_3, Y_3, Z_3) \leftarrow (x_2, y_2, 1)$
23: **si** `in2infty` **alors**
24: $\quad (X_3, Y_3, Z_3) \leftarrow (X_1, Y_1, Z_1)$
25: **retourner** (X_3, Y_3, Z_3)

Détails sur l'implémentation. L'algorithme est la méthode fenêtrée avec le réencodage de Booth décrite en section 3.3.2. Il s'agit d'une version légèrement différente lors d'une génération de signature ECDSA, car les fenêtres sont traitées de la droite vers la gauche (des bits de poids faible vers les plus élevés). Une autre caractéristique est l'utilisation d'une table distincte pour chaque position de fenêtre ce qui supprime la nécessité des doublements de points.

La longueur maximale en bits des scalaires est $n = 256$ et la taille de la fenêtre est $w = 7$, donc le nombre de fenêtres est $m = \lceil n/w \rceil = 37$. La dernière est composée des bits k_{251} à k_{255} qui, après encodage, ne peut être nulle que si ces 5 bits sont simultanément nuls. La première condition pour la réalisation de l'attaque est remplie.

Les opérations dans le corps $\mathbf{F}_p$ pour l'addition de points sont données dans l'algorithme 10. Il s'agit d'une addition de points mixte : le premier point est sous coordonnées projectives Jacobiennes (l'accumulateur), tandis que le second est sous représentation affine (le point extrait du tableau de valeurs précalculées). Le point à l'infini est représenté soit par la coordonnée projective Z nulle, soit par le couple $(0, 0)$ lorsqu'il s'agit du point extrait du tableau (point non valide sur la courbe). Comme on peut l'apercevoir, deux valeurs booléennes sont créées au début pour indiquer si l'un des points en entrée est le point à l'infini. Les opérations suivantes sont exécutées quelles que soient ces valeurs. Ensuite, si P_1 (respectivement P_2) est le point à l'infini, alors les coordonnées du point résultant sont remplacées par celles de P_2 (resp. P_1). Par conséquent, les calculs qui précèdent sont ignorés. La seconde condition pour l'attaque est elle aussi remplie, donc l'attaque contre ECDSA peut être réalisée avec cette implémentation.

Remarque 4. Il n'y a bien sûr pas de branchements conditionnels pour des raisons de sécurité dans le code exécuté. Cela est réalisé avec une sélection conditionnelle à temps constant avec des masques binaires créés à partir des valeurs booléennes, tout comme la sélection conditionnelle des points dans les tableaux précalculés.

Réalisation de la faute. Une faute de calcul est simulée en modifiant arbitrairement un registre pendant une multiplication modulaire dans l'addition de points. Premièrement on met un point d'arrêt à la fonction d'addition `ecp_nistz256_point_add_affine` qui est effectif lors du dernier appel seulement afin de cibler la bonne itération. Ensuite, on met un point d'arrêt sur une instruction au cours de l'exécution de la fonction de multiplication modulaire `ecp_nistz256_mul_mont`, suivi d'une modification d'un registre pour simuler une erreur produite par la faute lors de l'exécution de cette fonction. Le choix a été fait d'ignorer les quatre premières exécutions de la multiplication modulaire pour viser l'instruction en ligne 8 de l'algorithme 10.

Les instructions GDB sont présentées dans le listing 4.1 et ce procédé est automatisé par un script afin d'obtenir suffisamment de signatures avec une même clé privée.

```
(gdb) b main
(gdb) r privkey.pem message.txt sig.bin
(gdb) b *0xb6e94540
(gdb) ignore 2 35
(gdb) c
(gdb) b *0xb6e935a0
(gdb) ignore 3 4
(gdb) c
(gdb) set $r3=0x55adab
(gdb) disable 3
(gdb) c
```

Listing 4.1 : Simulation d'attaque *Safe-Error* sur OpenSSL pendant une génération de signature ECDSA (implémentation efficace de `secp256r1`).

Reconstruction de la clé. L'attaque produit une signature valide lors d'une addition factice de points et cela arrive en moyenne toutes les $2^5 = 32$ signatures. D'après le tableau 2.1, 54 signatures sont suffisantes dans la moitié des cas pour reconstruire la clé privée avec un réseau Euclidien, donc on s'attend à devoir effectuer plus de 1700 fautes.

Il a été choisi de simuler la faute sur 2000 signatures. Seules 75 sont valides ce qui est cohérent avec ce qui est attendu. La méthode pour résoudre le problème HNP du chapitre 2 est appliquée avec les signatures valides correspondant au cas où les bits de poids fort du nonce sont connus.

La clé privée a été reconstruite avec les 55 premières signatures valides.

4.1.5 Simulation d'attaque *Safe-Error* sur Montgomery ladder

De même que dans la section précédente, on présente le concept de l'attaque dans le cas de l'algorithme Montgomery ladder avec GDB pour simuler les fautes. L'implémentation visée est celle d'OpenSSL version 1.1.1k compilé sur un Raspberry Pi 4B et la simulation est disponible à la même adresse que précédemment.

La courbe elliptique choisie est `secp256k1` car elle utilise par défaut l'algorithme Montgomery ladder (il serait nécessaire de recompiler OpenSSL en désactivant l'option `no-asm` pour que la courbe `secp256r1` utilise cet algorithme).

Détails de l'implémentation. Il a été identifié dans la section 3.4 que l'algorithme est utilisé avec les formules projectives XZ. Bien que la reconstruction de la coordonnée affine y est effectuée, aucune validation de point n'a lieu, donc une faute qui n'aurait une influence que sur le point R_1 n'aura aucun impact sur la coordonnée affine x de la sortie.

L'addition et le doublement de points sont réunis au sein d'une même fonction, il convient alors de s'assurer quelles sont les opérations susceptibles d'être visées pour réaliser l'attaque *Safe-Error*. Ces opérations sont mises en évidence dans l'algorithme **11**. On note que l'une des opérations au cours de l'addition est également utilisée pour la partie qui concerne le doublement. Une faute sur cette opération aurait un impact sur les deux points en sortie de l'étape et nécessairement sur le résultat final.

Algorithme 11 : Étape de Montgomery ladder avec formules XZ (OpenSSL 1.1.1k)

Entrée : $P_1 = [X_1 : Z_1], \quad P_2 = [X_2 : Z_2], \quad x_0 = x(P_1 - P_2)$
Sortie : $P_1 + P_2 = [X_3 : Z_3], \quad [2]P_1 = [X_4 : Z_4]$

Addition

1: $\mathbf{t_6 \leftarrow X_1 \cdot X_2}$
2: $\mathbf{t_0 \leftarrow Z_1 \cdot Z_2}$
3: $\mathbf{t_4 \leftarrow X_1 \cdot Z_2}$
4: $\mathbf{t_3 \leftarrow Z_1 \cdot X_2}$
5: $\mathbf{t_5 \leftarrow A \cdot t_0}$
6: $\mathbf{t_5 \leftarrow t_6 + t_5}$
7: $\mathbf{t_6 \leftarrow t_3 + t_4}$
8: $\mathbf{t_5 \leftarrow t_6 \cdot t_5}$
9: $\mathbf{t_0 \leftarrow t_0^2}$
10: $t_2 \leftarrow 4 \cdot B$
11: $\mathbf{t_0 \leftarrow t_2 \cdot t_0}$
12: $\mathbf{t_5 \leftarrow 2 \cdot t_5}$
13: $\mathbf{t_3 \leftarrow t_4 - t_3}$
14: $\mathbf{Z_3 \leftarrow t_3^2}$
15: $\mathbf{t_4 \leftarrow Z_3 \cdot x0}$
16: $\mathbf{t_0 \leftarrow t_0 + t_5}$
17: $\mathbf{X_3 \leftarrow t_0 - t_4}$

Doublement

18: $t_4 \leftarrow X_1^2$
19: $t_5 \leftarrow Z_1^2$
20: $t_6 \leftarrow t_5 \cdot A$
21: $t_1 \leftarrow X_1 + Z_1$
22: $t_1 \leftarrow t_1^2$
23: $t_1 \leftarrow t_1 - t_4$
24: $t_1 \leftarrow t_1 - t_5$
25: $t_3 \leftarrow t_4 - t_6$
26: $t_3 \leftarrow t_3^2$
27: $t_0 \leftarrow t_5 \cdot t_1$
28: $t_0 \leftarrow t_2 \cdot t_0$
29: $X_4 \leftarrow t_3 - t_0$
30: $t_3 \leftarrow t_4 + t_6$
31: $t_4 \leftarrow t_5^2$
32: $t_4 \leftarrow t_4 \cdot t_2$
33: $t_1 \leftarrow t1 \cdot t_3$
34: $t_1 \leftarrow 2 \cdot t_1$
35: $Z_4 \leftarrow t_4 + t_1$

Réalisation de la faute. Une faute de calcul est simulée en modifiant arbitrairement un registre pendant la multiplication modulaire lors d'une des dernières étapes de l'algorithme Montgomery ladder. Un point d'arrêt est placé sur la fonction exécutant une étape de l'algorithme, `ec_GFp_simple_ladder_step`, en ignorant les 252 premiers appels pour que la faute soit faite pendant le traitement du bit k_j avec $j = 3$. Ensuite un point d'arrêt est mis sur une instruction de la fonction `bn_mul8x_mont_neon`, suivie d'une modification d'un registre. À nouveau, un choix arbitraire a été fait afin de viser la cinquième multiplication modulaire d'une étape de Montgomery ladder.

Les instructions GDB sont présentées dans le listing **4.2** et ce procédé est automatisé par un script afin d'obtenir suffisamment de signatures avec une même clé privée.

```
1  (gdb) b main
2  (gdb) r privkey.pem message.txt sig.bin
3  (gdb) b *0xb6e9808c
4  (gdb) ignore 2 252
5  (gdb) c
6  (gdb) b *0xb6e0cf48
7  (gdb) ignore 3 4
8  (gdb) c
9  (gdb) set $q9.u64={0x55adab,0xc0ffee}
10 (gdb) disable 2
11 (gdb) disable 3
12 (gdb) c
```

Listing 4.2 : Simulation d'attaque *C Safe-Error* sur OpenSSL pendant une génération de signature ECDSA (Montgomery ladder).

Reconstruction de la clé. Une faute lors de l'addition pendant l'étape j de l'algorithme n'a aucun effet sur le calcul de la signature si les bits k_i du nonce sont nuls pour $0 \leq i \leq j$. Avec $j = 3$, cela arrive pour une signature sur 16 en moyenne et d'après le tableau 2.1 il faut 70 signatures au moins dans la moitié des cas, soit plus de 1100 signatures à attaquer.

La simulation a été réalisée pour la génération de 1400 signatures avec une même clé privée et 78 des signatures sont valides. La méthode de résolution du problème HNP a été appliquée selon la construction donnée dans le chapitre 2 pour le cas où les bits de poids faible du nonce sont connus et en prenant en compte la présence d'un bourrage sur le nonce.

La clé privée a été reconstruite avec les 69 premières signatures valides.

4.2 Additions factices et consommation de courant

Dans [Gou03], il avait été proposé d'utiliser des points spéciaux ayant une coordonnée nulle, de telle sorte que si ce point apparaît au cours des calculs, le poids de Hamming d'une partie des variables pendant une addition de points sera nul. En choisissant bien le point d'entrée de la multiplication scalaire, le point spécial apparait selon certaines hypothèses sur les bits du scalaire secret, ce qui peut être confirmé sur la trace de la consommation de courant. Il s'agit d'un cas particulier d'attaque par simulation de courant nommé *Refined Power Analysis* (RPA).

La même stratégie peut s'appliquer si des additions de points factices sont introduites par des calculs faisant intervenir des valeurs nulles.

On présente les étapes de l'attaque puis on montre pourquoi elle s'applique sur l'implémentation efficace de la courbe `secp256r1`. On illustre le concept de l'attaque par une simulation de consommation de courant.

4.2.1 Modèle et étapes de l'attaque

L'attaque se réalise en deux étapes principales. La première consiste à capturer des traces de consommation de courant de plusieurs générations de signatures ECDSA avec une même clé privée (non connue de l'attaquant). Ainsi il est nécessaire d'avoir un accès physique à la cible. La second étape est l'analyse des traces pour filtrer les signatures pour reconstruire la clé privée par l'utilisation d'un réseau Euclidien. Comme pour l'attaque Safe-Error, il est alors nécessaire de récupérer les signatures, les messages, et bien sûr la clé publique.

On s'intéresse aux traces où le début de la multiplication scalaire pendant la génération d'une signature a une consommation de courant visiblement moins élevée. Cela indiquera qu'une addition factice de points a eu lieu et que les bits du scalaire secret a ses bits de poids faible ou de poids fort à zéro.

Voici un résumé de l'attaque :

1. Capturer la consommation de courant d'une génération de signature ECDSA ;
2. Récupérer la signature et le message correspondant ;
3. Conserver la signature et le message si la consommation de courant apparaît plus faible lors de la première addition de points ;

4. Appliquer l'attaque à base de réseau Euclidien pour tenter une reconstruction de la clé privée à partir des signatures valides ;
5. Retourner en étape 1 pour ajouter des signatures si l'étape précédente a échoué.

4.2.2 Addition factice dans l'implémentation efficace de secp256r1

L'addition de points a été donnée dans l'algorithme 10. On observe qu'il y a deux représentations utilisées :

- Le premier point est l'accumulateur donné par ses coordonnées projectives Jacobiennes X_1, Y_1 et Z_1 ; le point à l'infini est identifié par une valeur Z_1 nulle ;
- Le second point est le point précalculé sous forme affine ; le point à l'infini est identifié par le couple $(0, 0)$.

Lorsque la première fenêtre d_0 est nulle (si les 7 bits de poids faible du scalaire valent zéro), l'accumulateur est initialisé par $(X_1, Y_1, Z_1) = (0, 0, 0)$. En conséquence, presque tous les opérandes dans l'addition de points sont nuls comme montré en évidence dans l'algorithme 12. Chacune de ces opérations sont composées de dizaines d'instructions pour réaliser l'arithmétique de grands entiers avec des mots machines de poids de Hamming nuls.

Algorithme 12 : Addition de points dans l'implémentation efficace de `secp256r1` (en évidence : valeurs nulles lorsque $P_1 = (0, 0, 0)$)

Entrée : $P_1 = (X_1, Y_1, Z_1)$ et $P_2 = (x_2, y_2)$
Sortie : $P_1 + P_2 = (X_3, Y_3, Z_3)$

1: `in1infty` $\leftarrow (Z_1 == 0)$
2: `in2infty` $\leftarrow (x_2, y_2) == (0, 0)$
3: $\mathbf{t_0} \leftarrow \mathbf{Z_1}^2$
4: $\mathbf{t_1} \leftarrow x_2 \cdot \mathbf{t_0}$
5: $\mathbf{t_2} \leftarrow \mathbf{t_1} - \mathbf{X_1}$
6: $\mathbf{t_3} \leftarrow \mathbf{t_0} \cdot \mathbf{Z_1}$
7: $\mathbf{Z_3} \leftarrow \mathbf{t_2} \cdot \mathbf{Z_1}$
8: $\mathbf{t_3} \leftarrow \mathbf{t_3} \cdot y_2$
9: $\mathbf{t_4} \leftarrow \mathbf{t_3} - \mathbf{Y_1}$
10: $\mathbf{t_5} \leftarrow \mathbf{t_2}^2$
11: $\mathbf{t_6} \leftarrow \mathbf{t_4}^2$
12: $\mathbf{t_7} \leftarrow \mathbf{t_5} \cdot \mathbf{t_2}$
13: $\mathbf{t_1} \leftarrow \mathbf{X_1} \cdot \mathbf{t_5}$
14: $\mathbf{t_5} \leftarrow 2 \cdot \mathbf{t_1}$
15: $\mathbf{X_3} \leftarrow \mathbf{t_6} - \mathbf{t_5}$
16: $\mathbf{X_3} \leftarrow \mathbf{X_3} - \mathbf{t_7}$
17: $\mathbf{t_2} \leftarrow \mathbf{t_1} - \mathbf{X_3}$
18: $\mathbf{t_3} \leftarrow \mathbf{Y_1} \cdot \mathbf{t_7}$
19: $\mathbf{t_2} \leftarrow \mathbf{t_2} \cdot \mathbf{t_4}$
20: $\mathbf{Y_3} \leftarrow \mathbf{t_2} - \mathbf{t_3}$
21: **si** `in1infty` **alors**
22: $\quad (X_3, Y_3, Z_3) \leftarrow (x_2, y_2, 1)$
23: **si** `in2infty` **alors**
24: $\quad (X_3, Y_3, Z_3) \leftarrow (X_1, Y_1, Z_1)$
25: **retourner** (X_3, Y_3, Z_3)

Remarque 5. Lorsque la seconde fenêtre d_1 est nulle, cela signifie que $(x_2, y_2) = (0, 0)$ et que l'addition de points est aussi factice. Cependant, seules quatre opérations mettent en jeu un opérande nul (deux multiplications et deux soustractions) sur les 18 opérations arithmétiques. Il est donc attendu que ce cas soit plus difficile à distinguer des autres situations, et surtout ne pas être confondu avec le cas $d_0 = 0$.

Une dernière remarque est que pour que l'accumulateur soit $(0, 0, 0)$ pendant le traitement de la fenêtre d_i, il serait nécessaire que toutes les précédentes fenêtres soient nulles. Cela arrive seulement pour très peu de scalaires : d_0 est nul si les 7 bits de poids

faible sont nuls, mais pour que d_1 le soit aussi, il faudrait que les 7 bits suivants le soient aussi, et ainsi de suite. D'où l'intérêt de se focaliser sur la première addition.

4.2.3 Simulation de consommation de courant

Une simulation a été réalisée sur OpenSSL version 1.1.1k pour valider le modèle. Les outils pour reproduire les résultats de la simulation sont disponibles à l'adresse suivante :

https://github.com/orangecertcc/ecdummyrpa.

Simulation de consommation de courant. La simulation de traces a été effectuée avec `TracerGrind`, un des outils de Side-Channel Marvels.[1] Il s'agit d'un plugin de Valgrind qui permet d'enregistrer les instructions exécutées, ainsi que les valeurs lues et écrites en mémoire d'un programme en cours d'exécution.

Pour l'obtention de la trace, l'outil a été utilisé pour enregistrer les valeurs lues en mémoire pendant la génération d'une signature avec la commande ci-dessous :

```
valgrind --tool=tracergrind --output=memread.trace --trace-instr=no --trace-
    memread=yes --trace-memwrite=no openssl dgst -sha256    -sign privkey.pem -
    out sig.bin msg.txt
```

Listing 4.3 : Ligne de commande pour capturer la trace mémoire avec l'outil `TracerGrind`.

Ensuite, le modèle du poids de Hamming a été appliqué sur ces valeurs. Un exemple de trace est donnée dans la figure 4.2.

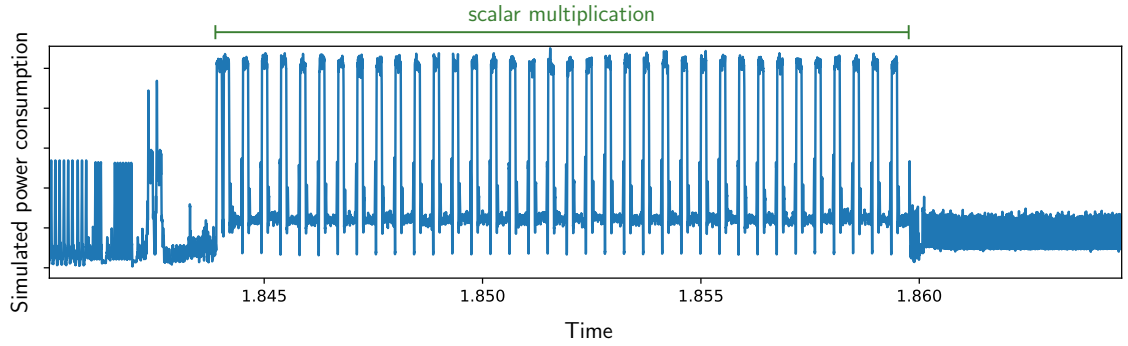


Figure 4.2 : Simulation de consommation de courant pendant la génération d'une signature avec OpenSSL sur l'implémentation efficace de la courbe `secp256r1`.

Analyse. La simulation a été réalisée pour obtenir 6000 traces. Dans cette configuration précise, la partie relative à la première addition est facile à extraire car elle est toujours située dans la même position. Dans la figure 4.3 sont données deux traces représentant les quatre premières additions au cours de la génération de signature.

Comme on peut le remarquer, la première addition de points dans la second trace a un creux considérablement plus bas que les autres, ce qui indique qu'une addition factice de points a eu lieu (plus particulièrement, les 7 bits de poids faible du nonce sont nuls comme il est expliqué ci-dessous).

1. https://github.com/SideChannelMarvels/Tracer.

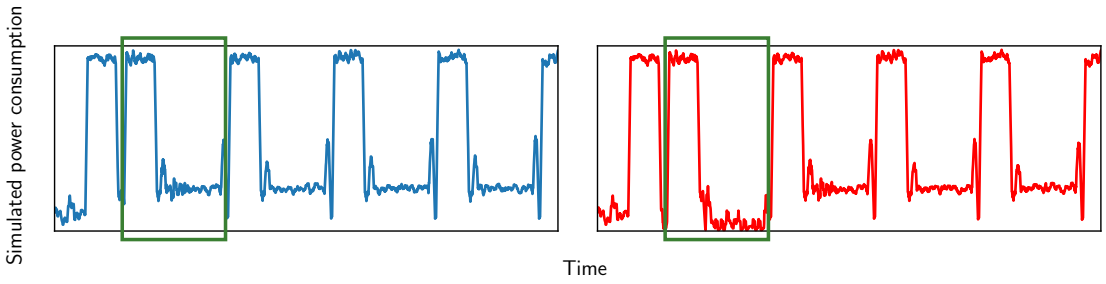


Figure 4.3 : Zoom sur le début de multiplication scalaire de deux générations de signatures avec l'implémentation efficace de la courbe `secp256r1` dans OpenSSL (en évidence : la première addition de points).

La classification des traces en deux catégories peut être réalisée à partir de la moyenne sur la partie concernée. Une autre façon est de laisser un outil faire ce classement lui-même, comme par exemple la bibliothèque Python `Scikit-learn` [SCIKIT11]. Dans cette situation, la classe `GaussianMixture` de l'outil convient très bien. Il parvient à identifier un ensemble de 50 traces parmi les 6000 et ce ratio est cohérent avec la probabilité que les 7 bits de poids faible d'un nonce valent simultanément 0.

Avec cette quantité de bits par nonce connus, on s'attend à ce que la clé privée puisse être reconstruire avec un nombre de signatures compris entre 36 et 40 d'après le tableau 2.1. Dans ce cas précis, la clé a été retrouvée avec les 38 premières signatures.

4.3 À propos des contremesures

Pour se protéger contre ces attaques, il y a plusieurs stratégies. On donne une analyse ci-dessous de plusieurs possibilités qui peuvent être envisagées et éventuellement leurs limites.

4.3.1 Formules complètes

Une première possibilité pour éviter l'introduction explicite d'une addition factice de points est d'utiliser une formule complète. De telles formules existent pour les courbes sous forme de Weierstraß courtes, notamment dans [RCB16]. Un exemple est donné dans l'algorithme 13 (qui correspond à l'algorithme 1 de l'article cité) pour les coordonnées projectives homogènes, compatible lorsque l'une des entrées est le point à l'infini dont les coordonnées sont $(0 : \lambda : 0)$ avec λ non nul.

Cependant, ces formules peuvent être visées par une attaque *Safe-Error*. En effet, on remarque que si P_1 est le point à l'infini, alors à la suite de l'opération en ligne 9, la variable t_4 est nulle. Cette variable est mise à jour avec une multiplication par t_5 en ligne 11, mais reste nulle quelle que soit la valeur de t_5, qui n'est plus utilisée dans la suite.

Ainsi n'importe quelle faute qui perturbe le calcul de t_5 en ligne 10 permet de savoir si oui ou non P_1 est le point à l'infini. Il en est de même pour P_2 si la faute a lieu à la ligne 9 pour perturber le calcul de t_4.

Cela est tout de même une contrainte plus importante par rapport aux situations présentées dans les sections 4.1.4 et 4.1.5 car la cible est beaucoup plus restreinte.

Algorithme 13 : Addition de point complète pour coordonnées projectives et courbes sous forme de Weierstraß courtes d'ordre premier

Entrée : $P_1 = (X_1 : Y_1 : Z_1)$, $P_2 = (X_2 : Y_2 : Z_2)$ et $b_3 = 3 \cdot b$
Sortie : $P_1 + P_2 = (X_3 : Y_3 : Z_3)$

1: $t_0 \leftarrow X_1 \cdot X_2$
2: $t_1 \leftarrow Y_1 \cdot Y_2$
3: $t_2 \leftarrow Z_1 \cdot Z_2$
4: $t_3 \leftarrow X_1 + Y_1$
5: $t_4 \leftarrow X_2 + Y_2$
6: $t_3 \leftarrow t_3 \cdot t_4$
7: $t_4 \leftarrow t_0 + t_1$
8: $t_3 \leftarrow t_3 - t_4$
9: $t_4 \leftarrow X_1 + Z_1$
10: $t_5 \leftarrow X_2 + Z_2$
11: $t_4 \leftarrow t_4 \cdot t_5$
12: $t_5 \leftarrow t_0 + t_2$
13: $t_4 \leftarrow t_4 - t_5$
14: $t_5 \leftarrow Y_1 + Z_1$
15: $X_3 \leftarrow Y_2 + Z_2$
16: $t_5 \leftarrow t_5 \cdot X_3$
17: $X_3 \leftarrow b_3 \cdot t_2$
18: $Z_3 \leftarrow X_3 + Z_3$
19: $X_3 \leftarrow t_1 - Z_3$
20: $Z_3 \leftarrow t_1 + Z_3$
21: $Y_3 \leftarrow X_3 \cdot Z_3$
22: $t_1 \leftarrow t_0 + t_0$
23: $t_1 \leftarrow t_1 + t_0$
24: $t_2 \leftarrow a \cdot t_2$
25: $t_4 \leftarrow b_3 \cdot t_4$
26: $t_1 \leftarrow t_1 + t_2$
27: $t_2 \leftarrow t_0 - t_2$
28: $t_2 \leftarrow a \cdot t_2$
29: $t_4 \leftarrow t_4 + t_2$
30: $t_0 \leftarrow t_1 \cdot t_4$
31: $Y_3 \leftarrow Y_3 + t_0$
32: $t_0 \leftarrow t_5 \cdot t_4$
33: $X_3 \leftarrow t_3 \cdot X_3$
34: $X_3 \leftarrow X_3 - t_0$
35: $t_0 \leftarrow t_3 \cdot t_1$
36: $Z_3 \leftarrow t_5 \cdot Z_3$
37: $Z_3 \leftarrow Z_3 + t_0$
38: **retourner** (X_3, Y_3, Z_3)

4.3.2 Encodage non nul pour algorithmes fenêtrés

Il existe plusieurs méthodes d'encodages pour ne pas avoir de fenêtres nulles dans les algorithmes de multiplication scalaire [Möl01, OT03, HPB04]. Malgré cela, il n'est pas exclu que cela empêche l'apparition de cas exceptionnels dans les formules d'addition de points comme il l'a été démontré pour le réencodage pour les méthodes *comb* en section 3.3.2. Malgré tout, ces cas sont très rares et n'ont pas d'impact cryptographique, donc ces algorithmes sont une solution viable.

