RAPPORT FINAL DE PSC

INF08
Échange de clé authentifié par mot de passe post-quantique

2023-2024

Christopher Calvet, Guillaume Chirache, Timothée Fisher,
Thomas Sauvage et Emre Ucar

Encadrés par Mélissa Rossi (ANSSI)

Résumé opérationnel

La sécurité de nombreuses applications informatiques repose sur la cryptographie symétrique, mais la clé commune (dite « de session ») est échangée via la cryptographie asymétrique. Cette dernière est cependant vulnérable à l'attaque dite de l'« homme du milieu ». Pour l'éviter, il est nécessaire que les interlocuteurs s'authentifient mutuellement. Cela se fait généralement, par exemple sur le web, en utilisant des infrastructures à clés publiques, mais les ressources et la connectivité qu'elles requièrent en font des solutions inadaptées dans certains cas, notamment pour la lecture des données biométriques sur les cartes d'identité et passeports. Dans ce cas, une authentification par mot de passe, y compris de faible entropie (comme un code PIN), peut être utilisée, formant un protocole appelé PAKE (*Password Authenticated Key Exchange*) et sécurisé face aux attaques hors-ligne sur des communications interceptées.

Notre PSC a consisté à élaborer un PAKE post-quantique, c'est-à-dire résistant aux attaques qui seraient menées par un ordinateur quantique. Celui-ci est