4.3.3 Montgomery ladder avec formules spécifiques

Comme il a été montré dans la section 4.1.3, les dernières opérations dans l'algorithme Montgomery ladder peuvent être des calculs inutiles. L'utilisation des formules `XZ` ou `XYcoZ` introduisent la nécessité d'utiliser ces calculs pour obtenir le résultat de la multiplication scalaire. Il convient cependant d'accompagner par une validation de point pour les formules `XZ` pour être certain qu'une faute soit détectée.

5

Scission Euclidienne du scalaire

Le contenu de ce chapitre est issu d'une partie de l'article [Rus21b] présenté à la conférence *Applied Cryptography and Network Security* édition 2021.

Ce chapitre étudie la technique de randomisation basée sur la scission d'un scalaire par une division Euclidienne introduite dans [CJ03]. Une vulnérabilité liée à l'utilisation de cette méthode est présentée où on montre qu'il est possible de réaliser une approximation d'un scalaire à partir d'une fuite légère observable par canaux auxiliaires.

Sommaire

5.1 Introduction

Les attaques mesurant le temps d'exécution ou par analyse de consommation de courant peuvent être évitées par l'utilisation d'algorithmes au comportement régulier par un flot d'opération indépendant du secret. Dans le cas où le scalaire est fixe (tel que dans les protocoles décrits en chapitre 7), les algorithmes de multiplication scalaire ou plus généralement d'exponentiation sont vulnérables à des attaques plus complexes qui nécessitent la capture de plusieurs traces d'exécution [KJJ99].

Des mécanismes basés sur la randomisation furent introduits pour protéger contre ces attaques. Un des plus connus est l'ajout au scalaire d'un multiple de l'ordre du groupe [Cor99], tandis que d'autres consistent à scinder le scalaire en plusieurs morceaux [CJ01, TB02, CJ03]. Cependant, ce type de contremesures ne protège pas nécessairement contre toutes les attaques. Par exemple la technique de randomisation qui ajoute un multiple de l'ordre du groupe ne masque pas entièrement le scalaire pour certaines courbes elliptiques dû à la nature particulière de l'ordre du groupe [FRV14, RIL19]. En ce qui concerne les méthodes de scission du scalaire il a été montré une corrélation entre les morceaux pour la version additive [MV06] et une attaque par profilage a été appliquée sur la version Euclidienne [GRV16].

Ce chapitre se concentre sur la méthode de scission Euclidienne.

L'organisation du chapitre est la suivante. La méthode de scission Euclidienne est présentée dans la section 5.2 avec la variation que celle-ci introduit qui peut mener à une vulnérabilité. Ensuite, la section 5.3 décrit l'utilisation de cette méthode dans la bibliothèque d'Apple pour la multiplication scalaire sur des courbes elliptiques, ainsi que dans l'exponentiation modulaire, avec des éléments de preuves que cette variation est observable par canaux auxiliaires. La section 5.4 analyse la vulnérabilité lorsque la scission Euclidienne est utilisée avec l'algorithme de double multiplication scalaire de Strauss-Shamir. Enfin, les contremesures sont discutées dans la section 5.5.

5.2 Fuite avec la scission Euclidienne

Cette section présente le principe de la randomisation du scalaire par la scission Euclidienne et la variation qui peut entraîner une vulnérabilité.

5.2.1 Description

Soit k un scalaire de n bits dont la représentation binaire est

$$k = \sum_{i=0}^{n-1} k_i 2^i.$$

Cette valeur est randomisée par une valeur aléatoire m de λ bits en effectuant la division Euclidienne de k by m : le quotient est $a = \lfloor k/m \rfloor$ et le reste est $b = k \bmod m$. Ainsi, la multiplication scalaire $[k]P$ est réécrite comme

$$[k]P = [m]([a]P) + [b]P,$$

sous la forme de trois multiplications scalaires.

Cette méthode est la scission Euclidienne et fut introduite dans [CJ03] comme une alternative à d'autres méthodes de masquage. Les auteurs proposent de calculer simultanément la multiplication scalaire avec le quotient a et le reste b par l'algorithme de Strauss-Shamir [Coh+05, algorithme 9.23] pour des raisons d'efficacité.

5.2.2 Variation de la taille du quotient

Une variation de la longueur en bits du quotient et liée au scalaire peut être observée : quand le scalaire est divisé par un entier de longueur fixe, alors la taille du quotient a deux tailles en bits possibles. La probabilité d'apparition pour chacune d'elle dépend de la position du scalaire dans l'intervalle $[2^{n-1}, 2^n)$.

La définition et le théorème ci-dessous donnent les détails sur la partition de l'intervalle et les probabilités associées.

Définition 5. Soit n, λ et β trois entiers positifs tels que $\lambda \leq n$ et $\beta \in [0, 2^{\lambda-1})$. On note

$$I(n, \lambda, \beta) = [\, 2^{n-\lambda}(2^{\lambda-1} + \beta),\ 2^{n-\lambda}(2^{\lambda-1} + \beta + 1)\,),$$

le sous-intervalle de $[2^{n-1}, 2^n)$ de largeur $2^{n-\lambda}$.

Théorème 2. *Soit d la longueur en bits du quotient de la division Euclidienne de l'entier k par un entier m choisi uniformément dans l'intervalle $[2^{\lambda-1}, 2^{\lambda})$. Alors d est égal soit à $n - \lambda$, soit à $n - \lambda + 1$ et on a*

$$\Pr[\, d = n - \lambda + 1 \mid k \in I(n, \lambda, \beta)\,] = \frac{\beta + 1}{2^{\lambda-1}}. \tag{5.1}$$

Démonstration. Soit $k = am + b$ la division Euclidienne de k par un entier m de λ bits. Supposons que $k \in I(n, \lambda, \beta)$. Les bornes inférieures et supérieures sur le quotient a sont

$$\frac{2^{n-1} - b}{2^{\lambda}} \leq a = \frac{k - b}{m} < \frac{2^{n} - b}{2^{\lambda-1}},$$

et vue que $b < 2^{\lambda}$ et a est un entier, on a $a \geq 2^{n-\lambda-1}$. Donc seules les longueurs en bits $n - \lambda$ et $n - \lambda + 1$ sont possibles.

Maintenant, si $m \leq 2^{\lambda-1} + \beta$, alors on a

$$a = \frac{k - b}{m} \geq \frac{2^{n-\lambda}(2^{\lambda-1} + \beta) - b}{m} \geq 2^{n-\lambda} - \frac{b}{m},$$

et sachant que $b < m$ et que a est un entier, on a $a \geq 2^{n-\lambda}$, donc la longueur en bits de a est $n - \lambda + 1$. Dans ce cas il y a $\beta + 1$ valeurs possibles pour m sur les $2^{\lambda-1}$, d'où la probabilité.

L'autre cas est lorsque $m > 2^{\lambda-1} + \beta$, alors on a

$$a < \frac{2^{n-\lambda}(2^{\lambda-1} + \beta + 1) - b}{m} \leq 2^{n-\lambda},$$

et la longueur en bit de a est $n - \lambda$. □

La conséquence est que si m est généré aléatoirement de façon uniforme et de longueur en bits λ, alors la probabilité que le quotient fasse $n - \lambda + 1$ bits dépend de quel intervalle $I(n, \lambda, \beta)$ contient le scalaire k. C'est illustré dans la figure 5.1 avec $\lambda = 4$ pour mettre en évidence la fonction en escalier, et $\lambda = 32$ (le cas de la bibliothèque d'Apple présenté dans les sections suivantes). Dans ce second cas, les intervalles sont petits et cela est proche d'une corrélation linéaire entre scalaire et probabilité.

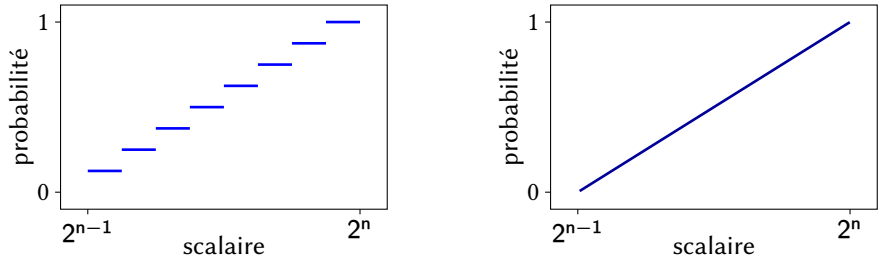


Figure 5.1 : Corrélation entre un scalaire de n bits et la probabilité que la longueur en bits du quotient de la division Euclidienne avec un diviseur de λ bits est $n - \lambda + 1$ (à gauche : $\lambda = 4$, à droite : $\lambda = 32$).

5.2.3 Approximation du scalaire

Un oracle qui révèle $d = \lceil \log_2(k/m) \rceil$ pour un entier fixe k et N entiers aléatoires m de longueur en bits identique λ suit une loi binomiale de paramètres N et de probabilité $(\beta + 1)/2^{\lambda-1}$ selon le théorème 2. Cet oracle permet de construire un intervalle de confiance sur la probabilité inconnue quand seuls sont connus le nombre de tests N et le nombre de succès n_{obs} que d prenne la valeur $n - \lambda + 1$. Comme cette probabilité est une caractéristique de l'intervalle $I(n, \lambda, \beta)$, alors une approximation de k peut en être déduite.

Les étapes sont directes. Un intervalle de confiance $[p_1, p_2]$ de la probabilité est obtenu par l'observation du nombre n_{obs} de succès (la longueur en bits observée du quotient est $d = n - \lambda + 1$) sur les N issues. Ensuite β peut être également estimé par

$$p_1 2^{\lambda-1} - 1 \leq \beta \leq p_2 2^{\lambda-1} - 1,$$

et comme il s'agit d'un entier, on note $\beta_{\min} = \lceil p_1 2^{\lambda-1} - 1 \rceil$ et $\beta_{\max} = \lfloor p_2 2^{\lambda-1} - 1 \rfloor$. Finalement, l'approximation de k est réalisée par la concaténation des intervalles contigus $I(n, \lambda, \beta_{\min})$ jusqu'à $I(n, \lambda, \beta_{\max})$:

$$\Big[2^{n-\lambda}(2^{\lambda-1} + \beta_{\min}),\ 2^{n-\lambda}(2^{\lambda-1} + \beta_{\max} + 1) \Big). \tag{5.2}$$

Cet intervalle contient probablement le scalaire k en fonction du niveau de confiance sur l'approximation de la probabilité.

Intervalle de Wilson. L'intervalle de confiance de la probabilité peut être obtenu avec l'intervalle de Wilson [Wil27]. Il a l'avantage d'être adapté pour des échantillons de petites tailles ou lorsque la probabilité est proche de 0 ou 1. Il est donné par la formule suivante où c est le niveau de confiance (1,96 pour 95 % par exemple) :

$$p_1, p_2 = \frac{n_{\text{obs}} S + c^2/2}{N + c^2} \pm \frac{c}{N + c^2} \sqrt{\frac{n_{\text{obs}}(N - n_{\text{obs}})}{N} + \frac{c^2}{4}}.$$

On voit par cette formule l'impact de la taille de l'échantillon N sur la largeur de l'intervalle. Dans le pire cas la largeur est $c/\sqrt{N + c^2}$ (lorsque la probabilité est proche de 0,5) donc il faudrait multiplier la taille de l'échantillon par $2M$ pour diviser par M la largeur.

Une seconde observation est que l'approximation est meilleure pour les probabilités proches de 0 ou 1. En conséquence les exposants ou scalaires situés au bord de l'intervalle $[2^{n-1}, 2^n]$ sont bien plus facile à approximer.

5.3 Scission Euclidienne dans CoreCrypto

Le code source de la bibliothèque cryptographique d'Apple nommée CoreCrypto est mis à disposition pour permettre la vérification des caractéristiques de sécurité et son bon fonctionnement.[1] Cette bibliothèque implémente les fonctions de bas niveau des primitives cryptographiques, accessibles par une interface de plus haut niveau pour les développeurs.

C'est dans cette dernière que l'on retrouve les détails des implémentations de courbes elliptiques. La méthode de scission pour randomiser le scalaire secret est présente dans la multiplication scalaire pour les courbes elliptiques `secp192r1`, `secp224r1`, `secp256r1`, `secp384r1` et `secp521r1`. Cette méthode est également utilisée pour l'exponentiation modulaire.

Dans cette section, on présente les spécificités pour chaque avec des éléments de preuves que la variation puisse être observée.

Les travaux présentés ont été communiqués à Apple Product Security en respectant leur procédure. Une mise à jour avec des correctifs a été appliquée à partir des versions iOS 14.5 et macOS 11.3.[2]

La version analysée du code est celle qui précède l'application de ces correctifs.

5.3.1 Multiplication scalaire dans CoreCrypto

La multiplication scalaire est décrite dans l'algorithme **14**. La méthode de bourrage est appliquée sur le scalaire en amont de la scission, donc il s'agit toujours de scalaires de n bits : 193, 225, 257, 385 et 522 respectivement pour les cinq courbes. La scission est effectuée avec un diviseur de longueur $\lambda = 32$ bits.

La multiplication scalaire est appliquée avec l'algorithme Montgomery ladder individuellement pour le quotient, diviseur et le reste. Cet algorithme est décrit dans la section **3.2.1** et son exécution est variable selon la longueur en bits du scalaire en entrée. En effet, le nombre d'itérations de la boucle principale de l'algorithme dépend directement de la longueur en bits du scalaire nommée `dbitlen` dans le listing **5.1** qui contient l'extrait du code source.

Cette valeur est donnée en entrée de la fonction `ccec_mult_ws` appelée à trois reprises. Pour la multiplication scalaire avec le diviseur, cette valeur est fixée par la constante `SCA_MASK_BITSIZE` qui vaut 32 car le diviseur est toujours un entier de 32 bits. Pour celle avec le reste la valeur est 33. Ceci est dû au fait que la multiplication scalaire effectuée est $[2^{32} + b]P$ et non $[b]P$ (la valeur 2^{32} est retranchée au scalaire avant la scission pour que le résultat soit correct). Cependant, la valeur `dbitlen` pour le quotient a besoin d'être calculée car elle ne peut être connue à l'avance. Ces informations se retrouvent aux lignes 3, 9 et 14 du listing **5.2**.

Ainsi, on note que seule la multiplication scalaire avec le quotient est variable dans l'exécution de la multiplication scalaire générale.

Mesure du temps d'exécution. Avec un programme de test compilé avec la bibliothèque CoreCrypto, une mesure du temps d'exécution a été réalisée pour plusieurs

1. https://developer.apple.com/security/ (bas de page).
2. https://support.apple.com/en-us/HT212325 (pour macOS 11.3).

Algorithme 14 : Multiplication scalaire et scission Euclidienne dans CoreCrypto

Entrée : P, k avec $1 < k < q - 1$

Sortie : $[k]P$

Bourrage et scission du scalaire

$k_1 \leftarrow k + q - 2^{32}$

$k_2 \leftarrow k + 2q - 2^{32}$

si $\lceil \log_2(k_1) \rceil = \lceil \log_2(q) \rceil + 1$ **alors**

    $k_{\text{pad}} \leftarrow k_1$

sinon

    $k_{\text{pad}} \leftarrow k_2$

$m \leftarrow$ entier aléatoire dans $[2^{31}, 2^{32} - 1]$

$a \leftarrow \lfloor k_{\text{pad}}/m \rfloor, \quad b \leftarrow k_{\text{pad}} \bmod m$

Multiplications scalaires individuelles

$Q \leftarrow [a]P$

$R \leftarrow [m]Q$

$S \leftarrow [b + 2^{32}]P$

retourner $R + S$

```
// Main loop
// Assumes that MSB is set: d_dbitlen-1 is == 1
// This algo does not read it to verify it is indeed one.
for (size_t i = dbitlen - 2; i > 0; --i) {
    dbit ^= ccn_bit(d, i);
    // Use buffer copy instead of pointer handling to prevent cache attacks
    cond_swap_points(n, dbit, R0, R1);
    XYCZaddC_ws(ws, cp, R0, R1);
    XYCZadd_ws(ws, cp, R0, R1);
    // Per Montgomery Ladder:
    // Invariably, R1 - R0 = P at this point of the loop
    dbit = ccn_bit(d, i);
}
```

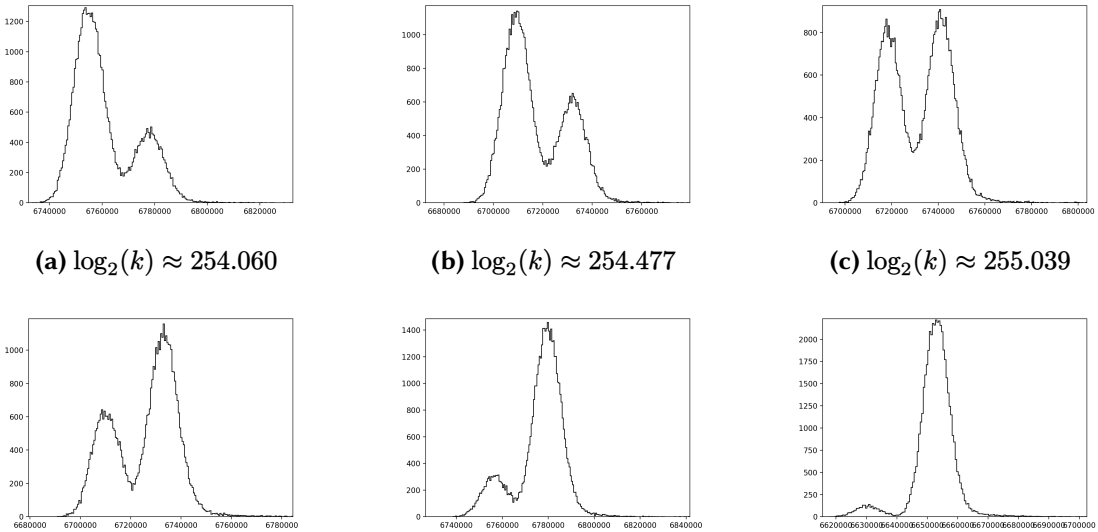
Listing 5.1 : Boucle principale de Montgomery ladder dans la multiplication scalaire de CoreCrypto (fichier `ccec/src/ccec_mult.c`).

```
// a.S
cc_require_action(ccn_is_zero(n, ccec_point_x(S, cp)) == 0, errOut, status =
    CCERR_PARAMETER);
dbitlen = ccn_bitlen(n + 1, dtmp1);
status = ccec_mult_ws(ws, cp, Q, dtmp1, dbitlen, S);
cc_require(status == 0, errOut);

// mask.a.S
cc_require_action(ccn_is_zero(n, ccec_point_x(Q, cp)) == 0, errOut, status =
    CCERR_PARAMETER);
dbitlen = SCA_MASK_BITSIZE;
status = ccec_mult_ws(ws, cp, R, &mask, dbitlen, Q);
cc_require(status == 0, errOut);

// b.S
dbitlen = SCA_MASK_BITSIZE + 1; // equivalent to b+(2*SCA_MASK_MSBIT)
status = ccec_mult_ws(ws, cp, Q, &b, dbitlen, S);
cc_require(status == 0, errOut);
```

Listing 5.2 : Multiplication scalaires individuelles avec le quotient, diviseur et reste.

scalaires. Cela a été répétée 50000 fois pour chacun d'entre eux et les résultats sont donnés dans la figure 5.2, où les scalaires sont rangés dans l'ordre croissant.

Deux temps d'exécutions sont observables avec une différence de proportion entre les pics, avec un second pic plus élevé pour les scalaires les plus grands. Cela est cohérent avec la description donnée ci-dessus.



(a) $\log_2(k) \approx 254.060$ **(b)** $\log_2(k) \approx 254.477$ **(c)** $\log_2(k) \approx 255.039$

(d) $\log_2(k) \approx 255.353$ **(e)** $\log_2(k) \approx 255.720$ **(f)** $\log_2(k) \approx 255.924$

Figure 5.2 : Temps d'exécution pour 50000 multiplications scalaires sur la courbe `secp256r1` avec un scalaire fixe sur CoreCrypto.

5.3.2 Exponentiation modulaire dans CoreCrypto

Il y a plusieurs algorithmes pour l'exponentiation modulaire dans cette bibliothèque. L'exponentiation qui utilise la scission Euclidienne est implémentée dans la fonction `ccdh_power_blinded`.[3] Comme avec les courbes elliptiques, les exponentiations avec le quotient, diviseur et le reste sont réalisées individuellement, avec d'autres techniques de randomisation. L'ensemble de l'exponentiation est résumé dans l'algorithme 15, où le paramètre λ est 32 pour la taille du diviseur aléatoire.

L'algorithme d'exponentiation utilisé aux lignes 5 à 7 est une méthode fenêtrée avec des fenêtres de 2 bits, donné dans l'algorithme 16 et nommé *square-square-multiply-always* (SSMA). Le nombre d'itérations dans cet algorithme est dépendant de la longueur en bits de l'exposant : si cette longueur est d, il y a alors $\lceil d/2 \rceil$ itérations de la boucle. Cela peut être révélé par une attaque par canaux cachés en comptant le nombre de motifs sur une trace de consommation de courant ou d'émanations électromagnétiques (voir ci-dessous). La longueur en bits du diviseur est fixe et connue à l'avance, mais celles du quotient et du reste sont variables.

Pour un exposant de taille fixe n, on a vu qu'il y a deux longueurs d en bits possibles pour le quotient, ce qui peut être utilisé pour effectuer une approximation. Les deux

3. Dans le fichier `ccdh/src/ccdh_power_blinded.c`.

Algorithme 15 : Exponentiation modulaire et scission Euclidienne dans CoreCrypto

Entrée : x, g, p
Sortie : $g^x \bmod p$

Masquage de la base, du module, et de l'exposant
1: $p^\star \leftarrow \textbf{blinding}(p)$ ▷ $p^\star = p \cdot \text{random}$
2: $g^\star \leftarrow \textbf{blinding}(g)$ ▷ $g^\star = g + p \cdot \text{random}$
3: $m \leftarrow$ entier aléatoire dans $[2^{31}, 2^{32})$
4: $a \leftarrow \lfloor x/m \rfloor, b \leftarrow x \bmod m$

Exponentiation
5: $t_1 \leftarrow g^{\star a} \bmod p^\star$
6: $t_2 \leftarrow {t_1}^m \bmod p^\star$
7: $t_3 \leftarrow g^{\star b} \bmod p^\star$
8: **retourner** $(t_2 \cdot t_3) \bmod p$ ▷ $(g + p \cdot \text{random})^{am+b} \bmod p = g^x \bmod p$

Algorithme 16 : Algorithme *square-square-multiply always*

Entrée : $g, a = (a_{d-1}, \ldots, a_0)_2$ with $a_{d-1} = 1$
Sortie : g^a

1: **pour** $i = 0$ **à** 3 **faire**
2: $\quad \text{tab}[i] \leftarrow g^i$
3: $r \leftarrow 1$
4: **pour** $i = \lceil d/2 \rceil - 1$ **à** 0 **faire**
5: $\quad r \leftarrow r^2$
6: $\quad r \leftarrow r^2$
7: $\quad r \leftarrow r \cdot \text{tab}[2a_{2i+1} + a_{2i}]$
8: **retourner** r

peuvent se produire pour différentes valeurs de m, mais il n'est pas toujours possible de distinguer les deux cas avec l'algorithme SSMA. En effet, si $n - \lambda$ est impair alors

$$\left\lceil \frac{n-\lambda}{2} \right\rceil = \left\lceil \frac{n-\lambda+1}{2} \right\rceil = \frac{n-\lambda+1}{2},$$

donc la longueur en bits du quotient ne fuite pas, mais n peut malgré tout être déduit de la formule. Au contraire, si $n - \lambda$ est pair, alors

$$\left\lceil \frac{n-\lambda+1}{2} \right\rceil = \left\lceil \frac{n-\lambda}{2} \right\rceil + 1, \tag{5.3}$$

donc l'algorithme est exécuté avec un nombre différent d'itérations pour chacune des valeurs possibles pour la taille du quotient en bits (et la longueur en bits n de l'exposant peut être déduit aussi). Malgré cela, il est nécessaire d'effectuer plusieurs observations de l'exponentiation pour être certain d'être dans l'un ou l'autre cas. Par conséquent, une seule mesure n'est pas suffisante pour connaître exactement la valeur n.

Capture de consommation de courant. Cette partie a été réalisée en collaboration avec Cyril Delétré, qui a réalisé la configuration et obtenu les captures.

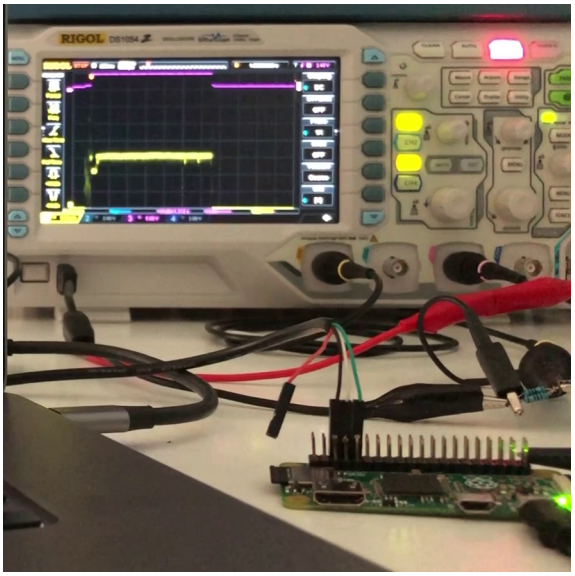


Figure 5.3 : Mesure de la consommation de courant sur un Raspberry Pi Zero.

La variation liée au quotient peut être révélée en comptant le nombre de motifs sur une trace de consommation de courant. Le Raspberry Pi Zero a été sélectionné car il offre un bon compromis entre les performances de calculs, la complexité matérielle, et la configuration de l'alimentation électrique. Premièrement, pour obtenir une capacité constante de calcul et minimiser le bruit sur la ligne électrique, la fréquence processeur a été figée à 700 MHz et la sortie HDMI/TV désactivée (ce qui a permis d'avoir des traces plus nettes). En partant d'une nouvelle installation du *Raspberry Pi OS* (auparavant nommé Raspbian car basé sur la distribution GNU/Linux Debian), la bibliothèque CoreCrypto a été compilée à partir des sources. Ensuite, un programme écrit en C bascule la ligne GPIO du Raspberry Pi Zero avant et après l'exécution de la fonction `ccdh_power_blinded`.

Ainsi, l'oscilloscope peut mesurer la consommation de courant avec une première sonde connectée en série à une résistance (3,3 Ohms dans notre cas) sur la ligne à 5 V. Ensuite il peut déclencher le début du calcul avec une seconde sonde connectée à la ligne GPIO.

Finalement, le processus a été automatisé avec un script Python pour lancer le calcul sur le Raspberry Pi Zero avec la console série UART (moins de perturbations comparé à une connexion USB ou une console SSH/telnet), et télécharger les données de l'oscilloscope après chaque exécution par une console TCP à distance.

Le groupe de 2048 bits donné dans RFC 5054 [Tay+07] a été choisi pour cette expérimentation, avec l'exposant

$$x = \texttt{d3afa905fededc64bc907b809da3dcb}\\ \texttt{484763c25c3b4728bb081a97cf9f0a5} \tag{5.4}$$

donné ici sous forme hexadécimale. Pour chaque exécution, le générateur pseudo-aléatoire de CoreCrypto utilisé dans la fonction `ccdh_power_blinded` a été initialisé avec une graine aléatoire (pour les techniques de randomisation). L'ensemble a été réalisé 10000 fois.

Deux échantillons de traces sont donnés dans la figure **5.5** où l'on peut voir que la majeure partie de la trace correspond à l'exponentiation avec le quotient (des lignes

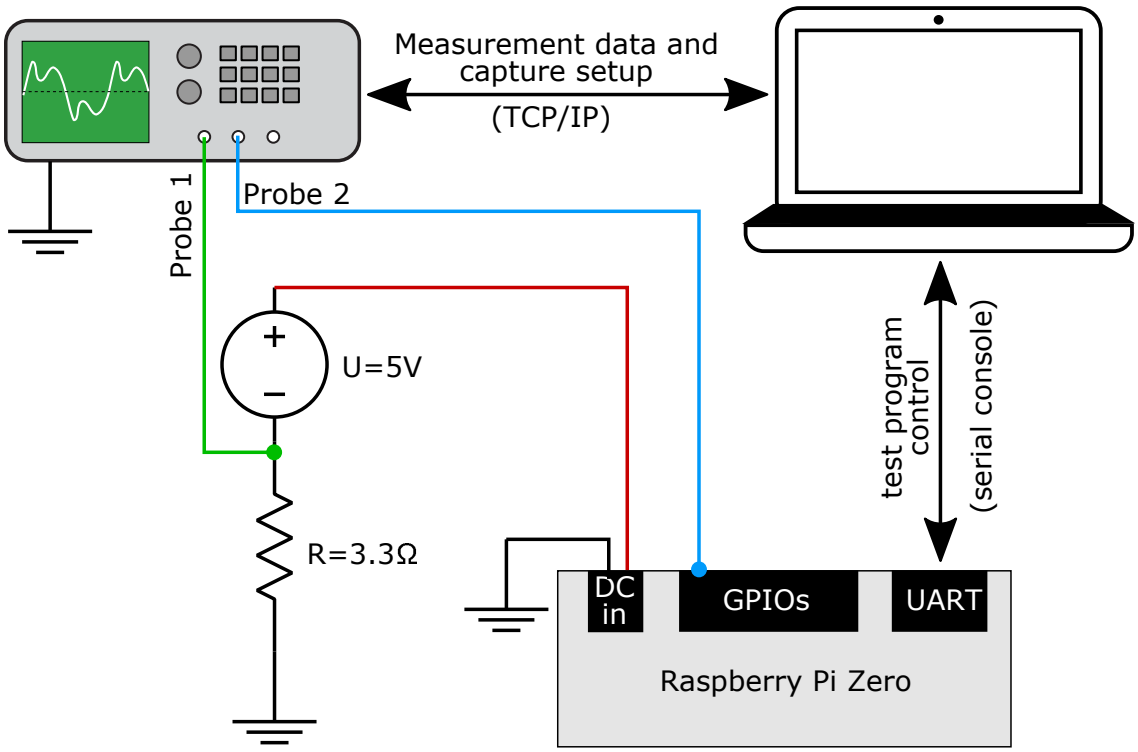


Figure 5.4 : Configuration de l'expérimentation avec l'oscilloscope et le Raspberry Pi Zero.

verticales indiquent le début des exponentiations individuelles). Dans la figure 5.6, un zoom sur la fin de celle-ci révèle qu'elle est plus courte dans la première trace d'un motif *square-square-multiply* par rapport à la seconde trace.

Il est intéressant de noter qu'il n'est pas nécessaire de compter les motifs *square-square-multiply* chaque fois, vue que l'approximation n'a besoin que de distinguer entre les deux cas. Cependant, il reste important de le faire au moins une fois pour trouver la longueur en bits n de l'exposant. Dans cet exemple, le nombre d'itérations pour l'exponentiation avec le quotient sont respectivement 108 et 109. D'après l'équation (5.3), on en déduit qu'il s'agit d'un exposant de 248 bits, ce qui est cohérent avec la valeur donnée dans l'équation (5.4).

On peut remarquer que cela peut être trouvé même si les motifs sont difficiles à distinguer si le bruit est trop élevé, tant que le début des exponentiations individuelles sont observables. En effet, comme le diviseur aléatoire est toujours un entier de 32 bits, alors il y a toujours 16 itérations de la boucle. Par conséquent, le nombre d'itérations avec le quotient peut être déduit de la longueur entre les deux premières lignes verticales (en prenant en compte le carré et la multiplication du précalcul).

Le début des exponentiations est facilement trouvable dans les captures, grâce au creux correspondant à la première itération de la boucle de l'algorithme SSMA. Cela est dû à l'initialisation de l'algorithme avec la valeur $p - 1$, donc le second carré est 1^2, suivi d'une multiplication par 1. Ces deux opérations manipulent des valeurs de poids de Hamming très faibles, ce qui a un impact significatif sur la consommation.

Cette caractéristique a été utilisée pour classifier les traces dans les deux groupes possibles, à partir des positions des creux pour l'exponentiation avec le diviseur aléatoire. Quelques centaines de traces se sont révélées inutilisables à cause de perturbations trop

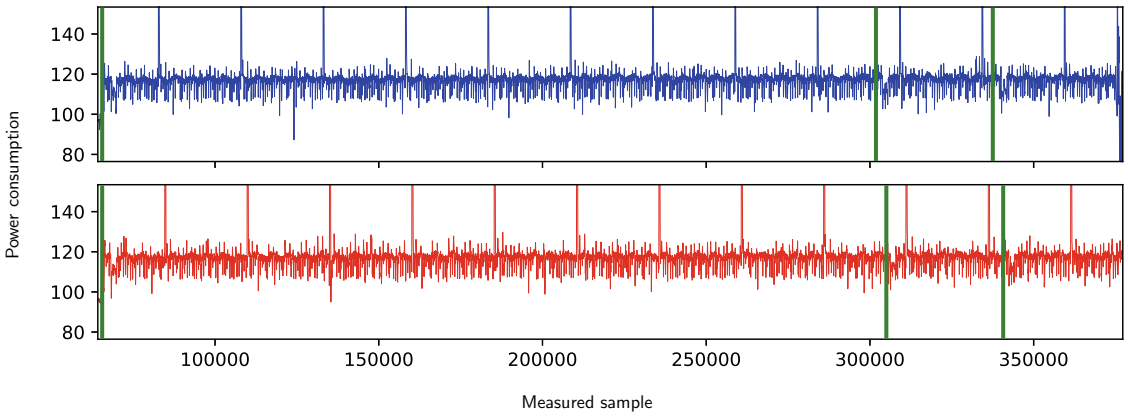


Figure 5.5 : Traces de deux exponentiations masqués avec la scission Euclidienne avec un exposant identique (les lignes verticales indiquent le début des exponentiations avec le quotient, le diviseur et le reste).

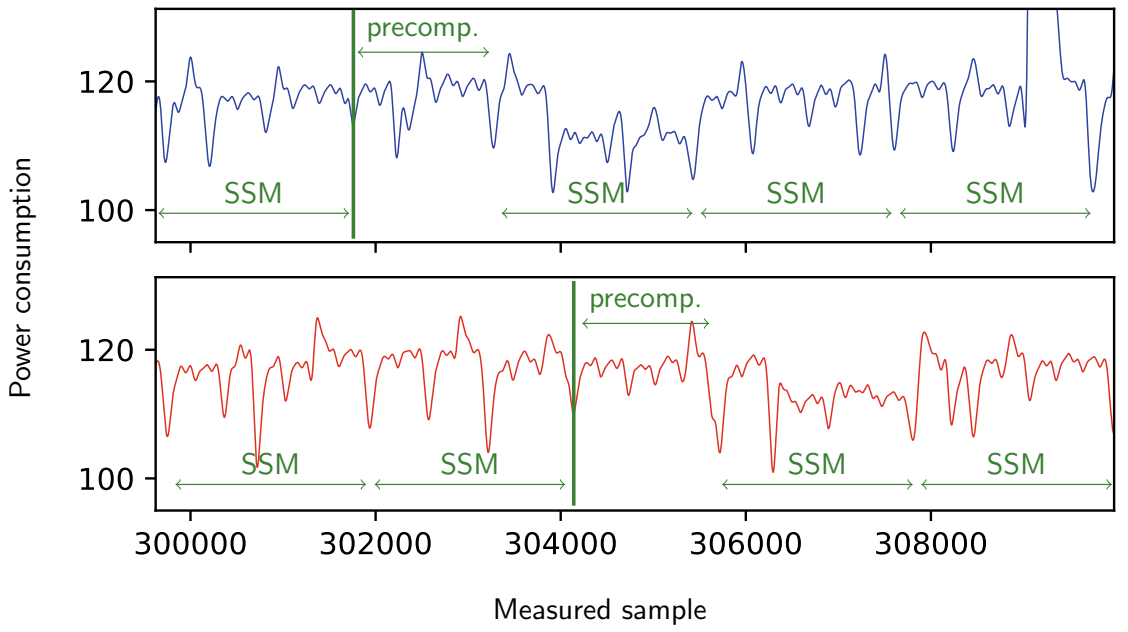


Figure 5.6 : Zoom sur la fin de l'exponentiation avec le quotient, et le début de celle avec le diviseur aléatoire.

importantes, mais cette méthode s'est révélée suffisamment efficace : 6099 traces sur 9356 ont été identifiés comme appartenant au groupe « 109 ». Ainsi, la fréquence observée est environ 0,6519 ce qui est proche de la probabilité cherchée qui vaut environ 0,6538.

En utilisant l'intervalle de Wilson pour l'approximation avec un niveau de confiance de 95 %, on détermine que l'exposant x est situé dans un intervalle de largeur environ $2^{241,3049}$ et qui contient bien l'exposant donné dans l'équation (5.4).

5.4 Scission Euclidienne avec l'algorithme de Strauss-Shamir

La fuite décrite dans la section 5.2 est applicable avec l'algorithme de Strauss-Shamir qui fut proposé pour être utilisé avec la scission Euclidienne dans l'article d'origine [CJ03].

5.4.1 Description de l'algorithme

Il fut proposé de calculer d'abord $S = [m]P$, puis utiliser une variante de l'algorithme de multi-exponentiation de Strauss-Shamir pour calculer simultanément $[a]S + [b]P$ au prix d'une seule multiplication scalaire. Il est présenté dans l'algorithme 17, où ℓ est la valeur maximale entre les longueurs en bits de a et b. On peut noter la présence d'une addition factice de points dans le cas où les deux bits courants des scalaires a et b sont nuls, pour qu'un doublement et une addition de points soit exécutés à chaque itération.

Algorithme 17 : Double multiplication régulière de Strauss-Shamir

Entrée : $a = (a_{\ell-1}, \ldots, a_0)$, S, $b = (b_{\ell-1}, \ldots, b_0)_2$, P, $(a_{\ell-1}, b_{\ell-1}) \neq (0, 0)$
Sortie : $Q = [a]S + [b]P$

1: $R_1 \leftarrow P$; $R_2 \leftarrow S$; $R_3 \leftarrow P + S$; $r_4 \leftarrow P$
2: $c \leftarrow 2a_{n-1} + b_{n-1}$; $R_0 \leftarrow R_c$
3: **pour** $i = \ell - 2$ **à** 0 **faire**
4: $\quad b \leftarrow \neg(a_i \vee b_i)$; $c \leftarrow 2a_i + b_i$
5: $\quad R_0 \leftarrow [2]R_0$
6: $\quad R_{4b} \leftarrow R_{4b} + R_c$
7: **retourner** R_0

5.4.2 Vulnérabilité

Pour exploiter la fuite et réaliser l'approximation, il est nécessaire de connaître les longueurs en bits de k, de a et et de m. On peut supposer que la longueur λ de m est connue (elle peut fuiter de la multiplication scalaire $[m]P$). Cependant, dans la double multiplication scalaire la longueur $d := \lceil \log_2(a) \rceil$ du quotient a peut être masquée par $e := \lceil \log_2(b) \rceil$, celle du reste b. En effet, le nombre d'itérations de la boucle dépend de $\ell = \max(d, e)$, et e peut être plus grand que d.

Malgré cela, il peut être possible dans certains cas d'observer la variation sur la longueur en bits du quotient. Si un bourrage est appliqué sur le scalaire k, alors sa longueur n devient fixe et connue et il ne reste qu'à connaître d. Dans le cas où λ est

plus petit que la moitié de n, alors e toujours inférieur à d, car $d - e \geq \ell - 2\lambda \geq 0$. Par conséquent, $\max(d, e)$ est toujours égal à d, qui est révélé par le nombre d'itérations de la boucle principale.

Si aucun bourrage n'est appliqué sur le scalaire, alors la longueur n de k est inconnue, mais il peut être possible de la retrouver avec quelques mesures. On sait que $\max(d, e)$ vaut soit $n - \lambda$, $n - \lambda + 1$ ou e. Avec quelques mesures, les valeurs $n - \lambda$ and $n - \lambda + 1$ peuvent être identifiées et n déduit. Par exemple, si $\lambda = 128$ et $n = 256$, alors d est égal à 128 ou à 129 d'après le théorème 2, et vue que $e \leq \lambda$, alors $\max(d, e)$ ne peut que valoir 128 ou 129, ce qui révèle d. Mais si $n = 255$, alors les seules valeurs possibles sont 127 et 128, et si $n = 254$ il s'agit de 126, 127, et 128. Et ainsi de suite pour des valeurs plus petites de n. Ainsi, après plusieurs exécutions, si la longueur 129 apparaît pour d, alors on est assuré que $n = 256$, et l'approximation peut être réalisée. Mais pour des valeurs plus petites pour n il peut devenir impossible de distinguer quand les deux valeurs possibles pour d apparaissent, qui est l'élément central pour réaliser une approximation. Toutefois, des valeurs plus petites pour n sont moins probables lorsque k est généré uniformément, donc l'attaque peut être réalisée dans une majorité de cas.

5.5 Contremesures

Dans cette section, des techniques sont proposées pour éviter la fuite avec la scission Euclidienne, ainsi que les mesures proposées par Apple dans le correctif déployé.

5.5.1 Protection pour Strauss-Shamir

Pour cacher la longueur en bits du quotient, on peut emprunter la technique du bourrage sur le reste dans la multiplication scalaire de CoreCrypto. En choisissant un diviseur aléatoire de la bonne taille, le nombre d'itérations de la boucle dans l'algorithme devient fixe et indépendant du scalaire secret. C'est ce qui est proposé dans l'algorithme 18.

Algorithme 18 : Scission Euclidienne protégée avec la double multiplication scalaire de Strauss-Shamir

Entrée : k, P, q
Sortie : $[k]P$

$\lambda \leftarrow \lceil (t+1)/2 \rceil$
$k_1 \leftarrow k + q - 2^\lambda$
$k_2 \leftarrow k + 2q - 2^\lambda$
si $\lceil \log_2(k_1) \rceil = \lceil \log_2(q) \rceil + 1$ **alors**
    $k_{\text{pad}} \leftarrow k_1$
sinon
    $k_{\text{pad}} \leftarrow k_2$
$m \leftarrow$ entier aléatoire dans $[2^{\lambda-1}, 2^\lambda - 1]$
$a \leftarrow \lfloor k_{\text{pad}}/m \rfloor$, $b \leftarrow k_{\text{pad}} \bmod m$
$S \leftarrow [m]P$
$Q \leftarrow [a]S + [b + 2^\lambda]P$ ▷ Algorithme Strauss-Shamir
retourner Q

La première multiplication scalaire a une entrée de longueur fixe. Le second scalaire dans la double multiplication scalaire a une longueur qui vaut $\lambda+1$. D'après le théorème 2, la longueur en bits du quotient vaut au plus $t+2-\lambda$. Le choix du paramètre λ permet de s'assurer de l'inégalité $\lambda+1 \geq t+2-\lambda$. Par conséquent le nombre d'itérations de la boucle est fixe et indépendant du scalaire k.

5.5.2 Protection pour l'exponentiation modulaire

Bourrage. Il est nécessaire dans un premier temps de cacher la longueur en bits de l'exposant. Cela peut être réalisé avec la méthode de bourrage donnée en section 3.2.2, en utilisant l'ordre du groupe q (ne changeant ainsi pas le résultat final). Cependant, dans certains contextes, l'ordre du groupe peut être beaucoup plus élevé que les exposants (voir par exemple le protocole SRP dans le chapitre 7), donc cette méthode de bourrage ajoute un surcoût non négligeable.

Une alternative est donnée ci-dessous, qui utilise une valeur précalculée. Si la valeur maximale possible pour l'exposant x est ℓ, alors on peut le remplacer par

$$x_{\text{pad}} = x + 2^{\ell}.$$

Posons $y := g^{-2^{\ell}} \bmod p$ une valeur précalculée (qui peut être inscrite en dur dans le code source), alors $g^x \bmod p$ peut être calculé comme suit :

$$g^{x_{\text{pad}}} \cdot y \bmod p.$$

Le coût supplémentaire est modéré avec une multiplication et une exponentiation qui met en jeu un exposant qui est un bit plus long que le maximum possible.

Avec un usage de la scission Euclidienne, il est également nécessaire de cacher la longueur en bits du quotient. Supposons que l'exposant a été remplacé par x_{pad} avec la technique donnée ci-dessus, et que la scission Euclidienne est appliquée avec un diviseur aléatoire m de λ bits :

$$a = \lfloor x_{\text{pad}}/m \rfloor, \quad b = x_{\text{pad}} \bmod m.$$

Avec la technique de bourrage, la longueur en bits n de l'exposant x_{pad} est connue en avance, donc les deux longueurs en bits possibles pour le quotient sont $n-\lambda$ et $n-\lambda+1$. Le quotient a peut être remplacé par

$$a_{\text{pad}} = a + 2^{n-\lambda+1}$$

pour cacher sa taille. La valeur précalculée $z := g^{\lambda-n-1}$ est utilisée pour calculer $g^{x_{\text{pad}}}$:

$$g^{x_{\text{pad}}} \bmod p = (g^{a_{\text{pad}}} \cdot z)^m \cdot g^b \bmod p.$$

Une fois encore, le coût supplémentaire est modéré car il n'ajoute qu'une multiplication et éventuellement 1 ou 2 bits au quotient.

Remarque 6. Dans le cas où l'algorithme d'exponentiation est SSMA comme dans la bibliothèque CoreCrypto d'Apple, la longueur en bits du quotient peut directement être cachée avec le bourrage sur x. En effet, si ℓ est pair, la valeur $x_{\text{pad}} = x + 2^{\ell}$ est un entier de longueur en bits $\ell+1$ qui est impaire. Dans la section 5.3.2 il a été montré que le nombre d'itérations de la boucle avec le quotient est toujours le même dans cette situation (lorsque λ est pair).

Le correctif d'Apple. Une analyse de la mise à jour effectuée par Apple montre que la solution apportée est l'utilisation de l'algorithme Montgomery ladder en place de SSMA.

```
1  ccn_set(n, r1, s);
2  ccn_seti(n, r, 1);
3  cczp_to_ws(ws, zp, r, r);
4
5  cc_unit ebit = 0;
6  for (int bit = (int)ebitlen - 1; bit >= 0; --bit) {
7      ebit ^= ccn_bit(e, bit);
8      ccn_cond_swap(n, ebit, r, r1);
9      cczp_mul_ws(ws, zp, r1, r, r1);
10     cczp_sqr_ws(ws, zp, r, r);
11     ebit = ccn_bit(e, bit);
12 }
13
14 // Might have to swap again.
15 ccn_cond_swap(n, ebit, r, r1);
```

Listing 5.3 : Extrait de l'algorithme Montgomery ladder dans la mise à jour de CoreCrypto.

Contrairement aux versions données dans la section 3.2, l'algorithme est initialisé par le couple $(1, g)$ de telle sorte que si les premiers bits de l'exposant sont nuls alors une étape de l'algorithme ne change pas l'état du couple :

$$(1, g) \mapsto (1^2, 1 \cdot g) = (1, g).$$

Cependant, cela implique une multiplication par 1 et une élévation au carré de 1, ce qui pourrait être facile à détecter sur une trace comme il a été montré sur les captures dans la section 5.3.2. Pour éviter cet écueil, l'arithmetique modulaire a été remplacée par la représentation de Montgomery, donc l'unité n'est plus représentée par une valeur de poids de Hamming faible. Cela est réalisé avec la fonction `cczp_to_ws` avant la boucle dans le listing 5.3.

5.5.3 Protection pour la multiplication scalaire

La méthode de bourrage pour l'exponentiation modulaire est également applicable pour la multiplication scalaire. Le bourrage avec l'ordre du groupe peut aussi être utilisé pour masquer le quotient, ce qui ajoute un coût à l'exécution, mais peut rester raisonnable si le paramètre λ est peu elevé comme dans le cas de CoreCrypto.

Le correctif apporté par Apple consiste en une modification de l'algorithme Montgomery ladder pour être compatible avec des scalaires dont les bits de poids fort sont nuls. Cela a déjà été présenté et analysé dans la section 3.2.2. La technique de bourrage sur le reste de la division Euclidienne est supprimée car devenu inutile.

5.6 Exploitation de la vulnérabilité

Dans cette section on analyse dans quels contextes il peut être possible d'exploiter cette vulnérabilité. Dans un premier temps on liste les primitives cryptographiques et protocoles où un exposant ou scalaire est fixe, et on regarde dans un second temps le cas où celui-ci est éphémère.

5.6.1 Exposant ou scalaire fixe

Dans un échange Diffie-Hellman statique (EC)DH ou le chiffrement intégré (EC)IES, l'approximation peut être réalisée. Cependant cela ne permet pas de reconstruire les valeurs secrètes et l'impact est limité.

Dans certains protocoles, les exposants ou scalaires sont issus de façon déterministe d'un mot de passe. Obtenir une approximation donne alors un indicateur pour filtrer les mots de passe potentiels dans un dictionnaire. Ce sujet est abordé en détails dans le chapitre 7 avec les protocoles SRP et SPAKE2+ qui sont notamment présents dans la bibliothèque cryptographique d'Apple.

Enfin, il y a un autre cas d'intérêt qui est la version déterministe de ECDSA. Le nonce généré est identique dès lors que le message et la clé privée ne changent pas. Dans le cadre où un attaquant est en mesure d'observer la génération d'une même signature un certain nombre de fois, alors il peut obtenir une approximation du nonce. Ensuite il ne reste plus qu'à appliquer l'attaque décrite dans le chapitre 2 pour reconstituer la clé privée.

5.6.2 Exposant ou scalaire éphémère

Dans le cas où l'exposant ou le scalaire est éphémère l'approximation n'est pas possible car une seule mesure est obtenue. Cependant, un léger biais reste observable et se pose la question de savoir si celui-ci peut être exploité dans un ECDSA classique.

Si on regarde la figure 5.1, on voit que la probabilité que la longueur en bits du quotient $(n - \lambda + 1)$ est entre 0,5 et 1 pour la moitié haute des scalaires situés dans l'intervalle $[2^{n-1}, 2^n]$. Si l'observation indique que le quotient est de $n-\lambda+1$ bits, alors il est davantage probable que le scalaire se situe dans cette seconde moitié. La relation étant presque linéaire entre la position du scalaire et la probabilité, on peut raisonnablement estimer que la probabilité que le scalaire soit effectivement dans la moitié haute de l'intervalle est 0,75 (cela a été vérifié expérimentalement). En conséquence, on obtient l'équivalent d'un bit du nonce avec une probabilité supérieure à 0,5.

Cette quantité d'information est trop faible pour une attaque par réseau Euclidien pour résoudre le problème HNP décrit dans le chapitre 2. Le record actuel utilisant la méthode de Bleichenbacher avec l'analyse de Fourier concerne le cas où un bit par nonce est connu avec une probabilité légèrement inférieure à 1 [Ara+20], en recueillant des milliards de signatures. Il semble actuellement hors de portée d'adapter l'attaque où la probabilité est bien inférieure.

5.7 Conclusion

Dans ce chapitre on a présenté une vulnérabilité sur la méthode de randomisation par division Euclidienne d'un scalaire (ou d'un exposant) avant une multiplication scalaire (ou une exponentiation modulaire). Elle introduit une variation qui peut être observée par canaux auxiliaires pendant l'exécution et mène à l'approximation d'un scalaire fixe si un grand nombre d'observations peut être effectué.

La méthode fut originellement proposée pour être utilisée avec la double multiplication scalaire de Strauss-Shamir et on montre que la vulnérabilité reste malgré tout possible.

Un exemple d'une réelle utilisation de cette méthode de randomisation est la bibliothèque cryptographique d'Apple qui l'utilise à la fois pour protéger la multiplication scalaire et l'exponentiation modulaire dans certains contextes. Des expérimentations ont été menées pour montrer que la fuite est observable. Cette vulnérabilité a été communiquée avec Apple et suite à plusieurs échanges avec leurs ingénieurs des correctifs ont été appliqués, présents dans les dernières versions des systèmes d'exploitation de leurs produits.

6
Attaque en faute différentielle

Ce chapitre est issu principalement de l'article [Rus21c] présenté à la conférence *Indocrypt*, édition 2021.

Les attaques en faute différentielles sont des techniques puissantes pour casser une primitive cryptographique, où un attaquant perturbe la bonne exécution d'un calcul afin de récupérer une clé secrète. Ces attaques ont été appliquées sur la cryptographie à base de courbes elliptiques avec une variété de types de fautes, qui visent différents aspects d'une implémentation.

Dans ce chapitre, on présente une nouvelle méthode pour attaquer l'algorithme Montgomery ladder, l'attaque par *changement de signe de l'invariant*. On analyse aussi plusieurs méthodes de randomisation de scalaires et leur vulnérabilité contre les attaques en faute différentielle. Plusieurs simulations sont également proposées.

Sommaire

6.1 Introduction

Les premières attaques en faute différentielle, *Differential Fault Analysis* en anglais (DFA), ont été introduites sur des algorithmes de chiffrements par blocs [BS97], et sur RSA [BDL97]. La faute injectée modifie le comportement de l'exécution et résulte en une sortie erronée. Les effets de la faute sur la sortie sont comparés avec le résultat correct afin de compromettre la clé secrète.

On s'intéresse à appliquer une attaque DFA sur l'algorithme Montgomery ladder avec une faute dont l'objectif est de perturber l'échange conditionnel qui a lieu lors de chaque itération. L'effet de l'usage des formules spécifiques à cet algorithme (les formules `XZ` et `XYcoZ`) est envisagé. On montre ainsi qu'une validation de point ou une vérification de l'invariant ne sont pas des mesures suffisantes contre cette nouvelle attaque.

Ensuite, on montre que les méthodes de randomisation du scalaire ne protègent pas contre des attaques DFA lorsque l'aléa généré est trop petit. La première méthode analysée est la première contremesure de Coron [Cor99] qui ajoute un multiple de l'ordre du groupe pour masquer le scalaire, tandis que les autres scindent le scalaire en plusieurs morceaux, soit de façon additive, multiplicative ou par une division Euclidienne [CJ01, TB02, CJ03].

Les attaques sont considérées dans le contexte de signatures ECDSA ainsi que le cas d'un scalaire fixe comme par exemple un échange de clé Diffie-Hellman statique. Entre autre, cela peut être combiné avec une attaque par réseau Euclidien pour reconstituer une clé privée en suivant les méthodes du chapitre 2.

L'organisation est la suivante. Dans la section 6.2 on introduit le principe de base d'une attaque DFA et comment elle peut être appliquée dans le cadre de génération de signatures ECDSA ou dans des échanges Diffie-Hellman statiques. Ensuite, la section 6.3 décrit l'attaque sur l'algorithme Montgomery ladder, puis dans la section 6.4 les méthodes de randomisation du scalaire et leur vulnérabilité contre une attaque DFA. Une évaluation pratique de la réalisation de la nouvelle attaque est donnée en section 6.5. Enfin, des contremesures sont discutées dans la section 6.6.

6.1.1 Travaux similaires

Les premières mentions d'une attaque DFA avec des courbes elliptiques furent présentées dans [BMM00]. La cible pour l'injection de faute est une coordonnée d'un point au cours de la multiplication scalaire, ce qui résulte en un point qui ne vérifie pas l'équation de la courbe. L'algorithme est exécuté à l'envers à la fois avec le résultat correct et celui qui est erroné en devinant les bits du scalaire secret traités après la faute. La comparaison avec la valeur correcte est utilisée pour vérifier quelles sont les valeurs correctement devinées. Cette attaque produit un point en dehors de la courbe elliptique, donc la mesure classique de validation de point en utilisant l'équation détecte la faute.

Une autre attaque DFA a été proposée dans [BOS06], avec l'avantage qu'une validation de point ne détecte pas la faute. En effet, cette attaque modifie le signe d'un point, donc il satisfait toujours l'équation de la courbe. Un exemple est donné pour la réalisation de l'effet souhaité avec la représentation NAF du scalaire secret, et les auteurs déclarent que cela pourrait être adapté à l'algorithme Montgomery ladder lorsque la coordonnée y est utilisée.

Dans la même lignée, des attaques DFA telles que la validation de point ne détecte pas de faute ont été présentées dans [SM09, SM10] sur l'algorithme Montgomery ladder. Les fautes en questions sont le saut d'une ou plusieurs opérations de l'algorithme de façon à ce qu'un bit du scalaire ne soit pas traité (dans le premier article), ou par le saut d'une multiplication ou d'un carré avec RSA (dans le second article, mais adaptable avec les courbes elliptiques en remplaçant avec le doublement et l'addition de points). À la suite de la faute, il est possible de retrouver les bits du scalaire traités après (ou avant) que la faute soit injectée.

La nouvelle attaque présentée dans ce chapitre est analogue aux précédentes sur Montgomery ladder, avec une faute qui ne fait pas quitter le point de la courbe. Mais elle partage également des similarités avec l'attaque du changement de signe de [BOS06], car l'objectif est de changer le signe de l'invariant de boucle implicite de l'algorithme.

6.2 Préliminaires

Dans cette section, on présente le principe d'une attaque DFA et comment elle peut s'appliquer dans le contexte de signature ECDSA ou d'un échange Diffie-Hellman statique.

6.2.1 Principe de base d'une attaque DFA

L'objectif de l'attaque présentée dans ce chapitre est de récupérer une information partielle d'un scalaire secret (par exemple quelques bits, mais pas nécessairement). Cela se réalise en deux phases :

1. Une faute est injectée pendant la multiplication scalaire avec un scalaire secret ;
2. Une analyse du résultat erroné et sa différence avec le bon.

L'idée repose sur une faute telle que la différence est un point qui dépend d'une partie du scalaire qui peut être trouvée avec un effort raisonnable de l'attaquant, soit par une recherche exhaustive, soit par une recherche de logarithme discret dans un intervalle restreint.

6.2.2 Signature ECDSA avec faute

Le schéma de signature ECDSA a été présenté dans la section **1.3.3**, mais on rappelle brièvement ici la composition d'une signature. Étant donné un message M, la signature consiste en un couple (r, s) tel que

$$\begin{cases} r = x_Q & \mod q \\ s = k^{-1}(\mathrm{H}(M) + \alpha r) & \mod q, \end{cases}$$

où $Q = [k]P$ avec k un nonce généré aléatoirement, et α la clé privée du signataire.

Si une faute est effectuée pendant la multiplication scalaire telle que la sortie soit Q', alors la signature obtenue (r', s') est

$$\begin{cases} r' = x_{Q'} & \mod q \\ s' = k^{-1}(\mathrm{H}(M) + \alpha r') & \mod q. \end{cases}$$

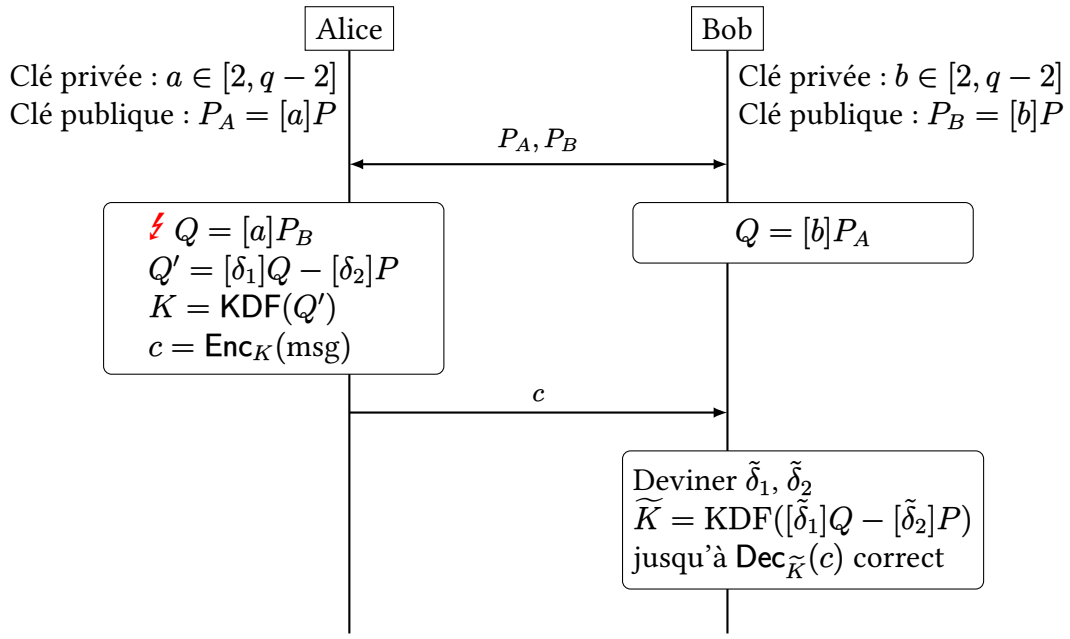


Figure 6.1 : Attaque DFA sur un échange Diffie-Hellman statique.

La signature fautée ne permet pas de reconstruire la signature complète (r, s), mais le point Q, de qui r est dérivé, peut l'être en utilisant le point public du signataire à partir de la relation de la vérification de signature :

$$[\mathrm{H}(M)s'^{-1}]P + [r's'^{-1}]P_{\text{pub}} = [k]P = Q.$$

Le point Q' peut lui aussi être obtenu en relevant l'entier r' vers un point de la courbe elliptique. Cependant, la valeur $x_{Q'}$ a été réduite modulo le nombre premier q. Il a été montré qu'en dehors de Q', il y a au plus quelques points possibles [Bar+16]. Comme le nombre premier q est généralement le cardinal de la courbe et est très proche de celui du corps de base, alors il est probable qu'il n'y ait que deux points possibles : Q et Q'.

Par conséquent, il est possible de récupérer à la fois la sortie correcte et celle erronée, ce qui est un aspect central d'une attaque en faute différentielle.

6.2.3 Diffie-Hellman statique avec faute

Pour un échange Diffie-Hellman statique, l'attaque a besoin d'être adaptée. L'objectif est de trouver la clé secrète de l'autre pair en injectant une faute pendant le calcul du secret partagé. L'attaque est résumée dans la figure 6.1.

L'attaquant (Bob) lance un échange de clé avec Alice et peut calculer le secret partagé $Q = [b]P_A = [a]P_B$, où (a, P_A) et (b, P_B) sont respectivement les paires de clés d'Alice et Bob. Le calcul du secret partagé par Alice est perturbé par l'attaquant avec une faute, ce qui résulte en un point Q' tel que

$$[\delta_1]Q - Q' = [\delta_2]P,$$

où δ_1 et δ_2 sont liés à une portion réduite de la clé privée a d'Alice.

Alice chiffre la communication avec une clé symétrique dérivée de Q', et l'attaquant reçoit le message chiffré. À partir du bon résultat Q, l'attaquant effectue une recherche

exhaustive sur δ_1 et δ_2 jusqu'à ce que le point erroné Q' soit trouvé (ce qui est confirmé lors d'un déchiffrement réussi).

Un cas classique est lorsque δ_1 vaut 1 et δ_2 sont les bits de poids faible de la clé privée d'Alice. Dans ce cas, l'attaque peut être répétée de façon itérative pour cibler d'autres portions du secret jusqu'à ce qu'il soit entièrement reconstruit.

6.3 Attaque DFA sur Montgomery ladder

Dans cette section on présente une nouvelle attaque DFA sur l'algorithme Montgomery ladder, notamment lorsque les formules `XZ` ou `XYcoZ` sont utilisées. Au passage, on redonne l'analyse du cas où l'addition ou le doublement est ignoré dans un calcul, présenté dans l'article [SM10].

Il est préférable d'être familier avec la description de l'algorithme donnée dans la section 3.2 avant de poursuivre la lecture.

6.3.1 Impact d'une faute

On rappelle que l'algorithme utilise un couple de points R_0 et R_1 qui satisfont l'invariant $R_1 - R_0 = P$ lors d'une exécution normale. À la suite d'une perturbation lors du traitement du bit k_j, ces deux points sont éventuellement modifiés. Alors, après l'exécution de la boucle, l'état interne du couple devient :

$$\begin{cases} R_0 = R' \\ R_1 = R' + I, \end{cases}$$

pour des points R' et I qui dépendent de la perturbation. En particulier, le point I remplace l'invariant P pour l'exécution des boucles restantes. Ainsi, d'après la description de l'algorithme dans la section 3.2, la sortie erronée de la multiplication scalaire est

$$Q' = [2^j]R' + [\overline{k}]I,$$

où $\overline{k} = (k_{j-1}, \ldots, k_0)_2$ sont les bits de poids faible du scalaire traités après la faute.

Cette description générique d'une perturbation facilite l'analyse en fonction du modèle de faute considéré.

Remarque 7. Dans le cas où la faute a pour effet que l'un des points R_0 ou R_1 n'est plus sur la courbe ou perturbe l'application des formules, les additions suivantes ne correspondent pas à la loi de groupe de la courbe elliptique (mais peut être vue comme une pseudo-addition). Excepté dans un cas particulier ci-dessous, on examine généralement des situations où la faute « maintient » les points sur la courbe elliptique.

6.3.2 Doublement ou addition ignorés

Dans [SM10] le modèle du saut d'instruction a été considéré pour attaquer l'algorithme Montgomery ladder : un carré ou une multiplication sont ignorés pendant le calcul d'une exponentiation modulaire avec RSA. L'équivalent pour les courbes elliptiques est d'ignorer un doublement ou une addition de points. On présente ci-dessous les conséquences.

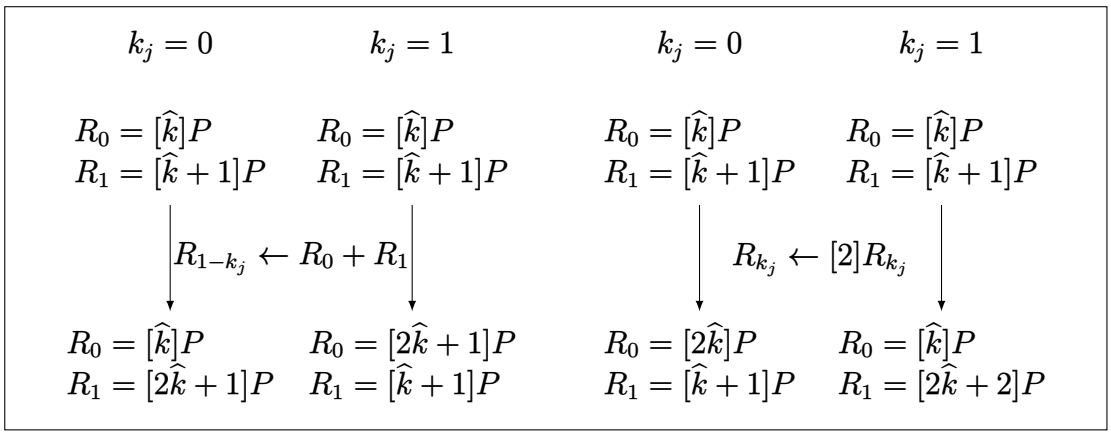


Figure 6.2 : Une étape de Montgomery ladder avec une opération ignorée (à gauche : doublement ignoré ; à droite : addition ignorée).

Dans cette partie on note $\widehat{k} = (k_{n-1}, \ldots, k_{j+1})_2$ les bits de poids fort du scalaire traités avant la faute, en excluant k_j.

Si seule l'addition est effectuée lors du traitement du bit k_j, alors à l'issue de la boucle on a $R' = [\widehat{k}]P$ et $I = [\widehat{k}+1]P$ lorsque k_j vaut 0, ou $R' = [2\widehat{k}+1]P$ et $I = [-\widehat{k}]P$ lorsque k_j vaut 1 (voir figure 6.2). Donc le résultat final de la multiplication scalaire est

$$Q' = \begin{cases} [\widehat{k}2^j + (\widehat{k}+1)\overline{k}]P & \text{si } k_j = 0, \\ [(2\widehat{k}+1)2^j - \widehat{k}\overline{k}]P & \text{si } k_j = 1. \end{cases}$$

Il est ainsi possible d'éliminer la dépendance aux bits de poids fort en calculant une différence avec les points Q et Q' :

$$\begin{cases} [2^{j+1}]Q' - [2^j + \overline{k}]Q = [\overline{k}2^j - \overline{k}^2]P & \text{si } k_j = 0, \\ [2^{j+1}]Q' - [2^{j+1} - \overline{k}]Q = [\overline{k}^2 - \overline{k}2^j]P & \text{si } k_j = 1. \end{cases}$$

Une recherche exhaustive sur les bits de poids faible est réalisée jusqu'à obtention de l'égalité (ou rejet si la faute n'a pas eu l'effet escompté).

La même méthode peut être appliquée dans le cas où c'est l'addition de points qui est ignorée lors du traitement du bit k_j. À l'issue de la multiplication scalaire on obtient :

$$Q' = \begin{cases} [(2\widehat{k})2^j + (1-\widehat{k})\overline{k}]P & \text{si } k_j = 0, \\ [\widehat{k}2^j + (\widehat{k}+2)\overline{k}]P & \text{si } k_j = 1. \end{cases}$$

Les différences ci-dessous entre les points Q et Q' ne dépendent que des bits de poids faible du scalaire :

$$\begin{cases} [2^{j+1}]Q' - [2^{j+1} - \overline{k}]Q = [\overline{k}^2]P & \text{si } k_j = 0, \\ [2^{j+1}]Q' - [2^j + \overline{k}]Q = [2^{j+1}\overline{k} - 2^{2j} - \overline{k}^2]P & \text{si } k_j = 1. \end{cases}$$

6.3.3 Nouvelle attaque : changement de signe de l'invariant

Dans cette partie on présente une nouvelle proposition pour attaquer l'algorithme Montgomery ladder. Le principe est d'inverser les points du couple (R_0, R_1) après le traitement du bit k_j (voir la section 6.5 pour des exemples de la manière dont cela peut être réalisé) :

$$\begin{cases} R_0 = [\widehat{k}]P \\ R_1 = [\widehat{k}+1]P \end{cases} \xrightarrow[\text{faute}]{\lightning} \begin{cases} R_0 = [\widehat{k}+1]P \\ R_1 = [\widehat{k}]P \end{cases} \tag{6.1}$$

Pour simplifier les formules ci-dessous, la notation $\widehat{k}$ correspond à l'ensemble des bits de poids fort jusqu'au bit k_j inclus.

La valeur R_0 pour le traitement du bit suivant est $R' = [\widehat{k}+1]P$, et l'invariant pour le restant de la multiplication scalaire est le point $I = -P$. Ainsi, le résultat final est :

$$Q' = [(\widehat{k}+1)2^j - \overline{k}]P. \tag{6.2}$$

La différence ou la somme des points Q et Q' ne dépend alors que d'une partie du scalaire k :

$$Q - Q' = [2\overline{k} - 2^j]P. \tag{6.3}$$
$$Q + Q' = [\widehat{k}2^{j+1} + 2^j]P. \tag{6.4}$$

Selon que l'étape j où la faute a eu lieu est plus proche du début ou de la fin, l'une de ces formules peut être utilisée pour retrouver soit les bits de poids faible dans le premier cas, soit les bits de poids fort dans le second. Une recherche exhaustive peut être réalisée ou éventuellement une recherche de logarithme discret dans un intervalle restreint (on peut facilement isoler $\overline{k}$ et $\widehat{k}$ dans chacune des formules).

Bien que ce type de faute ne peut être détecté par une validation de point, une vérification de l'invariant révèle qu'un mauvais calcul a eu lieu. Mais cela n'est vrai que pour des formules classiques : dans la suite on étudie le cas des formules spécifiques à l'algorithme Montgomery ladder où en particulier la vérification de l'invariant ne peut pas détecter la faute.

Remarque 8. Dans le cas particulier où $\overline{k} = 2^{j-1}$ les points Q and Q' sont égaux, donc la faute n'a aucun effet.

Étape inconnue. Les équations (6.3) et (6.4) montrent que les points obtenus ne dépendent pas seulement des bits de poids faible ou fort du scalaire secret, mais également de l'étape où la faute a lieu. Cela peut résulter en plusieurs valeurs candidates si plusieurs étapes j sont considérées dans l'analyse.

Des choix conservateurs peuvent être faits pour lever l'indéterminée, en perdant quelques bits du scalaire. Supposons que le logarithme discret $d = 2\overline{k} - 2^j$ du point différentiel de l'équation (6.3) a été obtenu, mais avec j inconnu. On peut calculer $d/2 \bmod 2^i$ pour un entier i dont on s'attend à être plus petit que j (par exemple 5 pour i alors que j vaut 10), et ainsi les i bits de poids faible sont récupérés.

Dans le cas où d est négatif, alors le plus petit i tel que $d/2 + 2^i$ soit positif peut être choisi, car $\overline{k}$ est censé être positif.

La perte de précision n'a pas un impact si important pour une attaque contre ECDSA, car la méthode à base de réseau Euclidien peut réussir avec quelques bits par nonce seulement.

6.3.4 DFA avec les formules XZ pour coordonnées projectives

La particularité de ces formules est que la coordonnée y des points n'est pas utilisée. Cependant, le bon fonctionnement de la formule d'addition repose sur le fait que l'abscisse affine x de la différence $R_1 - R_0$ est fixe.

En exécution normale, cette différence est le point P, l'entrée de la multiplication scalaire. Si le doublement ou l'addition est ignorée pendant le calcul, on a vu dans la section 6.3.2 que l'invariant est modifié et devient un point qui dépend des bits de poids fort déjà traités par l'algorithme. En conséquence toutes les additions de points qui suivent ne sont pas calculées correctement et l'analyse différentielle ne peut être réalisée.

Cependant, la nouvelle attaque sur le changement de signe de l'invariant proposée en section 6.3.3 n'a pas cet écueil. En effet, l'invariant est remplacé par $-P$ qui a la même coordonnée affine x que l'invariant d'origine, donc les additions de points dans les étapes qui suivent sont calculées correctement et l'analyse différentielle peut être réalisée.

De plus, un effet annexe intéressant se produit pour la reconstruction de la coordonnée y manquante, car les formules utilisent les deux coordonnées de l'invariant. Dans le cadre de l'attaque, l'invariant est passé de (x_P, y_P) à $(x_P, -y_P)$, mais le code exécuté prend généralement les valeurs de l'invariant d'origine figés dans des registres qui ne sont pas touchés pendant l'exécution. Comme on peut le voir dans les formules de reconstruction données dans l'équation (3.2), cette différence de signe n'impacte que la coordonnée affine y de la sortie Q'. Par conséquent, la valeur obtenue Q' en sortie est la même que dans l'équation (6.2), mais avec un changement de signe (et passe alors avec succès le test de la validation de point). De toute manière, l'attaque contre ECDSA construit des candidats pour Q' et inclut le point $-Q'$ donc cela n'a aucun impact dans l'analyse par rapport à l'usage de formules classiques.

De plus, une vérification de l'invariant ne peut détecter la faute. En effet, les points R_0 et R_1 sont reconstruits en tant que $R'_0 = -R'_0$ et $R'_1 = -R_1$ comme il vient d'être expliqué ci-dessus, donc la différence

$$R'_1 - R'_0 = -(R_1 - R_0) = P$$

produit l'invariant P comme si aucune faute n'avait eu lieu.

6.3.5 DFA avec les formules XYcoZ pour coordonnées Jacobiennes

On s'intéresse désormais aux formules `XYcoZ` présentées dans la section 3.1.3. On rappelle qu'une particularité de ces formules est la nécessité que les points R_0 et R_1 ont la coordonnée projective Z en commun, mais que cette dernière n'est pas utilisée.

On montre dans un premier temps la conséquence lorsqu'une des opérations d'addition est ignorée, et comment l'attaque peut malgré tout être réalisable, puis le cas de l'attaque par changement de signe de l'invariant.

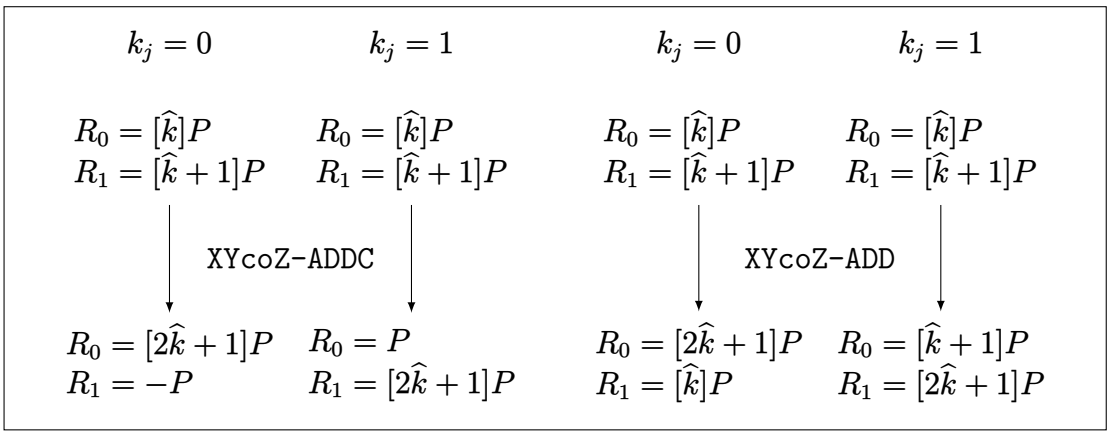


Figure 6.3 : Une étape de Montgomery ladder avec les formules XYcoZ et une opération ignorée (à gauche : XYcoZ-ADD ignoré ; à droite : XYcoZ-ADDC ignoré).

Opération XYcoZ-ADDC ignorée. Si l'addition conjuguée de points est ignorée pendant le traitement du bit k_j, alors à l'issue de la boucle on obtient $R' = [2\widehat{k}+1]P$ et le nouvel invariant $I = [-(\widehat{k}+1)]P$ lorsque $k_j = 0$, ou $R' = [\widehat{k}+1]P$ et $I = [\widehat{k}]P$ lorsque $k_j = 1$. Le résultat Q' de la multiplication est alors

$$Q' = \begin{cases} [(2\widehat{k}+1)2^j - (\widehat{k}+1)\overline{k}]P & \text{si } k_j = 0, \\ [(\widehat{k}+1)2^j + \widehat{k}\overline{k}]P & \text{si } k_j = 1. \end{cases}$$

Cependant, la reconstruction de la coordonnée Z en fin d'algorithme est basée sur l'invariant, supposé être le point P s'il n'y a pas eu d'erreur de calcul. Pour une reconstruction correcte, il faudrait que l'invariant $I = (x_I, y_I)$ soit utilisé dans la formule (3.1), tandis que le code exécuté va utiliser l'invariant d'origine, P, ce qui va aboutir à un point affine Q'' qui va probablement ne pas satisfaire l'équation de la courbe elliptique.

Néanmoins, il est possible de retrouver Q' à partir de Q'' en devinant l'invariant I suite à la faute. En effet, la relation entre les coordonnées de Q et Q' sont les suivantes :

$$x_{Q'} = x_{Q''}\left(\frac{y_I x_P}{x_I y_P}\right)^2 \quad \text{et} \quad y_{Q'} = y_{Q''}\left(\frac{y_I x_P}{x_I y_P}\right)^3 \tag{6.5}$$

Une recherche exhaustive sur I (qui dépend des bits de poids fort de k) peut être effectuée jusqu'à obtenir un point qui satisfait l'équation de la courbe elliptique.

Toutefois, la coordonnée y de Q'' n'est pas nécessairement disponible, mais le problème peut être contourné. L'invariant I vaut soit $[-(\widehat{k}+1)]P$, soit $[\widehat{k}]P$ selon le bit k_j, et dans les deux cas il y a une relation entre Q et Q' :

$$[\pm]Q' = [\mu]Q - [2^{j+1}\mu^2 - 2^j]P, \tag{6.6}$$

où μ est soit $(\widehat{k}+1)$ or $\widehat{k}$. L'analyse consiste alors à deviner cette valeur et vérifier la relation entre Q et Q'' à partir des équations :

1. Faire une recherche exhaustive sur μ pour construire une liste de valeurs candidates pour $x_{Q'}$ à partir de l'équation (6.6) et l'invariant $(x_I, y_I) = [\mu]P$;
2. Calculer la valeur candidate pour $x_{Q''}$ à partir de l'équation (6.5) pour chacun des candidats calculés pour $x_{Q'}$.

Dans les deux situations décrites dans la section 6.2, soit la valeur r de la signature pour ECDSA ou soit un déchiffrement correct pour ECDH statique confirme quelle valeur pour μ est correcte.

Il n'est pas encore possible à ce moment de savoir si μ correspond à $\widehat{k}$ ou $\widehat{k}+1$. Dans le cas de l'attaque contre ECDH, réaliser la faute à nouveau sur une étape quelques boucles plus loin permet de révéler laquelle des valeurs est correcte (tout en permettant de continuer à reconstruire le scalaire morceau par morceau). Dans le cas de l'attaque contre ECDSA, il n'est pas nécessaire de faire la distinction, car $\mu 2^{j+1}$ est une bonne approximation du nonce k :

$$-2^{j+1} \leq k - \mu 2^{j+1} < 2^{j+1}.$$

La reconstruction de la clé privée à base de réseau Euclidien peut s'appliquer (voir section 2.1.1).

Changement de signe de l'invariant. Les effets sont similaires au cas des formules XZ pour l'attaque proposée en section 6.3.3. Les calculs restent valides tant que les points partagent la même coordonnée projective Z et cette propriété n'est pas changée par l'attaque. Il ne reste donc qu'à observer l'effet sur la coordonnée Z. Le nouvel invariant est $(x_P, -y_P)$ et d'après la formule (3.1), cela provoque l'apparition d'un facteur -1 dans la coordonnée Z reconstruite pour les points R_0 et R_1. Mais lors de la mise sous forme affine, cela ne modifie que le signe de la coordonnée y (à cause des coordonnées Jacobiennes). Donc la valeur erronée Q' obtenue en sortie est un point valide de la courbe elliptique.

Enfin, comme avec les formules projectives XZ, vérifier l'invariant ne peut pas détecter la faute. On a $R_1 - R_0 = -P$ après la faute, et les points sont reconstruits en tant que $R'_0 = -R_0$ et $R'_1 = -R_1$ donc la différence $R'_1 - R'_0$ produit l'invariant d'origine.

Remarque 9. Une façon alternative de voir les choses pour les formules XZ et XYcoZ est que l'invariant n'est plus le point P, mais seulement sa coordonnée affine x. Comme celle-ci reste intacte même en cas de changement de signe, cela ne peut être détecté.

6.4 DFA avec randomisation du scalaire

Dans cette section on montre comment des attaques en faute différentielle peuvent être réalisées lorsqu'une méthode de randomisation du scalaire est appliquée, soit pour attaquer ECDSA, soit pour attaquer un scalaire fixe dont le résultat erroné peut être obtenu.

6.4.1 Randomisation par l'ordre du groupe

Cette méthode est une mesure classique proposée par Coron dans [Cor99]. Le scalaire secret k est remplacé par

$$k^\star = k + mq,$$

où m est un entier aléatoire de λ bits. Comme q est l'ordre du point de base P, alors on a

$$Q = [k^\star]P = [k]P + [mq]P = [k]P,$$

Algorithme 19 : Récupération de bits d'un scalaire randomisé dans un contexte DFA

Entrée : $k_0 := k \bmod 2^i$ avec $i \geq \lambda$, $T := [(k + mq) \bmod 2^{i+w}]P$, P
Sortie : $k \bmod 2^{i+w}$

1: **pour** $\tilde{m}$ **de** $2^{\lambda-1}$ **à** $2^\lambda - 1$ **faire**
2: $\quad \tilde{k}_0^\star \leftarrow (k_0 + \tilde{m}q) \bmod 2^i$
3: $\quad U \leftarrow [1/2^i \bmod q](T - [\tilde{k}_0^\star]P)$
4: $\quad \tilde{k}_1^\star \leftarrow$ log. discret de U en base P dans l'intervalle $[0, 2^w)$
5: $\quad$ **si** $\tilde{k}_1^\star$ est trouvé **alors**
6: $\quad\quad$ **retourner** $(\tilde{k}_1^\star 2^i + \tilde{k}_0^\star - \tilde{m}q) \bmod 2^{i+w}$

donc cela ne change pas le résultat final, mais cela implique un surcoût pour traiter les bits supplémentaires.

Notons $k^\star = \widehat{k}^\star 2^j + \overline{k}^\star$, où $\overline{k}^\star$ sont les j bits de poids faible du scalaire randomisé (et $\widehat{k}^\star$ ses bits de poids fort). On suppose qu'une attaque DFA révèle $\overline{k}^\star$ (comme dans les situations de la section précédente).

Attaque contre ECDSA. Dans la section 2.1.1, on a vu qu'il est nécessaire que k soit plus élevé que λ de façon à encadrer la partie inconnue du scalaire dans un intervalle de largeur $q/2^{j-\lambda}$.

En conséquence l'attaque devient difficile à réaliser lorsque le paramètre λ est élevé pour retrouver les bits de poids faible. Selon les situations, on peut se ramener à résoudre un logarithme discret dans un intervalle plus réduit (comme par exemple la situation de l'attaque présentée en section 6.3.3), qui, au mieux, est de largeur 2^j. Dans ce cas un algorithme tel que *pas de bébés – pas de géants* ou l'algorithme kangourou de Pollard (voir section 1.2) dont le coût est $O(2^{j/2})$ peut être utilisé.

Lorsque λ est 20, alors une faute à l'étape $j = 24$ donne un logarithme discret facile à calculer et est suffisant pour l'attaque contre ECDSA. Ainsi, le paramètre λ doit être choisi assez grand pour rendre l'attaque difficile voire impossible à réaliser.

Attaque contre un scalaire fixe. Si une attaque DFA permet de récupérer les bits de poids faible d'un scalaire randomisé, cela ne donne pas directement ceux de l'original. Cependant, il est possible de les trouver et de construire de façon itérative les bits suivants morceaux par morceaux avec des fautes en différentes positions en utilisant l'algorithme 19.

L'idée est similaire à celle présentée dans [SW15], où elle est appliquée avec des scalaires randomisés bruités. Tandis que dans le cas présenté ici, on possède un point qui dépend partiellement d'un scalaire randomisé.

Lorsque le point différentiel dépend des $i+w$ bits de poids faible, calculer un logarithme discret pour des valeurs de i élevées devient difficile. Cette dépendance est supprimée en utilisant la connaissance des i bits de poids faible : on teste toutes les valeurs possibles pour m et pour la bonne valeur le point construit en ligne 3 de l'algorithme 19 ne dépend que de w bits du scalaire randomisé. À partir de m et de ces bits, on en déduit les w bits suivant du scalaire d'origine.

Pour chaque position i le coût de l'algorithme dépend du paramètre λ et de la fenêtre w, soit un coût en $O(2^{\lambda-1+w/2})$ et doit être répété pour reconstruire k intégralement.

L'inconvénient est que les premiers λ bits de k doivent être connus pour que l'algorithme fonctionne. À partir de plusieurs scalaires randomisés où les λ bits de poids faible sont obtenus, on peut calculer une liste des valeurs possibles pour $k \bmod 2^\lambda$: la bonne valeur sera à l'intersection des listes. Il pourrait nécessiter de nombreux scalaires randomisés pour réduire le nombre de possibilités. Cependant, l'exécution de l'algorithme **19** avec une mauvaise valeur pour $k \bmod 2^i$ pourrait ne pas aboutir à un résultat pour produire le morceau suivant du scalaire, donc il pourrait être rejeté assez rapidement.

6.4.2 Randomisation par scission Euclidienne

Cette méthode a été présentée dans le chapitre **5**. On rappelle ici son fonctionnement. Le scalaire k est réécrit

$$k = am + b,$$

où m est un entier aléatoire de λ bits avec $a = \lfloor k/m \rfloor$ et $b = k \bmod m$. Ensuite la multiplication scalaire $Q = [k]P$ est calculée comme $Q = [m]([a]P) + [b]P$ avec trois multiplications scalaires individuelles et une addition de points pour recombiner.

On montre ici comment retrouver le diviseur aléatoire (ou un de ses facteurs) et le reste de la division Euclidienne à partir d'une seule faute. Le résultat est exploitable avec ECDSA de la même manière que si les bits de poids faible étaient révélés, ou avec un scalaire fixe en utilisant le théorème des restes chinois.

Principe général. Notons $R = [a]P$ la multiplication scalaire avec le quotient. La sortie est donnée par

$$Q = [m]R + [b]P.$$

Si le point R est connu, alors la stratégie de l'algorithme *pas de bébés – pas de géants* peut être appliquée pour récupérer m et b. D'un côté on calcule la liste des valeurs possibles pour $[b]P$ (les *pas de bébés*), de l'autre une liste des valeurs possibles pour $Q - [m]P$ (les *pas de géants*) jusqu'à ce qu'une collision révèle m et b.

La première liste dépend seulement du point de base, donc elle peut être calculée une seule fois et réutilisée par la suite. Comme m et b sont tous deux plus petit que 2^λ, la complexité en temps et en mémoire de l'algorithme est $O(2^\lambda)$ en nombre d'additions sur la courbe. L'attaque est ainsi réalisable pour un paramètre λ assez petit (on a vu comme exemple la bibliothèque cryptographique d'Apple où ce paramètre vaut 32, ce qui reste assez bas).

La cible est la multiplication scalaire $[m]R$ avec le diviseur aléatoire. Supposons qu'une faute a lieu qui résulte en le calcul $[m']R$ avec la différence $\delta = m - m'$ appartenant à un ensemble de taille B. Alors le résultat de la multiplication scalaire complète $Q = [m]R + [b]P$ est altérée en un point $Q' = [m']R + [b]P$, dont leur est différence est

$$Q - Q' = [\delta]R.$$

Pour chaque $\tilde{\delta}$ candidat pour δ on construit un candidat pour R :

$$\tilde{R} = [1/\tilde{\delta}](Q - Q')$$

La stratégie *pas de bébés – pas de géants* est ensuite appliquée pour obtenir des candidats $(\tilde{m}, \tilde{b})$ qui satisfont l'égalité

$$Q - [\tilde{m}]\tilde{R} = [\tilde{b}]P.$$

Il y a B valeurs possibles pour δ, donc le coût total est $O(2^\lambda B)$.

Gérer plusieurs candidats. Dans l'éventualité où la méthode produit plusieurs candidats pour le couple (m, b), il peut malgré tout être possible de récupérer de l'information sur le scalaire k.

Soit $(\tilde{m}, \tilde{b})$ un candidat et la valeur correspondante $\tilde{\delta}$, et les valeurs correctes (m, b) et δ qui font aussi partie des candidats.

Commençons avec le cas $b \neq \tilde{b}$. Comme $(\tilde{m}, \tilde{b})$ est un candidat, on a

$$[\tilde{m}]\tilde{R} + [\tilde{b}]P = [m]R + [b]P, \tag{6.7}$$

dont on déduit la relation

$$a \equiv \tilde{\delta}(\tilde{b} - b)(m\tilde{\delta} - \tilde{m}\delta)^{-1} \pmod{q}. \tag{6.8}$$

Le quotient a est retrouvé, donc le scalaire k peut être entièrement reconstruit et vérifié avec la relation $Q = [k]P$. Ce cas semble peu probable.

Si $b = \tilde{b}$, on peut déduire de l'équation (6.7) que $m\tilde{\delta} \equiv \tilde{m}\delta \pmod{q}$. Posons $\tilde{m}' = \tilde{m} - \tilde{\delta}$, et la relation devient

$$m\tilde{m}' = m'\tilde{m},$$

car on a supposé que les valeurs sont suffisamment petites donc l'égalité est vraie sur les entiers.

Soit $m = \mu g$ où $g = \gcd(m, m')$, alors on déduit que μ divise $\tilde{m}$. Par conséquent, s'il y a plusieurs candidats de la forme $(\tilde{m}_1, b), \ldots, (\tilde{m}_N, b)$, alors μ divise $\tilde{g} = \gcd(\tilde{m}_1, \ldots, \tilde{m}_N)$ qui divise m à son tour. Finalement, on a

$$k \equiv b \pmod{\tilde{g}}$$

avec $\tilde{g}$ un facteur de m.

Exemple avec le changement de signe de l'invariant. On applique l'attaque de la section 6.3.3. Après une faute sur la multiplication scalaire $[m]R$, alors le résultat obtenu est $[m']R$ avec $m' = \widehat{m}2^j + 2^j - \overline{m}$. La différence avec le résultat correct Q est

$$Q - Q' = [2\overline{m} - 2^j]R,$$

et la valeur $\delta = 2\overline{m} - 2^j$ ne dépend que des bits de poids faible de m. Ainsi, pour la partie *pas de bébés – pas de géants* de l'attaque ne nécessite une recherche exhaustive seulement sur les bits de poids fort de m. Au total, le coût est alors $O(2^\lambda)$.

6.4.3 Randomisation par scission multiplicative

Cette technique a été proposée dans [TB02]. Une valeur m aléatoire de λ bits est générée aléatoirement, et γ est défini pour satisfaire la relation

$$k \equiv m\gamma \pmod{q}.$$

La multiplication scalaire $Q = [k]P$ est calculée avec deux multiplications scalaires successives $R = [m]P$ et $Q = [\gamma]R$.

L'attaque DFA peut être appliquée avec une faute sur la seconde multiplication scalaire. Supposons qu'une faute a modifié le calcul de $[\gamma]R$ en $[\gamma']R$ où la différence $\delta = \gamma - \gamma'$ appartient à un ensemble de taille B. Alors le résultat de la multiplication scalaire $Q = [\gamma m]P$ est altéré en un point $Q' = [\gamma' m]P$, et leur différence est

$$Q - Q' = [\delta m]P.$$

Pour chaque valeur $\tilde{\delta}$ possible pour δ, on construit un candidat possible pour R :

$$\tilde{R} = [1/\tilde{\delta}](Q - Q').$$

Lorsque δ est correctement deviné, on récupère $R = [m]P$, et un calcul de logarithme discret dans le sous-intervalle contenant m révèle cette valeur. Comme m un entier positif de λ bits, le coût total est $O(B2^{\lambda/2})$. Le coût est similaire à l'attaque contre la randomisation par l'ordre du groupe, donc cela est réalisable pour des valeurs λ assez petites comme 20 ou 32.

Dans certaines situations, un seul logarithme discret est suffisant lorsque δ représente une petite quantité. Par exemple, si on applique la faute de la section **6.3.3** pendant le traitement du bit γ_j, alors la différence avec le bon résultat de la multiplication scalaire complète est

$$Q - Q' = [m(2\overline{\gamma} - 2^j)]P.$$

Il s'agit d'un point dont le logarithme discret en base P est inférieur à $2^{\lambda+j}$ (en valeur absolue), donc la complexité est $O(2^{(\lambda+j)/2})$. Une fois de plus, c'est réalisable en pratique pour des valeurs λ assez petites (comme suggéré dans l'article qui propose la méthode).

Ce logarithme discret est utilisable pour une attaque par réseau Euclidien contre ECDSA. En effet, on dispose de la relation

$$\frac{k - m\overline{\gamma}}{m2^j} \equiv \widehat{\gamma} \pmod{q}, \tag{6.9}$$

où $\widehat{\gamma}$ est relativement petit comparé à l'ordre q (au moins j bits de moins). Lorsqu'il est possible de séparer m de $(2\overline{\gamma} - 2^j)$ dans le logarithme discret calculé (par exemple si m est un nombre premier de la taille adéquate), alors un réseau Euclidien peut être construit.

Dans le cas où le scalaire k est fixe, une attaque par réseau Euclidien peut également être utilisée. En effet, l'équation (**6.9**) est une équation linéaire où les variables inconnues sont la clé privée k et la valeur $\widehat{\gamma}$ qui est différente à chaque exécution. En posant $u = 1/(m2^j) \bmod q$ et $v = -\overline{\gamma}/2^j \bmod q$, alors on a

$$|uk + v|_q < q/2^j,$$

ce qui correspond à la définition du problème HNP dans le chapitre **2**.

6.4.4 Randomisation avec la scission additive

Cette troisième méthode pour scinder le scalaire a été proposée dans [CJ01]. Le scalaire est coupé en deux de façon additive avec une valeur m générée aléatoirement :

$$k = m + \gamma.$$

Le résultat final est calculé avec deux multiplications scalaires $Q = [m]P + [\gamma]P$ et combiné avec une addition de points.

Une analyse de cette méthode a été réalisée dans [MV06], et un biais statistique a été démontré entre les bits des deux morceaux. Plusieurs attaques par injection de fautes ont été proposées mais qui nécessitent deux fautes par exécution (une sur un bit de m et une sur un bit de γ).

Une attaque différentielle par injection de faute pourrait être appliquée de façon similaire en réalisant la même faute sur chacun des deux multiplications scalaires aux mêmes positions.

6.5 Évaluation pratique

Dans cette section, on évalue comment l'attaque sur le changement de signe de l'invariant sur Montgomery ladder peut être réalisée en pratique, avec notamment des simulations pour valider.

6.5.1 Implémentations de Montgomery ladder

Dans la plupart des bibliothèques cryptographiques, la variante implémentée est celle de l'algorithme 4 où un seul échange conditionnel est réalisé par itération de la boucle.

Les différentes variantes selon les bibliothèques sont répertoriées dans le tableau 6.1, accompagné d'autres informations utiles comme les formules. Les courbes elliptiques sous forme de Montgomery comme la courbe `Curve25519` sont également données. Bien qu'elles ne soient pas utilisées avec ECDSA, l'attaque peut être appliquée dans les situations où un attaquant peut obtenir les résultats corrects et erronés.

Table 6.1 : Vue d'ensemble de l'algorithme Montgomery ladder dans quelques bibliothèques cryptographiques.

Bibliothèque	Init.	Variante	Formules	Remarques
Forme de Weierstraß				
OpenSSL 1.1.1k	$(P, 2P)$	Alg. 4*	`XZ`	
LibreSSL 3.2.4	$(P, 2P)$	Alg. 4*	Jac	
CoreCrypto	$(P, 2P)$	Alg. 5	`XYcoZ`	Point valid., Scission Eucl.
Forme de Montgomery				
SymCrypt	$(\mathcal{O}, P)$	Alg. 4	`XZ`	
Mbed TLS	$(\mathcal{O}, P)$	Alg. 3	`XZ`	
libsodium †	$(\mathcal{O}, P)$	Alg. 4	`XZ`	

* Le code source est légèrement différent, mais le code compilé correspond à l'algorithme 4.

† Implémentation `ref10` de `Curve25519` présente dans d'autres bibliothèques.

On peut noter que dans certains cas une attaque par canaux auxiliaires peut être suffisante, mais ces dernières sont liées à la façon dont l'échange conditionnel est mis en œuvre. Il y a la méthode avec un masque binaire, où l'échange est réalisé avec un masque composé de bits valant tous 1 ou tous 0. Un attaque par profilage a été appliquée sur la fuite

venant de l'opérateur binaire AND [Nas+16]. Dans le cas de Mbed TLS, une multiplication par 0 ou par 1 est utilisée pour réaliser l'échange, et est également vulnérable à une attaque par profilage [LLF20]. Dans les deux cas cités, une seule trace révèle le scalaire entier.

Sous l'hypothèse d'une implémentation protégée contre ces attaques, une attaque par injection de faute devient pertinente. Dans la suite on présente une stratégie pour réaliser l'attaque présentée dans la section 6.3.3 sur une implémentation basée sur l'algorithme 4.

6.5.2 Réalisation de la faute

En prérequis il est d'abord nécessaire d'avoir un accès à l'appareil exécutant le code, et l'attaquant doit être en mesure de perturber le calcul à un moment spécifique dans l'exécution.

Saut d'instruction. Le premier modèle de faute considéré est le saut d'instruction qui fut appliqué avec succès en pratique sur une exponentiation RSA pour ignorer l'exécution d'un carré [SH08]. Plus récemment, ce modèle de faute a été appliqué sur l'algorithme de décompression de point pour que ce dernier appartienne à une courbe où le logarithme discret est plus facile à calculer [BG15, TT19].

L'effet décrit dans la section 6.3.3 peut être réalisé si la ligne 5 de l'algorithme 4 est ignorée lors d'une itération de la boucle. En effet, la variable pbit au début de la boucle contient la valeur du bit précédemment traité et garde trace du l'état actuel du couple (R_0, R_1) si bien que l'invariant de boucle est :

$$R_{1-\text{pbit}} - R_{\text{pbit}} = P.$$

Ainsi, si la ligne « pbit $\leftarrow$ pbit $\oplus\, k_i$ » n'est pas exécutée et que le bit k_i vaut 1, alors la variable pbit n'est pas mise à jour :

- Si pbit valait 0, on a $R_1 - R_0 = P$, les points ne sont pas échangés, donc on a encore $R_1 - R_0 = P$;
- Si pbit valait 1, on a $R_0 - R_1 = P$, les points sont échangés, donc on a désormais $R_1 - R_0 = P$.

Dans les deux cas, pbit récupère la valeur 1 en fin de boucle, donc à partir du tour suivant on a

$$R_{1-\text{pbit}} - R_{\text{pbit}} = R_0 - R_1 = -P,$$

et l'invariant de boucle a changé de signe.

Une alternative est de viser la ligne « pbit $\leftarrow k_i$ ». Si le bit k_i est différent de la valeur dans pbit, alors à nouveau cette variable ne sera pas cohérente avec l'état actuel du couple (R_0, R_1), mais ne sera effectif qu'à partir du tour suivant.

Remarque 10. Dans la moitié des cas, sauter l'une des instructions citées n'a aucun effet (si le scalaire k est effectivement aléatoire) et donc cela produit un résultat final correct.

Faute dans un registre. Une méthode couramment utilisée pour effectuer un échange conditionnel est d'utiliser des masques binaires comme présenté dans l'algorithme 20. La partie qui nous intéresse ici est la façon dont le masque est construit. Généralement,

Algorithme 20 : Échange conditionnel en temps constant de deux valeurs avec des opérateurs binaires (en commentaire : version alternative)

Entrée : (w_0, w_1), bit b
Sortie : (w_b, w_{1-b})

mask $\leftarrow (b, \ldots, b)_2$
tmp $\leftarrow$ mask $\wedge\ (w_0 \oplus w_1)$ ▷ tmp $\leftarrow w_0$
$w_0 \leftarrow w_0 \oplus$ tmp ▷ $w_0 \leftarrow (w_0 \wedge \neg\text{mask}) \vee (w_1 \wedge \text{mask})$
$w_1 \leftarrow w_1 \oplus$ tmp ▷ $w_1 \leftarrow (w_1 \wedge \neg\text{mask}) \vee (\text{tmp} \wedge \text{mask})$
retourner (w_0, w_1)

celle-ci est réalisée en exploitant la représentation binaire de l'entier -1, qui n'est autre qu'un mot machine composé uniquement de bits à 1. Ainsi, la quantité $-b$ donne un masque nul si b vaut 0, et donne un masque binaire composés de 1 si b vaut 1.

Cependant il existe aussi des constructions telles que toute valeur non nulle b aboutit à un masque binaire composé de 1. Si n est le nombre de bits des mots machine alors il est possible de construire le masque avec l'une des deux formules suivantes (la première est celle utilisée dans Mbed TLS et la seconde dans OpenSSL) :

$$-\Big((b \vee (-b)) \gg (n-1)\Big) \quad \text{ou} \quad \Big((\neg b \wedge (b-1)) \gg (n-1)\Big) - 1.$$

Une faute modifiant le bit b en une valeur quelconque non nulle va donc avoir l'effet de forcer la réalisation de l'échange conditionnel si la valeur d'origine est 0. Cela peut être réalisé par le modèle de faute aléatoire sur un registre. Il faut tout de même s'assurer que la modification effectuée n'impacte pas aussi la mise à jour de la valeur pbit pour le tour suivant.

6.5.3 Simulations

Plusieurs simulations ont été mises en place pour illustrer l'attaque et évaluer les différentes situations présentées. La première teste les deux modèles de fautes précédents sur l'implémentation de Montgomery ladder dans OpenSSL avec des fautes simulées avec GDB, le débogueur du projet GNU. La seconde utilise l'émulateur Unicorn [1] pour tester les effets de mauvaises instructions ignorées. Enfin, des simulations sont aussi disponible en Python avec la bibliothèque `fpylll` pour réaliser l'attaque par réseau Euclidien.

Toutes ces simulations sont disponibles publiquement sur le lien suivant :

https://github.com/orangecertcc/dfa-ladder.

Fautes sur OpenSSL. Une partie du code assembleur de l'algorithme Montgomery ladder de OpenSSL compilé sur un Raspberry Pi est donnée dans le listing 6.1.

```
1 dd1e8:  mov    r1, r8
2 dd1ec:  ldr    r0, [sp, #8]
3 dd1f0:  bl     8b440 <BN_is_bit_set>      ; r0 <- bit courant k_i
4 dd1f4:  ldr    r6, [sp, #12]              ; r6 <- pbit
5 dd1f8:  mov    r3, r9
6 dd1fc:  ldr    r2, [fp, #8]
```

1. https://www.unicorn-engine.org/.

```
dd200:   ldr     r1, [r7, #8]
dd204:   sub     r8, r8, #1
dd208:   eor     r6, r6, r0                  ; r6 <- pbit XOR k_i
dd20c:   mov     sl, r0
dd210:   mov     r0, r6
dd214:   bl      8b554 <BN_consttime_swap>   ; échange coordonnée X si r0 = 1
dd218:   mov     r0, r6
dd21c:   mov     r3, r9
dd220:   ldr     r2, [fp, #12]
dd224:   ldr     r1, [r7, #12]
dd228:   str     sl, [sp, #12]               ; sauvegarde de k_i dans pbit
dd22c:   bl      8b554 <BN_consttime_swap>   ; échange coordonnée Y si r0 = 1
dd230:   mov     r0, r6
dd234:   mov     r3, r9
dd238:   ldr     r2, [fp, #16]
dd23c:   ldr     r1, [r7, #16]
dd240:   bl      8b554 <BN_consttime_swap>   ; échange coordonnée Z si r0 = 1
```

Listing 6.1 : Extrait du code assembleur de la fonction `ec_scalar_mul_ladder` dans OpenSSL.

On repère à l'adresse `0xdd208` l'instruction correspondant à la ligne « pbit $\leftarrow$ pbit $\oplus$ k_i » qui est à ignorer pour le premier modèle de faute considéré. Cela est facile à mettre en place avec GDB :

1. Identifier l'adresse correspondante dans le programme exécutée (car `0xdd208` est celle de la bibliothèque partagée `libcrypto` d'OpenSSL qui est liée au programme) ; dans notre cas on a `0xb6be68208`
2. Mettre un point d'arrêt à cette adresse ;
3. Ignorer cette instruction lors d'une des itérations en sautant directement vers l'instruction qui suit.

L'ensemble de ces étapes est résumé dans le listing 6.2 où le saut d'instruction a lieu lors de la 239^{e} itération avec la courbe `secp256k1`.

```
(gdb) b main
(gdb) r
(gdb) b *0xb6e68208
(gdb) ignore 2 238
(gdb) c
(gdb) jump *0xb6e6820c
(gdb) disable 2
(gdb) c
Continuing.[Inferior 1 (process 17874) exited normally]
```

Listing 6.2 : Simulation d'injection de faute par saut d'instruction avec GDB dans OpenSSL.

Le second modèle de faute peut être réalisé en modifiant le registre `r6` à l'issue de la même instruction. En effet, cette variable n'est utilisée que pour effectuer l'échange conditionnel qui n'est effectif que si la valeur de ce registre est non nulle. Avec GDB il suffit d'utiliser la commande `set $r6=0x55adab` à la place de la commande `jump` dans le listing 6.2.

Dans les deux modèles de fautes, les signatures générées permettent de reconstruire la clé privée avec succès.

Simulation avec Unicorn. L'émulateur Unicorn a été utilisé par l'intermédiaire de l'outil Rainbow [2], qui facilite le traçage des instructions d'un binaire. L'exécution peut être arrêtée à tout moment et l'instruction suivante peut être lue. Il est ainsi assez facile

2. https://github.com/Ledger-Donjon/rainbow.

de réaliser le modèle du saut d'instruction : l'instruction suivante est lue, puis et ignorée en reprenant à l'instruction située juste après (grâce au nombre d'octets de l'instruction ignorée).

L'arithmétique modulaire de grands entiers de la courbe `secp256r1` écrit en assembleur a été choisie (à partir d'OpenSSL), car elle n'a pas de dépendance externe et est plus facile pour fonctionner avec l'émulateur.

Deux exécutables ont été créés avec une implémentation de Montgomery ladder de l'algorithme 4 : la première utilise les formules pour coordonnée Jacobiennes complètes, et la seconde les formules `XYcoZ`.

Les instructions liées aux lignes « pbit $\leftarrow$ pbit $\oplus\, k_i$ » et « pbit $\leftarrow k_i$ » sont bien présentes dans le code assembleur généré. Lorsqu'elles sont ignorées pendant une itération l'analyse aboutit et les bits de poids faible du scalaire sont obtenus.

Cependant une situation d'un faux positif a été repérée lorsque la faute touche la fonction qui extrait un bit du scalaire en début de boucle. La conséquence est que le bit extrait est erroné : le bit k_j est remplacé par $1 - k_j$, mais le reste de la multiplication scalaire est correctement effectué. Cela est équivalent à un *bitflip* sur le scalaire et en conséquence on a

$$Q - Q' = \begin{cases} [-2^j]P & \text{si } k_j = 0, \\ [2^j]P & \text{si } k_j = 1. \end{cases}$$

Si l'on regarde l'équation (6.3), l'analyse va aboutir et révéler que les bits de poids faible du scalaire sont nuls (alors que ce n'est pas nécessairement le cas). Pour éviter qu'une mauvaise signature soit prise en compte pour l'attaque contre ECDSA, il est préférable d'ignorer ce résultat (si j vaut 16, il n'y a qu'une chance sur 65536 que les 16 bits de poids faible soient effectivement nuls, donc de toute façon le rejet d'un résultat correct serait rare).

Un autre faux positif a été observé avec les formules pour coordonnées Jacobiennes : après un saut d'une instruction spécifique dans la fonction d'addition de points, un des points en entrée n'est pas chargé correctement et l'addition est effectuée avec le même point. La fonction de doublement est alors appelée, et l'analyse aboutit et donne une valeur incorrecte.

Enfin, pour les formules `XYcoZ`, une vérification de l'invariant a été appliquée en fin d'exécution basé sur une proposition donnée dans [VS12]. La fonction `XYcoZ-ADD` a été adaptée pour que la différence des entrées soit calculée (l'invariant) au lieu de la somme. Une fois la coordonnée Z récupérée et les points convertis sous leur représentation affine, l'opération binaire `XOR` est appliquée entre l'invariant calculé I, l'invariant correct P et la sortie Q :

$$Q \oplus I \oplus P.$$

Si l'invariant calculé est correct, alors il devrait s'annuler avec P. Comme attendu d'après la section 6.3.5, I et l'invariant P coïncident lorsque la faute est correctement injectée (ou lorsqu'il n'y a pas de faute), donc la sortie est valide et n'empêche pas la réalisation de l'attaque.

Simulation avec Python. Les simulations Python incluent des tests pour l'attaque par réseau Euclidien avec ou sans randomisation du scalaire, avec les formules pour

coordonnées Jacobiennes classiques, XYcoZ ou les formules projectives XZ. La faute est réalisée en omettant la ligne « pbit $\leftarrow k_i$ » en fin de boucle.

Bien que l'attaque par réseau Euclidien a un coût négligeable dans les différentes configurations, on note une différence dans l'efficacité dans le cas de la scission multiplicative. En effet, l'attaque contre un scalaire fixe nécessite un réseau Euclidien plus large par rapport à l'attaque contre ECDSA lorsque le même nombre de bits du scalaire randomisé est obtenu. Rappelons que dans ce cas le scalaire k est randomisé en une valeur γ avec un entier aléatoire m tel que $k \equiv m\gamma \bmod q$. L'analyse DFA révèle m et j bits de γ. Lorsque j vaut 5 il est nécessaire d'avoir 54 signatures fautées en moyenne (comme dans le cas d'un scalaire non randomisé, voir tableau **2.1**), tandis que pour un scalaire fixe environ 70 points fautés doivent être récupérés.

Un explication partielle viendrait de la construction d'une des lignes dans la matrice de la base du réseau Euclidien, qui ne dépend que des j bits récupérés dans le cas d'un scalaire fixe, tandis que pour ECDSA ces entrées sont davantage diversifiées.

6.6 Contremesures

Dans certains cas présentés, une validation de point en fin d'algorithme suffit à détecter qu'une faute a eu lieu. Par contre dans le cas de l'attaque sur le changement de signe de l'invariant cette protection ne suffit pas, même dans le cas des formules XZ ou XYcoZ (il fut suggéré dans [EL09] de reconstruire la coordonnée manquante et ensuite réaliser la vérification, mais dans le contexte de l'attaque présentée dans [Fou+08]).

La vérification de l'invariant fut proposée dans [DHA11, VS12] contre des attaques en faute. D'après la section **6.3.3**, cela reste vrai en général, sauf dans les cas des formules XZ et XYcoZ (l'invariant reconstruit est celui attendu).

Cependant, une autre idée présentée dans [DHA11] peut empêcher l'attaque du changement de signe de l'invariant dans le cas des formules spécifiques. Elle est basée sur une randomisation du point de base (seconde contremesure de Coron [Cor99]) : l'algorithme est initialisé avec $R_0 = P + R$ et $R_1 = [2]P + R$ pour un point aléatoire R, et l'invariant est toujours égal à l'entrée P. À la fin de la multiplication scalaire on a $R_0 = [k]P + [2^{n-1}]R$, donc une soustraction par $[2^{n-1}]P$ est nécessaire pour obtenir le bon résultat. Dans l'attaque les points R_0 et R_1 sont inversés et l'invariant devient $-P$:

$$\begin{aligned} R_0 &= [\tilde{k}]P + [2^{n-1}]R \\ R_1 &= R_0 - P. \end{aligned}$$

On a vu que cela change le signe lors de la reconstruction affine après la récupération de la coordonnée manquante. Par conséquent, la soustraction avec le point aléatoire donnerait le résultat

$$Q' = -R_0 - [2^{n-1}]R = -[\tilde{k}]P - [2^n]R.$$

Sans connaissance du point R, la sortie Q' est inutile pour l'attaquant.

D'autres contremesures classiques sont également applicables telle que répéter les calculs deux fois et vérifier la cohérence. Pour réduire le coût, il fut proposé dans [BOS06] d'effectuer un second calcul sur une courbe elliptique $E_{p'}$ définie sur un corps $\mathbf{F}_{p'}$ plus petit, et le premier sur une courbe elliptique $E_{pp'}$ sur l'anneau $\mathbf{Z}/pp'\mathbf{Z}$. D'un côté la

réduction modulo p donne le résultat souhaité, et de l'autre la réduction modulo p' est vérifiée avec le calcul sur la courbe $E_{p'}$.

Une variante de la méthode précédente a été proposée dans [SM09] et [Joy12] où le second calcul est effectué sur un groupe auxiliaire en parallèle des opérations sur la courbe elliptique. Par exemple, cela peut être réalisé en ajoutant un entier aux coordonnées des points pour garder trace du logarithme discret des points avec les règles suivantes :

$$(P_1, \ell_1) + (P_2, \ell_2) = (P_1 + P_2, \ell_1 + \ell_2), \quad [2](P, \ell) = ([2]P, 2\ell).$$

Sans faute, le résultat attendu est le couple $([k]P, k)$ où k est le scalaire secret. Cette méthode devrait détecter la faute de l'attaque du changement de signe de l'invariant car la valeur auxiliaire est cohérente avec le point donc un changement dans ce dernier a aussi un effet sur cette valeur.

Enfin, dans le cadre de l'attaque contre ECDSA, il reste toujours possible de vérifier la signature à la fin des calculs.

6.7 Conclusion

Dans ce chapitre on a présenté une nouvelle attaque en faute différentielle sur l'algorithme Montgomery ladder. Selon deux modèles de fautes considérés (saut d'instruction ou modification aléatoire d'un registre), le flot d'exécution d'un programme est altéré ayant pour effet principal d'interchanger deux points. En conséquence, plusieurs bits consécutifs du scalaire secret peuvent être déterminés à partir de la différence du résultat correct avec celui qui est erroné.

Aussi, cette attaque contourne certaines des mesures de protections contre des attaques par injection de faute sur les courbes elliptiques. En conséquence, un soin particulier est nécessaire dans le choix des contremesures lorsque ce genre d'attaquant fait partie du modèle de sécurité.

Enfin, des éléments sont présentés prouvant que la randomisation du scalaire par les méthodes les plus usuelles n'est pas suffisante pour déjouer des attaques DFA. Pour cela, il est nécessaire que les valeurs aléatoires utilisées dans ces méthodes soient suffisamment petites pour que l'attaque soit réalisable. Cependant, on peut noter que c'est souvent le cas pour limiter le coût supplémentaire qu'elles impliquent.

7

Attaque par dictionnaire sur les protocoles SRP et SPAKE2+

Le contenu de ce chapitre est issu d'une partie de l'article [Rus21b].

On s'intéresse à l'exploitation de fuites qui proviennent de canaux auxiliaires et concernent des exposants ou des scalaires. Dans les protocoles d'authentification par mot de passe tels que SRP ou SPAKE2+, cela révèle des indicateurs sur les mots de passe permettant de réaliser une attaque par dictionnaire.

Sommaire

7.1 Introduction

Les protocoles SRP et SPAKE2+ appartiennent à la classe des protocole d'authentification et d'échange de clé basé sur un mot de passe, *Password-based Authenticated Key-Exchange* (PAKE) en anglais. En particulier, ils attribuent un rôle asymétrique dans un modèle où seul le client connaît le mot de passe, tandis que le serveur détient un vérificateur. Les avantages par rapport à un échange de clé Diffie-Hellman est que leur conception assure une authentification mutuelle et protège le mot de passe contre un espionnage des communications entre le client et le serveur.

Les calculs dans le protocole SRP nécessitent plusieurs exponentiations modulaires dans un corps fini défini par un grand nombre premier. Il en est de même pour SPAKE2+, mais ce dernier est également compatible avec les courbes elliptiques. Les exposants sont secrets et certains sont spécifiquement dérivés d'un mot de passe. La connaissance de ces exposants est suffisante pour se faire passer pour un client.

Récemment, il a été montré que des implémentations d'un protocole PAKE mènent à des différences de temps d'exécutions directement liés au mot de passe [VR20, BFS20]. Les auteurs utilisent cette fuite comme critère pour filtrer des mots de passe dans un dictionnaire. Bien qu'il ne s'agit pas d'un exposant qui est dérivé du mot de passe, l'approche peut être adaptée aux protocoles SRP ou SPAKE2+ si l'algorithme d'exponentiation ou de multiplication scalaire est vulnérable à des attaques par canaux auxiliaires. Cela est notamment l'objet de l'attaque PARASITE [BFS21], exploitant une vulnérabilité par attaque cache contre le protocole SRP.

La section 7.2 donne une description générale du protocole SRP. L'attaque contre le protocole est introduite dans la section 7.3, où le client est supposé utiliser un algorithme d'exponentiation vulnérable, même dans le cas d'une fuite mineure. Enfin, la section 7.4 présente le protocole SPAKE2+ et l'adaptation de l'attaque.

7.2 Le protocole Secure Remote Password

Le protocole Secure Remote Password (SRP) fut introduit dans [Wu98] et est décrit sous la version SRP-3 dans RFC 2945 [Wu00], et la version SRP-6 pour une utilisation dans l'authentification TLS est décrite dans RFC 5054 [Tay+07]. La différence entre ces versions est mineure et concerne des vulnérabilités qui ne sont pas pertinentes ici. Dans la suite, le protocole sera supposé être la version SRP-6 ou sa variante SRP-6a. Il s'agit d'un protocole à base de mots de passe dont le but principal est d'établir un accord sur une clé entre deux entités sous un modèle client/serveur de façon authentifiée. Le mot de passe n'est connu que du client, tandis que le serveur conserve un *vérificateur*. Ils s'authentifient entre eux en envoyant des données éphémères de façon similaire à un échange Diffie-Hellman. Le résultat d'une authentification réussie est le partage d'un secret cryptographique entre les deux entités.

Le protocole est conçu tel que le mot de passe ne peut être extrait des communications. Ainsi, une personne qui monitore les communications n'est pas en mesure de monter une attaque par dictionnaire en mode hors ligne, ce qui rend le protocole sécurisé même en cas de mot de passe faible (par exemple un mot de passe de quelques chiffres). De plus, en cas de compromission du serveur, les données fuitées sont insuffisantes afin de se faire

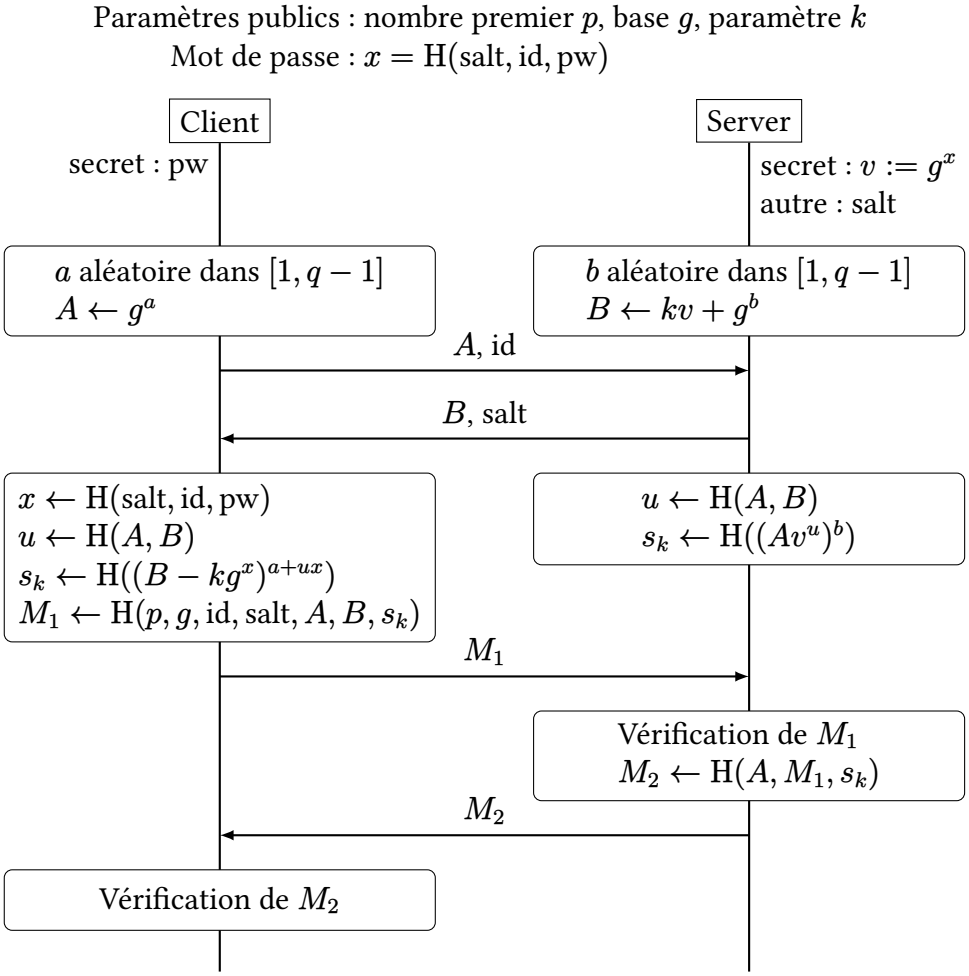


Figure 7.1 : Le protocole Secure Remote Password (SRP), version SRP-6(a).

passer pour le client (si le mot de passe est fort, car dans cette situation une attaque par dictionnaire serait toujours possible).

La description du protocole est donnée ci-dessous, ainsi que dans la figure 7.1.

7.2.1 Description

Initialisation. Tous les calculs sont réalisés sur un corps fini premier avec un grand nombre premier p et un élément g qui génère un grand sous-groupe d'ordre q. En particulier, il est recommandé que p soit un nombre premier sûr pour une meilleure sécurité (i.e. $p = 2q + 1$ avec q premier). De tels paramètres sont définis dans [Tay+07] pour être utilisé dans l'authentification TLS.

Avant une authentification, un mot de passe **pw** et un sel doivent être choisis par le client, et un exposant secret x en est dérivé comme suit :

$$x = \mathrm{H}(\mathrm{salt} \mid \mathrm{H}(\mathrm{id} \mid \text{“:”} \mid \mathrm{pw})),$$

où H est une fonction de hachage sécurisée et « | » une concaténation. Par simplicité, les constructions spécifiques en entrée de la fonction de hachage seront omises et seules les valeurs qui influent sur la sortie seront indiquées. Le client calcule $v := g^x \bmod p$, et le

serveur conserve le vérificateur v et le sel. Le client n'a pas besoin de conserver l'exposant x. La façon dont le serveur récupère le vérificateur et le sel dépend de l'application qui utilise le protocole et ne fait pas partie de la description.

Début de l'authentification. Dans la première phase le client envoie son identifiant et la valeur publique $A := g^a \bmod p$ où a est un exposant éphémère généré aléatoirement. Le serveur récupère le vérificateur et le sel à partir de l'identité du client, puis envoie le sel et la valeur publique $B := kv + g^b$ où b est un exposant éphémère généré aléatoirement, et k un paramètre public du protocole ($k = 3$ dans SRP-6, ou est généré de façon déterministe à partir des autres paramètres dans la variante SRP-6a).

Calcul du secret partagé et preuve. Les deux entités peuvent calculer une valeur u à partir des valeurs publiques échangées A et B. Ensuite, on a l'égalité suivante :

$$(B - kg^x)^{a+ux} \bmod p = (Av^u)^b \bmod p.$$

Le client peut calculer l'expression de gauche (en utilisant le sel pour obtenir x), tandis que le serveur peut calculer celle de droite. Une clé de session s_k est construite à partir de ce secret partagé. Une dernière étape est nécessaire pour prouver à l'un et à l'autre que la clé est identique en échangeant des preuves. L'authentification échoue en cas de désaccord.

7.2.2 Sécurité

Pour se faire passer pour un client il est nécessaire de connaître l'exposant x pour construire le secret partagé avec le serveur. Cette valeur n'est ni échangée, ni stockée, et ne peut être dérivée des communications. Dans le cas où le serveur est compromis, l'attaquant obtient $v := g^x \bmod p$ (et le sel) et x peut être récupéré avec une attaque par dictionnaire si le mot de passe est faible (l'attaquant calcule des candidats x' pour x jusqu'à ce que $g^{x'} \bmod p$ soit à égal à v). Une alternative serait de résoudre le problème du logarithme discret qui est un problème difficile lorsque les paramètres du groupe sont sûrs (le record actuel est un logarithme discret dans un corps fini de 795 bits [Bou+20]).

Du côté du client, la valeur x est reconstruite lorsque le client entre son mot de passe, et est suivi par le calcul de l'exponentiation $g^x \bmod p$. Par conséquent, cette opération est susceptible d'être réalisée plusieurs fois et est le sujet de l'attaque présentée dans la section suivante.

7.3 Attaque par dictionnaire sur SRP

Dans cette section on présente une attaque sur le protocole SRP. L'objectif est de monter une attaque par dictionnaire en mode hors ligne en utilisant une connaissance partielle de l'exposant secret, obtenue d'une exponentiation vulnérable.

7.3.1 Modèle d'attaque

La cible de l'attaque est l'implémentation du protocole côté client concernant l'exponentiation avec l'exposant secret x qui est dérivé du mot de passe. L'attaque est constituée de trois étapes :

1. Récupérer le sel et l'identifiant du client (une fois pour chaque sel) ;
2. Observer l'implémentation vulnérable par canaux auxiliaires (cette étape peut nécessiter d'être répétée selon la vulnérabilité) ;
3. Lancer une attaque par dictionnaire à partir des données obtenues.

La première étape est nécessaire car ce sont les valeurs en entrées de la dérivation de l'exposant en dehors du mot de passe. La seconde permet à l'attaquant de récupérer des données relatives à l'exposant x qui sont fuitées par l'implémentation.

Lorsque ces conditions sont réunies, l'attaquant crée une valeur distinctive de l'exposant secret à partir des fuites observées. Ensuite, à partir du sel et l'identifiant du client précédemment récupérés, l'attaque par dictionnaire peut être réalisée et est résumée dans l'algorithme 21. La valeur distinctive agit comme indicateur pour retirer les mauvais mots de passe. En effet, cette valeur est dérivée du bon exposant, donc le bon mot de passe satisfait ce critère. Cependant, cela ne garantit par qu'il sera trouvé, et à moins d'une grande précision, de nombreux autres mots de passe peuvent passer le test de sélection.

Algorithme 21 : Filtration des mots de passe

Entrée : salt, id, dictionnaire, distinguisher
Sortie : Liste de mots de passe candidats

```
liste ← {}
pour pw' ∈ dictionnaire faire
    x' ← H(salt, id, pw')
    si check(x', distinguisher) alors
        liste ← list ∪ {pw'}
retourner liste
```

7.3.2 Attaquant passif

Dans cette version, l'attaquant ne fait qu'observer l'exécution du côté du client. À partir de l'observation d'une ou plusieurs exécutions, une seule valeur distinctive est créée. La filtration des mots de passe est limitée lorsque la fuite est identique à chaque exécution. Le problème principal est l'impossibilité d'améliorer la précision obtenue par la fuite avec d'autres observations. Par contre, cela reste intéressant lorsque chaque nouvelle exécution fuite davantage de données, ce qui est le cas notamment de la vulnérabilité liée à la scission Euclidienne du chapitre 5.

Une possibilité pour rendre cette version plus efficace est de comparer les listes de mots de passe candidats obtenues pour deux utilisateurs. Si le nombre de mots de passe candidats dans chaque liste est suffisant bas, seuls quelques mauvais mots de passe devraient apparaître sur les deux listes. Ainsi, un mot de passe commun peut être trouvé dans l'intersection des listes.

7.3.3 Attaquant actif

L'attaquant précédent est limité à une seule valeur distinctive et sa précision. Dans l'attaque Dragonblood [VR20], il a été remarqué que la modification d'une adresse MAC permet l'acquisition de nouvelles mesures liées à un mot de passe identique : de nouvelles valeurs distinctives peuvent être créées. La même idée est applicable avec le protocole SRP, en se servant du sel qui influence la sortie de la fonction de hachage. Une modification de cette valeur (par exemple avec un *Man in the Middle*) donne lieu à un nouvel exposant dérivé du même mot de passe. Une nouvelle valeur distinctive est créée à partir des fuites observées par canaux auxiliaires, et la filtration de l'algorithme **21** peut être utilisée pour réduire davantage le nombre de candidats de façon itérative.

Cette variante signifie que la clé de session calculée par le client est incorrecte et peut se faire détecter après plusieurs tentatives. L'attaquant pourrait se faire passer pour le serveur et envoyer directement des valeurs choisies pour le sel. Bien que cela empêcherait une détection côté serveur, il y aurait toujours un échec d'authentification pour le client.

Une autre possibilité serait un service ou un produit qui change lui-même la valeur du sel même si le mot de passe est identique.

7.3.4 Considérations pratiques

Récupérer l'identifiant du client et le sel. L'identifiant du client (adresse email, nom d'utilisateur, numéro, etc) peut être relativement facile à deviner si la cible est connue. Quant au sel, il est à découvert si les communications sont envoyées en clair.

Cependant, les communications peuvent être chiffrées (par TLS), empêchant l'exposition du sel. Un contournement est possible si l'attaquant initialise l'authentification avec le serveur en se faisant passer pour le client. L'attaquant recevrait la valeur éphémère publique B ainsi que le sel qui correspond au client visé. Un exemple pour tester est ProtonMail dont l'authentification est basée sur SRP depuis la version 3.6 [PM16]. Les données du protocoles sont échangées au format JSON et peuvent être extraites avec les outils à dispositions dans les navigateurs web.

Une autre couche de sécurité peut être employée telle qu'un second facteur d'authentification avant la première phase du protocole SRP. Dans ce cas il devient plus difficile de se faire passer pour le client et obtenir le sel. Un exemple est donné dans la section **7.3.5** dans le cas du service de recouvrement de trousseau de clés d'Apple iCloud.

Observation par canaux auxiliaires. Une fuite observable à distance comme la mesure du temps d'exécution est difficile à obtenir dans ce protocole : les calculs qui mettent en jeu l'exposant secret ne sont qu'une part de la procédure, donc l'analyse du temps d'exécution pourrait être difficile voire impossible à exploiter. D'autres moyens d'attaques nécessitent un accès physique à l'appareil du client, que ce soit un téléphone ou un ordinateur. Cela limite la réalisation de l'attaque car l'observation fait suite à l'entrée d'un mot de passe sur l'appareil.

Néanmoins, les auteurs de [Gen+16] ont montré comment réaliser une attaque par canaux auxiliaire avec une sonde magnétique proche de l'appareil, ou une sonde d'alimentation sur le port USB d'un chargeur de téléphone.

On peut aussi citer les attaques par cache qui passent par un programme espion qui partage le cache de l'application comme dans l'attaque PARASITE [BFS21].

Man in the Middle. Le problème principal de la second variante de l'attaque est une communication chiffrée par TLS entre le client et le serveur qui rend plus difficile de modifier les valeurs transmises. Donc une attaque *Man in the Middle* nécessiterait de se faire passer pour le serveur. Une modification du sel a pour conséquence un échec d'authentification, et le client pourrait croire à une erreur de frappe de sa part. Il est nécessaire de réaliser une modification quelques fois seulement, l'attaquant peut espacer dans le temps l'attaque pour rester inaperçu.

Il y a d'autres possibilités pour rendre cette seconde variante possible telle qu'une injection de faute pour altérer la valeur du sel de l'identifiant sur l'appareil du client. Ces solutions ont également leurs difficultés ; le point important est que l'effet doit être maîtrisé pour que la valeur modifiée soit prévisible pour l'attaquant.

Efficacité de l'attaque par dictionnaire. Pour estimer l'efficacité de l'attaque, on illustre avec la fuite suivante : l'exponentiation révèle que l'exposant se situe dans un intervalle de largeur w. Plus cette valeur est faible et plus la filtration est efficace.

Cela s'explique par la construction de l'exposant dans le protocole SRP où il s'agit de la représentation en tant qu'entier de la sortie d'une fonction de hachage cryptographique. Soit ℓ la longueur en bits maximale possible de la fonction de hachage (par exemple ℓ vaut 256 pour SHA-256), alors il est attendu que les sorties soient uniformément distribuées dans l'intervalle $[0, 2^\ell - 1]$. Avec un sel et un identifiant fixe, parcourir tous les mots de passe dans un dictionnaire donne des exposants uniformément répartis, donc environ une proportion de $w/2^\ell$ d'entre eux devraient satisfaire le critère. Si D est la taille du dictionnaire, alors le nombre de mots de passe candidats est proche de

$$D \cdot \frac{w}{2^\ell}.$$

La vulnérabilité présentée dans le chapitre 5 qui fut présente dans la bibliothèque cryptographique d'Apple correspond à cet exemple, avec la particularité que la largeur w de l'intervalle diminue en fonction du nombre d'observations faites sur l'exponentiation. Cela rend la première variante de l'attaque intéressante en pratique pour réduire le nombre de mots de passe candidats.

Cependant, l'utilisation de plusieurs valeurs distinctives dans la seconde variante rend la filtration plus efficace. En effet, une altération du sel signifie qu'un nouvel exposant et une nouvelle valeur distinctive sont acquis, et que ceux-ci sont indépendants des valeurs originales grâce à la fonction de hachage. La liste des mots de passe candidats est ainsi réduite significativement par chaque nouvelle valeur distinctive de largeur w_i :

$$D \cdot \prod_i \frac{w_i}{2^\ell}.$$

En particulier, la situation où deux utilisateurs ont le même mot de passe est équivalente à la possession de deux valeurs distinctives de largeurs w_1 et w_2.

Si la valeur distinctive est la longueur en bits n de l'exposant, alors celui-ci se situe dans un intervalle de largeur $w = 2^{n-1}$. Par conséquent, environ $1/2^{\ell-n+1}$ des mots de passe du dictionnaire correspondent, et, dans le pire cas, la moitié des mots de passe seront éliminés. Quelques dizaines de valeurs distinctives de la longueur en bits de l'exposant sont généralement suffisants (en fonction de la taille du dictionnaire). On peut se référer

aux articles sur l'attaque Dragonblood [BFS20, VR20] pour ce cas particulier : bien que ce ne soit pas lié à la longueur en bits de l'exposant, la fuite suit elle aussi une distribution géométrique.

Remarque 11. La fiabilité de la valeur distinctive est importante. Par exemple, il n'est pas garanti que l'exposant se situe dans l'intervalle obtenu après l'analyse de la fuite dans l'exponentiation modulaire de la bibliothèque cryptographique d'Apple, ce qui éliminerait le bon mot de passe.

7.3.5 iCloud Keychain Recovery

L'écosystème iCloud est au cœur des services en ligne d'Apple et peut être partagé à travers plusieurs appareils liés au même compte, et le protocole SRP est employé en plusieurs endroits. Un des services est *iCloud Keychain Recovery* qui permet aux utilisateurs de séquestrer leur trousseau de clés avec Apple (qui contient des données sensibles comme des mots de passe personnels, ou bien le numéro de carte de crédit). Ce service a une couche supplémentaire de protection par le protocole SRP (avec le groupe de 2048 bits de la RFC 5054, SHA-256 comme fonction de hachage et des sels de 64 octets) avec un mot de passé distinct de celui du compte iCloud : pour chaque appareil, iCloud conserve une sauvegarde du trousseau protégé par le mot de passe de l'appareil (comportement par défaut si le compte iCloud a le second facteur d'authentification activé). Ainsi, le trousseau est sécurisé si le compte iCloud venait à être compromis, et Apple n'a pas accès au contenu.

Lorsque l'utilisateur déconnecte puis reconnecte son compte iCloud sur son appareil, les requêtes HTTP suivantes ont lieu :

1. Une requête `get_records` est transmise et le serveur répond avec une liste des entrées sauvegardées pour ce compte iCloud ;
2. Une requête `srp_init` initialise la première phase du protocole SRP (contenant la valeur éphémère du client) en indiquant quelle entrée est souhaitée, et le serveur répond avec les données correspondantes : `DSID` (numéro unique du compte iCloud, utilisé comme identifiant dans SRP), la valeur éphémère du serveur, ainsi que le sel ;
3. Une requête `recover` pour la seconde phase avec la preuve client, et le serveur répond avec sa preuve ainsi que la sauvegarde ;
4. Le client envoie une requête `enroll` avec une nouvelle sauvegarde protégée par le même mot de passe, mais un sel différent.

Deux situations ont pu être observée où la sauvegarde récupérée est soit l'ancienne associée à l'appareil, soit celle d'un autre appareil associé au même compte. Le premier cas est intéressant pour la variante de l'attaque avec plusieurs valeurs distinctives.

Par contre une des contraintes de l'attaque est la récupération du sel et de l'identifiant. Ce souci peut être surmonté par la surveillance des communications chiffrées entre la cible et les serveur d'Apple. Cependant des tentatives de mises en place d'un serveur proxy ont échoué sur un MacBook Pro. Une alternative est de lancer la requête `srp_init`, mais un jeton d'authentification est nécessaire pour les comptes iCloud avec le second facteur d'authentification activé (ce qui est devenu obligatoire pour tout nouveau compte et ne peut être désactivé). Ce jeton est obtenu après une authentification réussie du compte iCloud.

Dans le cas où un attaquant a déjà compromis ce compte et a réussi à passer le second facteur d'authentification une fois, alors son appareil est enregistré comme appareil de confiance. En faisant cela il se peut que le trousseau se synchronise directement avec les autres appareils déjà enregistrés si l'option est cochée dans les préférences. Cette synchronisation n'utilise pas SRP, donc cela rend l'attaque inutile si le but est de récupérer le trousseau, mais elle peut toujours être utilisée pour obtenir le mot de passe de l'appareil de l'utilisateur ciblé.

L'attaque pour récupérer un mot de passe faible sur un appareil Apple peut être réalisée sous les hypothèses suivantes :

- Compromettre le mot de passe du compte iCloud de l'utilisateur ciblé ;
- Passer le second facteur d'authentification au moins une fois ;
- Forcer l'utilisateur à se déconnecter du compte iCloud sur leur appareil plusieurs fois ;
- Observer par canaux auxiliaires la fuite durant l'exécution du protocole SRP dans le service *iCloud Keychain Recovery* lorsque l'utilisateur se reconnecte.

Chaque fois qu'une nouvelle fuite et un sel sont récupérés, l'attaque par dictionnaire peut être appliquée telle qu'elle est donnée dans l'algorithme **21**.

Requêtes et réponses. On donne ci-dessous un extrait des requêtes et réponses HTTP. Les captures ont été obtenues sur un second appareil avec Wireshark [1], et Frida [2] pour exporter les clés de sessions TLS pour déchiffrer.

```
POST /escrowproxy/api/srp_init HTTP/1.1
Host: p49-escrowproxy.icloud.com:443
(...)
Authorization: Basic Y29 (...) BFVA==
(...)

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
    PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
 <key>blob</key>
   <string>pIC8heH+SbClHjnugsfBBc (...) +2+CM8q8hItheOscqwA==</string>
 <key>command</key>
   <string>SRP_INIT</string>
 <key>label</key>
   <string>com.apple.icdp.record.et3n (...) 8HqM</string>
 (...)
</dict>
</plist>
```

Listing 7.1 : Phase d'initialisation du protocole SRP côté client dans iCloud Keychain Recovery.

Dans le listing **7.1**, la valeur éphémère A est encodé en base 64 et correspond à 256 octets, cohérent avec la taille du groupe de 2048 bits. La réponse du serveur est donnée dans la figure **7.2**, et contient le sel et la valeur éphémère B à la fin du bloc de données `respBlob` en base64, chacun précédé par leur longueur en octets : 64 and 256.

Il a été confirmé sur le premier appareil que le protocole SRP est exécuté pour récupérer le même enregistrement et le sel après une reconnexion. Le nouvel enregistrement créé pour le premier appareil est celui utilisé dans le second quand l'expérience est répétée.

1. https://www.wireshark.org
2. https://frida.re

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
    PropertyList-1.0.dtd">
<plist version="1.0">
 <dict>
  <key>status</key>
    <string>0</string>
  <key>message</key>
    <string>Success</string>
  <key>version</key>
    <integer>1</integer>
  <key>dsid</key>
    <string>28 (...)</string>
  <key>ClubTypeID</key>
    <integer>0</integer>
  <key>respBlob</key>
    <string>AAABiAAAAKQAAAAAPbZQrX (...) jsxg48nknPybRNHkTM=</string>
 </dict>
</plist>
```

Listing 7.2 : Phase d'initialisation du protocole SRP côté serveur dans iCloud Keychain Recovery.

7.3.6 Expérimentation

Le cas où plusieurs valeurs distinctives donnant la longueur en bits de l'exposant a déjà été étudié dans [BFS20, VR20], donc on se concentre ici sur l'indicateur donné par la vulnérabilité liée à la scission Euclidienne de l'exposant telle qu'implémentée par Apple exposée dans le chapitre 5. Celle-ci a l'avantage que l'indicateur devient plus précis lorsqu'on observe de plus en plus d'exponentiations avec le même secret.

L'expérimentation a été menée avec les conditions suivantes :

- SHA-256 comme fonction de hachage (les exposants sont dans l'intervalle $[0, 2^{256}-1]$) ;
- Le groupe de 2048 bits dont les paramètres sont dans la RFC 5054 ;
- Un sel de 16 octets ;
- Un mot de passe composé de 6 chiffres.

Étant donnés un sel et mot de passe aléatoire, l'exposant secret a été dérivé selon le protocole SRP (avec « id » comme identité du client). La fuite a été simulée pour $N = 1000$ exponentiations telle que décrite par le code source d'Apple, puis une approximation a été réalisée sous la forme $[x_{\min}, x_{\max}]$ par l'intervalle de Wilson avec un niveau de confiance de 95 %. Tous les mots de passe dont l'exposant se situe dans cet intervalle sont gardés comme candidats.

Cela a été répété pour 10000 mots de passe différents, et les exposants, nombre de mots de passe candidats et si le bon mot de passe est inclus dans les candidats ont été conservés. Les résultats sont donnés dans la figure 7.2.

On rappelle ici que la vulnérabilité n'est pas observable pour les exposants de taille impaire en bits et donc le nombre de mots de passe candidats n'apparaît pas (sauf pour les exposants de longueur 251 bits ou moins représentés dans les points isolés sur la gauche de la figure).

Au total, il y a seulement 6846 cas avec moins de 32000 mots de passe candidats, et le bon mot de passe est inclus dans la liste dans 95,5 % des cas, ce qui est cohérent avec le niveau de confiance sur l'intervalle calculé.

On note au passage la remarque faite dans la section 5.2.3 où les exposants proche

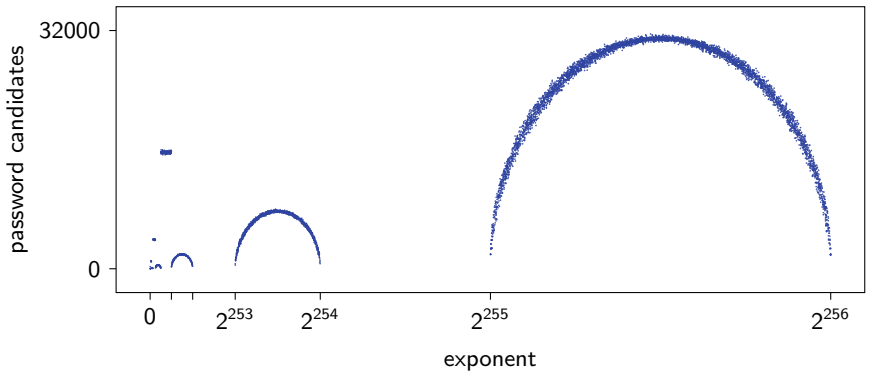


Figure 7.2 : Nombre de mots de passe candidats pour $N = 1000$ mesures en fonction de l'exposant.

d'une puissance de 2 sont plus faciles à approximer et par conséquent l'attaque par dictionnaire est plus efficace.

Le meilleur résultat est un mot de passe dont l'exposant correspondant vaut environ $2^{243,17}$ et fut inclus dans une liste de 8 candidats.

7.4 Le protocole SPAKE2+

Ce protocole a des propriétés similaires à SRP et a été introduit dans [CKS09]. Une différence majeure est qu'il est compatible avec les courbes elliptiques, ce qui n'est pas possible pour SRP.

Il est présenté ci-dessous avec les courbes elliptiques comme groupe où les calculs sont effectués.

7.4.1 Description

Étant donné un mot de passe **pw** et des données liées à l'identité des deux entités, deux valeurs secrètes w_0 et w_1 sont créées par une fonction de dérivation simplement notée H ici. Un point de base P et deux points additionnels M et N sur une courbe elliptique font parties des paramètres publics. La première entité est un client qui connaît w_0 et w_1, qu'il calcule à partir d'un mot de passe, tandis que la seconde entité est un serveur qui n'a besoin que de w_0 et $L := [w_1]P$.

Les deux entités calculent des points publics X et Y qu'il s'échangent, créés à partir de valeurs éphémères et w_0, puis calculent un secret partagé de manière similaire à un échange de clé Diffie-Hellman classique.

Les différentes étapes sont résumées dans la figure 7.3.

7.4.2 Sécurité

Tout comme dans le protocole SRP, l'espionnage des communications ne peuvent permettre d'obtenir le mot de passe, et une compromission du serveur ne révèle que w_0 et L. Si le mot de passe a une entropie élevé, alors w_1 ne peut être retrouvé, et un attaquant ne peut pas se faire passer pour un client vu qu'il serait nécessaire de calculer

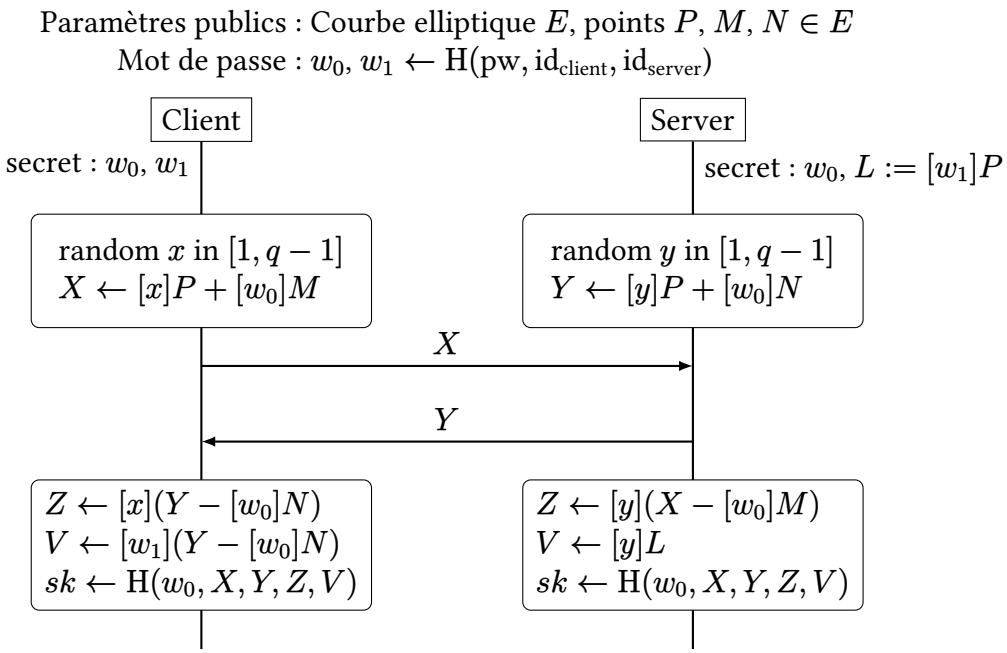


Figure 7.3 : Le protocole SPAKE2+.

$[w_1 y]P$ à partir de $Y - [w_0]N = [y]P$ et $L = [w_1]P$. Il s'agit du problème Diffie-Hellman calculatoire (CDH) supposé difficile.

7.4.3 Présence dans CoreCrypto

Dans la bibliothèque cryptographique d'Apple, les multiplications scalaires sont calculées individuellement avec la fonction `ccec_mult`.[3]

Du côté du client, ces calculs sont :

$$\begin{cases} S = [x]P \\ \mathbf{T} = [\mathbf{w_0}]\mathbf{M} \\ X = S + T \end{cases} \qquad \begin{cases} \mathbf{S'} = [\mathbf{w_0}]\mathbf{N} \\ T' = Y - S' \\ Z = [x]T' \\ \mathbf{V} = [\mathbf{w_1}]\mathbf{T'}. \end{cases}$$

Les multiplications scalaires en gras sont réalisées avec un scalaire fixe (pour un mot de passe et identités des entités identiques).

Du côté du serveur, on a :

$$\begin{cases} S = [y]P \\ \mathbf{T} = [\mathbf{w_0}]\mathbf{N} \\ Y = S + T \end{cases} \qquad \begin{cases} \mathbf{S'} = [\mathbf{w_0}]\mathbf{M} \\ T' = X - S' \\ Z = [y]T' \\ V = [y]L. \end{cases}$$

Une multiplication scalaire vulnérable, telle que celle présentée dans le chapitre 5, fait fuiter de l'information à la fois sur w_0 et w_1 du côté du client. L'attaque par dictionnaire

3. Avant les correctifs appliqués en 2021, voir le chapitre 5.

décrite dans la section 7.3 est alors applicable mais avec deux critères pour filtrer les mots de passe. De plus, d'après la description du protocole, les identités du client et du serveur font partie des valeurs pour la dérivation du mot de passe, alors une modification malveillante peut amener à l'obtention de nouveaux critères et améliorer la filtration.

7.5 Conclusion

Dans ce chapitre on a présenté une attaque contre les protocoles SRP et SPAKE2+ lorsque l'exponentiation modulaire ou la multiplication scalaire est vulnérable à une attaque par canaux auxiliaires. Ainsi, par la fuite observée il est possible de construire un indicateur pour filtrer les mots de passe.

Les contraintes pour réaliser l'attaque sur le protocole SRP ont été présentées et un cas pratique a été étudié. Celui-ci est l'utilisation de SRP comme authentification supplémentaire sur le service *iCloud Keychain Recovery* d'Apple. Une expérimentation a été réalisée en reprenant la vulnérabilité du chapitre 5 afin d'illustrer le concept de l'attaque.

Conclusion

Dans cette thèse, les travaux qui ont été présentés concernent essentiellement les soucis d'implémentations de courbes elliptiques en vue de leur usage en cryptographie. Cela s'est manifesté par une analyse en premier lieu des différentes formules, algorithmes et courbes elliptiques utilisées en pratique, ce qui a abouti à la présentation des cas exceptionnels dans leur utilisation. Plus précisément, on a montré pour plusieurs formules les situations où celles-ci ne donnent pas le bon résultat, puis si ces cas particuliers pouvaient apparaître en fonction de l'algorithme de multiplication scalaire.

La gestion de ces exceptions est importante pour garantir une exécution résistante à des attaquants plus ou moins fort. En premier exemple on montre les conséquences de l'utilisation de calculs factices par l'introduction d'additions de points inutiles pour rendre une exécution régulière. Plusieurs implémentations reposent sur cette technique et celles-ci peuvent être visées par une attaque *Safe-Error* qui, par l'injection d'une perturbation, permet de détecter la présence de ces calculs factices. Ce type d'attaque a déjà établi par le passé et on a identifié plusieurs implémentations qui y sont vulnérables, accompagné de simulations pour montrer le concept de l'attaque. Une nouvelle proposition a été faite pour identifier ces calculs factices dans le cas spécifique d'une implémentation où ils sont réalisés essentiellement avec des opérandes nuls. En effet, le faible poids de Hamming de ces valeurs est suffisamment visible pour apparaître sur une trace de consommation de courant, ce qui révèle leur position ainsi qu'un morceau assez large pour être exploité.

Dans la suite, on a montré qu'une méthode de protection basée sur une division Euclidienne pour randomiser les calculs introduit un biais dans l'exécution. Cette légère variation permet de réaliser une approximation du scalaire secret manipulé, à condition que cette valeur soit fixe et que plusieurs observations peuvent être faites. Un exemple d'application peut se trouver dans les protocoles d'authentification par mot de passe tels que SRP (pour l'exponentiation modulaire) ou SPAKE2+ (compatible avec les courbes elliptiques). Le mot de passe est utilisé pour générer un exposant ou scalaire, donc une approximation ou d'autres types de fuites peuvent être utilisées comme indicateurs pour appliquer une attaque par dictionnaire.

Cette méthode de randomisation est notamment utilisée dans la bibliothèque cryptographique présente dans les appareils Apple, ce qui a conduit à des communications avec les ingénieurs afin d'échanger sur ces découvertes. Les correctifs proposés ont été analysés et présentés, en particulier les modifications apportées à l'algorithme de multiplication scalaire.

Enfin, une dernière contribution de cette thèse est l'étude d'une attaque par injection de faute DFA. Une partie de ces travaux poursuivent l'étude des techniques de randomisation comme celle basée sur la scission Euclidienne. Des stratégies pour exploiter une faute

dans ces contextes sont présentées ce qui montre qu'elles n'apportent pas nécessairement une protection contre ce type d'attaquants. Aussi, on a proposé une nouvelle attaque DFA sur l'algorithme Montgomery ladder qui a l'avantage de contourner certaines protections comme la validation de point, voire une vérification de l'invariant lorsque des formules spécifiques n'utilisant par toutes les coordonnées sont utilisées.

A

Invariant et formules XYcoZ

Dans l'expérimentation du chapitre 6, un exécutable a été créé qui effectue une multiplication scalaire avec les formules XYcoZ et qui intègre une vérification de l'invariant. Pour cela il est nécessaire dans un premier temps de reconstruire la coordonnée Z des points R_0 et R_1 manipulés pendant l'exécution et de calculer l'invariant. Ceci a été réalisé avec l'introduction des formules XYcoZ-SUB.

A.1 Formules XYcoZ-SUB

L'addition avec mise à jour XYcoZ-ADD calcule à la fois l'addition $P_1 + P_2$ et met à jour P_1 de façon à ce que les deux points en sortie partagent la même coordonnée Z. Une légère modification de cette addition permet de calculer la différence $P_2 - P_1$. Soit $P_1 = (X_1 : Y_1 : Z)$ et $P_2 = (X_2 : Y_2 : Z)$ deux points avec Z non nul, alors $P_2 - P_1 = (X_4 : Y_4 : Z_4)$ et $P_1' = (X_1' : Y_1' : Z_1')$ la mise de jour de P_1 sont calculés par les formules XYcoZ-SUB suivantes :

$$\begin{cases} X_4 = (Y_2 + Y_1)^2 - (X_2 - X_1)^2(X_1 + X_2) \\ Y_4 = (Y_2 + Y_1)(X_1(X_2 - X_1)^2 - X_4) + Y_1(X_2 - X_1)^3 \\ X_1' = X_1(X_2 - X_1)^2 \\ Y_1' = Y_1(X_2 - X_1)^3 \\ Z_4 = Z_1' = (X_2 - X_1)Z. \end{cases}$$

A.2 Modification de l'algorithme

Comme à la fin de l'exécution de l'algorithme Montgomery ladder on a $(R_0, R_1) = ([k]P, [k+1]P)$, alors l'application de la fonction XYcoZ-SUB aboutit au couple $(P, [k]P)$.

L'invariant est disponible et l'application de la formule (3.1.3) permet de récupérer la coordonnée Z commune à la sortie de l'algorithme $Q = [k]P$ et de l'invariant calculé. Une fois sous forme affine, l'invariant calculé peut être comparé avec l'invariant P donné en entrée.

La multiplication scalaire avec les formules XYcoZ et qui inclut la vérification de l'invariant avec l'utilisation de la formule XYcoZ-SUB est donnée dans l'algorithme 22.

Algorithme 22 : Montgomery ladder avec les formules XYcoZ et vérification de l'invariant

Entrée : P, $k = (k_{n-1}, \ldots, k_0)_2$, $1 < k < q-1$ et $k_{n-1} = 1$
Sortie : $[k]P$

1: $(R_0, R_1) \leftarrow$ XYcoZ-DBL(P) ▷ $(R_0, R_1) = ([2]P, P)$
2: pbit $\leftarrow 1$
3: **pour** $i = n-2$ **à** 0 **faire**
4: pbit $\leftarrow$ pbit $\oplus\, k_i$
5: conditional_swap(pbit, R_0, R_1)
6: $(R_0, R_1) \leftarrow$ XYcoZ-ADDC(R_0, R_1) ▷ $(R_0, R_1) \leftarrow (R_0 + R_1, R_0 - R_1)$
7: $(R_0, R_1) \leftarrow$ XYcoZ-ADD(R_0, R_1) ▷ $(R_0, R_1) \leftarrow (R_0 + R_1, R_0')$
8: pbit $\leftarrow k_i$
9: conditional_swap(pbit, R_0, R_1)
10: $(R_0, R_1) \leftarrow$ XYcoZ-SUB(R_0, R_1) ▷ $(R_0, R_1) \leftarrow (R_1 - R_0, R_0')$
11: $\lambda \leftarrow y(P) \cdot X(R_0)/(Y(R_0) \cdot x(P))$ ▷ $\lambda = Z^{-1}$
12: $Q \leftarrow (\lambda^2 \cdot X(R_1), \lambda^3 \cdot Y(R_1))$
13: $I \leftarrow (\lambda^2 \cdot X(R_0), \lambda^3 \cdot Y(R_0))$
14: **retourner** $Q \oplus I \oplus P$

Bibliographie

[AH21] Martin R. Albrecht and Nadia Heninger, “On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem”, in: *EUROCRYPT 2021*, ed. by Anne Canteaut and François-Xavier Standaert, vol. 12696, LNCS, Springer, 2021, pp. 528–558, DOI: 10.1007/978-3-030-77870-5_19.

[ANS20] ANSSI, *Guide des mécanismes cryptographiques – Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques, version 2.04*, voir www.ssi.gouv.fr, 2020.

[Ara+14] Diego F. Aranha et al., “GLV/GLS Decomposition, Power Analysis, and Attacks on ECDSA Signatures with Single-Bit Nonce Bias”, in: *ASIACRYPT 2014*, ed. by Palash Sarkar and Tetsu Iwata, vol. 8873, LNCS, Springer, 2014, pp. 262–281, DOI: 10.1007/978-3-662-45611-8_14.

[Ara+20] Diego F. Aranha et al., “LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage”, in: *CCS ’20: 2020 ACM SIGSAC*, ed. by Jay Ligatti et al., ACM, 2020, pp. 225–242, DOI: 10.1145/3372297.3417268.

[Bar+16] Alessandro Barenghi et al., “A Fault-Based Secret Key Retrieval Method for ECDSA: Analysis and Countermeasure”, in: *ACM J. Emerg. Technol. Comput. Syst.* 13.1 (2016), 8:1–8:26, DOI: 10.1145/2767132.

[BDL97] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton, “On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)”, in: *EUROCRYPT ’97*, ed. by Walter Fumy, vol. 1233, LNCS, Springer, 1997, pp. 37–51, DOI: 10.1007/3-540-69053-0_4.

[Ber06] Daniel J. Bernstein, “Curve25519: New Diffie-Hellman Speed Records”, in: *PKC 2006*, ed. by Moti Yung et al., vol. 3958, LNCS, Springer, 2006, pp. 207–228, DOI: 10.1007/11745853_14.

[BFS20] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt, “Dragonblood is Still Leaking: Practical Cache-based Side-Channel in the Wild”, in: *ACSAC ’20*, ACM, 2020, pp. 291–303, DOI: 10.1145/3427228.3427295.

[BFS21] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt, “PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild”, in: *CCS ’21: 2021 ACM SIGSAC*, ed. by Yongdae Kim et al., ACM, 2021, pp. 2497–2512, DOI: 10.1145/3460120.3484563.

[BG15] Johannes Blömer and Peter Günther, “Singular Curve Point Decompression Attack”, in: *FDTC 2015*, ed. by Naofumi Homma and Victor Lomné, IEEE Computer Society, 2015, pp. 71–84, DOI: 10.1109/FDTC.2015.17.

[BJ02] Eric Brier and Marc Joye, "Weierstraß Elliptic Curves and Side-Channel Attacks", in: *PKC 2002*, ed. by David Naccache and Pascal Paillier, vol. 2274, LNCS, Springer, 2002, pp. 335–345, DOI: 10.1007/3-540-45664-3_24.

[BL07] Daniel J. Bernstein and Tanja Lange, "Faster Addition and Doubling on Elliptic Curves", in: *ASIACRYPT 2007*, ed. by Kaoru Kurosawa, vol. 4833, LNCS, Springer, 2007, pp. 29–50, DOI: 10.1007/978-3-540-76900-2_3.

[Ble00] Daniel Bleichenbacher, "On the generation of one-time keys in DL signature schemes", in: *Presentation at IEEE P1363 working group meeting*, 2000, p. 81.

[BMM00] Ingrid Biehl, Bernd Meyer, and Volker Müller, "Differential Fault Attacks on Elliptic Curve Cryptosystems", in: *CRYPTO 2000*, ed. by Mihir Bellare, vol. 1880, LNCS, Springer, 2000, pp. 131–146, DOI: 10.1007/3-540-44598-6_8.

[Boo51] Andrew D. Booth, "A signed binary multiplication technique", in: *The Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (Jan. 1951), pp. 236–240, ISSN: 0033-5614, DOI: 10.1093/qjmam/4.2.236.

[BOS06] Johannes Blömer, Martin Otto, and Jean-Pierre Seifert, "Sign Change Fault Attacks on Elliptic Curve Cryptosystems", in: *FDTC 2006*, ed. by Luca Breveglieri et al., vol. 4236, LNCS, Springer, 2006, pp. 36–52, DOI: 10.1007/11889700_4.

[Bou+20] Fabrice Boudot et al., "Comparing the Difficulty of Factorization and Discrete Logarithm: A 240-Digit Experiment", in: *CRYPTO 2020*, ed. by Daniele Micciancio and Thomas Ristenpart, vol. 12171, LNCS, Springer, 2020, pp. 62–91, DOI: 10.1007/978-3-030-56880-1_3.

[BS97] Eli Biham and Adi Shamir, "Differential Fault Analysis of Secret Key Cryptosystems", in: *CRYPTO '97*, ed. by Burton S. Kaliski Jr., vol. 1294, LNCS, Springer, 1997, pp. 513–525, DOI: 10.1007/BFb0052259.

[BT11] Billy Bob Brumley and Nicola Tuveri, "Remote Timing Attacks Are Still Practical", in: *ESORICS 2011*, ed. by Vijay Atluri and Claudia Díaz, vol. 6879, LNCS, Springer, 2011, pp. 355–371, DOI: 10.1007/978-3-642-23822-2_20.

[BV96] Dan Boneh and Ramarathnam Venkatesan, "Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes", in: *CRYPTO '96*, ed. by Neal Koblitz, vol. 1109, LNCS, Springer, 1996, pp. 129–142, DOI: 10.1007/3-540-68697-5_11.

[CJ01] Christophe Clavier and Marc Joye, "Universal Exponentiation Algorithm", in: *CHES 2001*, ed. by Çetin Kaya Koç, David Naccache, and Christof Paar, vol. 2162, LNCS Generators, Springer, 2001, pp. 300–308, DOI: 10.1007/3-540-44709-1_25.

[CJ03] Mathieu Ciet and Marc Joye, "(Virtually) Free Randomization Techniques for Elliptic Curve Cryptography", in: *ICICS 2003*, ed. by Sihan Qing, Dieter Gollmann, and Jianying Zhou, vol. 2836, LNCS, Springer, 2003, pp. 348–359, DOI: 10.1007/978-3-540-39927-8_32.

[CKS09] David Cash, Eike Kiltz, and Victor Shoup, "The Twin Diffie-Hellman Problem and Applications", in: *J. Cryptol.* 22.4 (2009), pp. 470–504, DOI: 10.1007/s00145-009-9041-6.

[Coh+05] Henri Cohen et al., eds., *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Chapman and Hall/CRC, 2005, ISBN: 978-1-58488-518-4, DOI: 10.1201/9781420034981.

[Cor99] Jean-Sébastien Coron, "Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems", in: *CHES'99*, ed. by Çetin Kaya Koç and Christof Paar, vol. 1717, LNCS, Springer, 1999, pp. 292–302, DOI: 10.1007/3-540-48059-5_25.

[DH76] Whitfield Diffie and Martin E. Hellman, "New directions in cryptography", in: *IEEE Trans. Inf. Theory* 22.6 (1976), pp. 644–654, DOI: 10.1109/TIT.1976.1055638.

[DHA11] Agustin Dominguez-Oviedo, M. Anwar Hasan, and Bijan Ansari, "Fault-Based Attack on Montgomery's Ladder Algorithm", in: *J. Cryptol.* 24.2 (2011), pp. 346–374, DOI: 10.1007/s00145-010-9087-5.

[DHB17] Jeremy Dubeuf, David Hély, and Vincent Beroulle, "Enhanced Elliptic Curve Scalar Multiplication Secure Against Side Channel Attacks and Safe Errors", in: *COSADE 2017*, ed. by Sylvain Guilley, vol. 10348, LNCS, Springer, 2017, pp. 65–82, DOI: 10.1007/978-3-319-64647-3_5.

[Edw07] H. Edwards, "A normal form for elliptic curves", in: *Bulletin of the American Mathematical Society* 44 (2007), pp. 393–422, DOI: 10.1090/S0273-0979-07-01153-6.

[EL09] Nevine Maurice Ebeid and Rob Lambert, "Securing the Elliptic Curve Montgomery Ladder against Fault Attacks", in: *FDTC 2009*, ed. by Luca Breveglieri et al., IEEE Computer Society, 2009, pp. 46–50, DOI: 10.1109/FDTC.2009.35.

[Fou+08] Pierre-Alain Fouque et al., "Fault Attack on Elliptic Curve Montgomery Ladder Implementation", in: *FDTC 2008*, ed. by Luca Breveglieri et al., IEEE Computer Society, 2008, pp. 92–98, DOI: 10.1109/FDTC.2008.15.

[Fou+16] Pierre-Alain Fouque et al., "Safe-Errors on SPA Protected Implementations with the Atomicity Technique", in: *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, ed. by Peter Y. A. Ryan, David Naccache, and Jean-Jacques Quisquater, vol. 9100, LNCS, Springer, 2016, pp. 479–493, DOI: 10.1007/978-3-662-49301-4_30.

[FPLLL21] The FPLLL development team, "fpylll, a Python wraper for the fplll lattice reduction library, Version: 0.5.6", Available at https://github.com/fplll/fpylll, 2021.

[FRV14] Benoit Feix, Mylène Roussellet, and Alexandre Venelli, "Side-Channel Analysis on Blinded Regular Scalar Multiplications", in: *INDOCRYPT 2014*, ed. by Willi Meier and Debdeep Mukhopadhyay, vol. 8885, LNCS, Springer, 2014, pp. 3–20, DOI: 10.1007/978-3-319-13039-2_1.

[Gau09] Pierrick Gaudry, "Index calculus for abelian varieties of small dimension and the elliptic curve discrete logarithm problem", in: *J. Symb. Comput.* 44.12 (2009), pp. 1690–1702, DOI: 10.1016/j.jsc.2008.08.005.

[Gen+16] Daniel Genkin et al., "ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels", in: *SIGSAC 2016*, ed. by Edgar R. Weippl et al., ACM, 2016, pp. 1626–1638, DOI: 10.1145/2976749.2978353.

[GHS02] Pierrick Gaudry, Florian Hess, and Nigel P. Smart, "Constructive and Destructive Facets of Weil Descent on Elliptic Curves", in: *J. Cryptol.* 15.1 (2002), pp. 19–46, DOI: 10.1007/s00145-001-0011-x.

[GJ21] Robert Granger and Antoine Joux, *Computing Discrete Logarithms*, Cryptology ePrint Archive, Report 2021/1140, 2021.

[GK15] Shay Gueron and Vlad Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes", in: *J. Cryptogr. Eng.* 5.2 (2015), pp. 141–151, DOI: 10.1007/s13389-014-0090-x.

[Gou+11] Raveen R. Goundar et al., “Scalar multiplication on Weierstraß elliptic curves from Co-Z arithmetic”, in: *J. Cryptogr. Eng.* 1.2 (2011), pp. 161–176, DOI: 10.1007/s13389-011-0012-0.

[Gou03] Louis Goubin, “A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems”, in: *PKC 2003*, ed. by Yvo Desmedt, vol. 2567, LNCS, Springer, 2003, pp. 199–210, DOI: 10.1007/3-540-36288-6_15.

[GRV16] Dahmun Goudarzi, Matthieu Rivain, and Damien Vergnaud, “Lattice Attacks Against Elliptic-Curve Signatures with Blinded Scalar Multiplication”, in: *SAC 2016*, ed. by Roberto Avanzi and Howard M. Heys, vol. 10532, LNCS, Springer, 2016, pp. 120–139, DOI: 10.1007/978-3-319-69453-5_7.

[GV10] Christophe Giraud and Vincent Verneuil, “Atomicity Improvement for Elliptic Curve Scalar Multiplication”, in: *CARDIS 2010*, ed. by Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, vol. 6035, LNCS, Springer, 2010, pp. 80–101, DOI: 10.1007/978-3-642-12510-2_7.

[HPB04] Mustapha Hedabou, Pierre Pinel, and Lucien Bénéteau, “A comb method to render ECC resistant against Side Channel Attacks”, in: *IACR Cryptol. ePrint Arch.* 2004 (2004), p. 342.

[IT02] Tetsuya Izu and Tsuyoshi Takagi, “A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks”, in: *PKC 2002*, ed. by David Naccache and Pascal Paillier, vol. 2274, LNCS, Springer, 2002, pp. 280–296, DOI: 10.1007/3-540-45664-3_20.

[Joy12] Marc Joye, “A Method for Preventing “Skipping” Attacks”, in: *2012 IEEE Symposium on Security and Privacy Workshops*, IEEE Computer Society, 2012, pp. 12–15, DOI: 10.1109/SPW.2012.14.

[JY02] Marc Joye and Sung-Ming Yen, “The Montgomery Powering Ladder”, in: *CHES 2002*, ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, vol. 2523, LNCS, Springer, 2002, pp. 291–302, DOI: 10.1007/3-540-36400-5_22.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun, “Differential Power Analysis”, in: *CRYPTO ’99*, ed. by Michael J. Wiener, vol. 1666, LNCS, Springer, 1999, pp. 388–397, DOI: 10.1007/3-540-48405-1_25.

[Kob87] Neal Koblitz, “Elliptic Curve Cryptosystems”, in: *Mathematics of computation* 48.177 (1987), pp. 203–209, DOI: 10.1090/S0025-5718-1987-0866109-5.

[LLF20] Antoine Loiseau, Maxime Lecomte, and Jacques J. A. Fournier, “Template Attacks against ECC: practical implementation against Curve25519”, in: *HOST 2020*, IEEE, 2020, pp. 13–22, DOI: 10.1109/HOST45689.2020.9300261.

[LLL82] Arjen K. Lenstra, Hendrik W. Jr. Lenstra, and László Lovász, “Factoring polynomials with rational coefficients”, in: *Mathematische Annalen* 261 (1982), pp. 515–534, ISSN: 0025–5831, DOI: 10.1007/BF01457454.

[Mer+21] Robert Merget et al., “Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)”, in: *USENIX Security 21*, USENIX Association, Aug. 2021, pp. 213–230, ISBN: 978-1-939133-24-3.

[Mil85] Victor S. Miller, “Use of Elliptic Curves in Cryptography”, in: *CRYPTO ’85*, ed. by Hugh C. Williams, vol. 218, LNCS, Springer, 1985, pp. 417–426, DOI: 10.1007/3-540-39799-X_31.

[Möl01] Bodo Möller, "Securing Elliptic Curve Point Multiplication against Side-Channel Attacks", in: *ISC 2001*, ed. by George I. Davida and Yair Frankel, vol. 2200, LNCS, Springer, 2001, pp. 324–334, DOI: 10.1007/3-540-45439-X_22.

[Mon87] Peter L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization", in: *Mathematics of Computation* 48 (1987), pp. 243–264, DOI: 10.1090/S0025-5718-1987-0866113-7.

[MOV93] Alfred Menezes, Tatsuaki Okamoto, and Scott A. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field", in: *IEEE Trans. Inf. Theory* 39.5 (1993), pp. 1639–1646, DOI: 10.1109/18.259647.

[MPP20] Gabrielle De Micheli, Rémi Piau, and Cécile Pierrot, "A Tale of Three Signatures: Practical Attack of ECDSA with wNAF", in: *AFRICACRYPT 2020*, ed. by Abderrahmane Nitaj and Amr M. Youssef, vol. 12174, LNCS, Springer, 2020, pp. 361–381, DOI: 10.1007/978-3-030-51938-4_18.

[Mul+14] Elke De Mulder et al., "Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA: extended version", in: *J. Cryptogr. Eng.* 4.1 (2014), pp. 33–45, DOI: 10.1007/s13389-014-0072-z.

[MV06] Frédéric Muller and Frédéric Valette, "High-Order Attacks Against the Exponent Splitting Protection", in: *PKC 2006*, ed. by Moti Yung et al., vol. 3958, LNCS, Springer, 2006, pp. 315–329, DOI: 10.1007/11745853_21.

[Nas+16] Erick Nascimento et al., "Attacking Embedded ECC Implementations Through cmov Side Channels", in: *SAC 2016*, ed. by Roberto Avanzi and Howard M. Heys, vol. 10532, LNCS, Springer, 2016, pp. 99–119, DOI: 10.1007/978-3-319-69453-5_6.

[Nat13] National Institute of Standards and Technology, *FIPS PUB 186-4 Digital Signature Standard (DSS)*, 2013.

[NS02] Phong Q. Nguyen and Igor E. Shparlinski, "The Insecurity of the Digital Signature Algorithm with Partially Known Nonces", in: *J. Cryptol.* 15.3 (2002), pp. 151–176, DOI: 10.1007/s00145-002-0021-3.

[NS03] Phong Q. Nguyen and Igor E. Shparlinski, "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces", in: *Des. Codes Cryptogr.* 30.2 (2003), pp. 201–217, DOI: 10.1023/A:1025436905711.

[OT03] Katsuyuki Okeya and Tsuyoshi Takagi, "The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks", in: *CT-RSA 2003*, ed. by Marc Joye, vol. 2612, LNCS, Springer, 2003, pp. 328–342, DOI: 10.1007/3-540-36563-X_23.

[PH78] Stephen C. Pohlig and Martin E. Hellman, "An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance (Corresp.)", in: *IEEE Trans. Inf. Theory* 24.1 (1978), pp. 106–110, DOI: 10.1109/TIT.1978.1055817.

[PM16] Proton Technologies AG, *ProtonMail v3.6 Release Notes*, 2016.

[Pol78] J. Pollard, "Monte Carlo methods for index computation ()", in: *Mathematics of Computation* 32 (1978), pp. 918–924.

[RCB16] Joost Renes, Craig Costello, and Lejla Batina, "Complete Addition Formulas for Prime Order Elliptic Curves", in: *EUROCRYPT 2016*, ed. by Marc Fischlin and Jean-Sébastien Coron, vol. 9665, LNCS, Springer, 2016, pp. 403–428, DOI: 10.1007/978-3-662-49890-3_16.

[Res09] Certicom Research, *Standards for Efficient Cryptography, SEC 1: Elliptic Curve Cyptography, version 2*, Disponible à l'adresse https://secg.org, 2009.

[RIL19] Thomas Roche, Laurent Imbert, and Victor Lomné, "Side-Channel Attacks on Blinded Scalar Multiplications Revisited", in: *CARDIS 2019*, ed. by Sonia Belaïd and Tim Güneysu, vol. 11833, LNCS, Springer, 2019, pp. 95–108, DOI: 10.1007/978-3-030-42068-0_6.

[Ron13] Franck Rondepierre, "Revisiting Atomic Patterns for Scalar Multiplications on Elliptic Curves", in: *CARDIS 2013*, ed. by Aurélien Francillon and Pankaj Rohatgi, vol. 8419, LNCS, Springer, 2013, pp. 171–186, DOI: 10.1007/978-3-319-08302-5_12.

[Rus20] Andy Russon, "Exploiting dummy codes in Elliptic Curve Cryptography implementations", in: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2020*, 2020, pp. 229–249.

[Rus21a] Andy Russon, "Return of ECC dummy point additions: Simple Power Analysis on efficient P-256 implementation", in: *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 2-4 2021*, 2021, pp. 367–375.

[Rus21b] Andy Russon, "Threat for the Secure Remote Password Protocol and a Leak in Apple's Cryptographic Library", in: *ACNS 2021*, ed. by Kazue Sako and Nils Ole Tippenhauer, vol. 12727, LNCS, Springer, 2021, pp. 49–75, DOI: 10.1007/978-3-030-78375-4_3.

[Rus21c] Andy Russon, "Differential Fault Attack on Montgomery Ladder and in the Presence of Scalar Randomization", in: *INDOCRYPT 2021, Proceedings*, ed. by Avishek Adhikari, Ralf Küsters, and Bart Preneel, vol. 13143, LNCS, Springer, 2021, pp. 287–310, DOI: 10.1007/978-3-030-92518-5_14.

[SA98] T. Satoh and Kiyomichi Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", in: *Commentarii Math. Univ. St. Pauli.* 47 (1998), https://ci.nii.ac.jp/naid/110007696197/en/, pp. 81–92, ISSN: 0010258X.

[SCIKIT11] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python", in: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[SE94] Claus-Peter Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems", in: *Math. Program.* 66 (1994), pp. 181–199, DOI: 10.1007/BF01581144.

[Sem98] Igor A. Semaev, "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p", in: *Math. Comput.* 67.221 (1998), pp. 353–356, DOI: 10.1090/S0025-5718-98-00887-4.

[SH08] Jörn-Marc Schmidt and Christoph Herbst, "A Practical Fault Attack on Square and Multiply", in: *FDTC 2008*, ed. by Luca Breveglieri et al., IEEE Computer Society, 2008, pp. 53–58, DOI: 10.1109/FDTC.2008.10.

[Sha71] Daniel Shanks, "Class number, a theory of factorization, and genera", in: *Proceedings of Symposia in Pure Mathematics*, ed. by Donald J. Lewis, vol. 20, 1971, DOI: 10.1090/pspum/020.

[SM09] Jörn-Marc Schmidt and Marcel Medwed, "A Fault Attack on ECDSA", in: *FDTC 2009*, ed. by Luca Breveglieri et al., IEEE Computer Society, 2009, pp. 93–99, DOI: 10.1109/FDTC.2009.38.

[SM10] Jörn-Marc Schmidt and Marcel Medwed, "Fault Attacks on the Montgomery Powering Ladder", in: *ICISC 2010*, ed. by Kyung Hyune Rhee and DaeHun Nyang, vol. 6829, LNCS, Springer, 2010, pp. 396–406, DOI: 10.1007/978-3-642-24209-0_26.

[Sma99] Nigel P. Smart, "The Discrete Logarithm Problem on Elliptic Curves of Trace One", in: *J. Cryptol.* 12.3 (1999), pp. 193–196, DOI: 10.1007/s001459900052.

[ST20] Massimiliano Sala and Daniele Taufer, *The group structure of elliptic curves over Z/NZ*, 2020, arXiv: 2010.15543 [math.NT].

[Sun+21] Chao Sun et al., "Guessing Bits: Improved Lattice Attacks on (EC)DSA", in: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 455, URL: https://eprint.iacr.org/2021/455.

[SW15] Werner Schindler and Andreas Wiemers, "Efficient Side-Channel Attacks on Scalar Blinding on Elliptic Curves with Special Structure", in: NIST Workshop on ECC Standards, 2015.

[Tay+07] David Taylor et al., "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", in: *RFC* 5054 (2007), pp. 1–24, DOI: 10.17487/RFC5054.

[TB02] Elena Trichina and Antonio Bellazza, "Implementation of Elliptic Curve Cyptography with Built-In Counter Measures against Side Channel Attacks", in: *CHES 2002*, ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, vol. 2523, LNCS, 2002, pp. 98–113, DOI: 10.1007/3-540-36400-5_9.

[TT19] Akira Takahashi and Mehdi Tibouchi, "Degenerate Fault Attacks on Elliptic Curve Parameters in OpenSSL", in: *EuroS&P 2019*, IEEE, 2019, pp. 371–386, DOI: 10.1109/EuroSP.2019.00035.

[VR20] Mathy Vanhoef and Eyal Ronen, "Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd", in: *SP 2020*, IEEE, 2020, pp. 517–533, DOI: 10.1109/SP40000.2020.00031.

[VS12] Ihor Vasyltsov and Gökay Saldamli, "Fault detection and a differential fault analysis countermeasure for the Montgomery power ladder in elliptic curve cryptography", in: *Math. Comput. Model.* 55.1-2 (2012), pp. 256–267, DOI: 10.1016/j.mcm.2011.06.017.

[Wil27] Edwin B. Wilson, "Probable Inference, the Law of Succession, and Statistical Inference", in: *Journal of the American Statistical Association* 22.158 (1927), pp. 209–212, DOI: 10.1080/01621459.1927.10502953.

[Wu00] Thomas Wu, "The SRP Authentication and Key Exchange System", in: *RFC* 2945 (2000), pp. 1–8, DOI: 10.17487/RFC2945.

[Wu98] Thomas D. Wu, "The Secure Remote Password Protocol", in: *NDSS 1998*, The Internet Society, 1998, URL: https://www.ndss-symposium.org/ndss1998/secure-remote-password-protocol/.

[Yen+01] Sung-Ming Yen et al., "A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack", in: *ICISC 2001*, ed. by Kwangjo Kim, vol. 2288, LNCS, Springer, 2001, pp. 414–427, DOI: 10.1007/3-540-45861-1_31.

[YJ00] Sung-Ming Yen and Marc Joye, “Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis”, in: *IEEE Trans. Computers* 49.9 (2000), pp. 967–970, DOI: 10.1109/12.869328.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Problème du logarithme discret sur des courbes elliptiques

Mots clés : courbes elliptiques, cryptographie, sécurité d'implémentations

Résumé : L'usage des courbes elliptiques en cryptographie s'est largement répandu pour assurer la sécurité des communications ou des transactions financières. Cela est dû notamment au fait que la sécurité repose sur la difficulté du problème du logarithme discret qui permet d'utiliser les courbes elliptiques avec des paramètres qui assurent une efficacité.

Dans cette thèse, nous abordons principalement l'aspect sécurité d'implémentations des courbes elliptiques. L'utilisation de données auxiliaires par le biais de fuites liées à l'exécution du code informatique peut remettre en cause la sécurité des protocoles. Nous analysons tout d'abord plusieurs formules et algorithmes couramment utilisés dans des implémentations pour montrer les difficultés qui font face à un développeur afin de réaliser une implémentation sécurisée. Nous montrons ensuite que certaines techniques de protection peuvent amener une vulnérabilité, dont l'une d'entre elles est nouvelle et a été signalée aux développeurs. Enfin, nous proposons également une nouvelle attaque par injection de faute sur un algorithme et nous montrons également que certaines méthodes de protection basées sur l'introduction d'une randomisation de la valeur secrète n'immunisent pas nécessairement contre ce type d'attaques.

Title: Discrete logarithm problem on elliptic curves

Mots clés: elliptic curves, cryptography, implementation security

Abstract: The use of elliptic curves in cryptography has become widespread to ensure the security of communications or financial transactions. This is mainly due to the fact that its security relies on the difficulty of the discrete logarithm problem which allows the use of elliptic curves with parameters that ensure efficiency.

In this thesis, we mainly address the security aspect of elliptic curves implementations. The use of auxiliary data obtained through leaks related to the execution of the code can compromise protocols security. First, we analyze several formulas and algorithms commonly used in implementations to show the difficulties that a developer can face in order to realize a secure implementation. Then we show that some protection techniques can lead to a vulnerability, one of which is new and has been reported to developers. Finally, we propose a new fault injection attack on an algorithm and we also show that some protection methods based on the introduction of a randomization of the secret value are not necessarily immune to this type of attack.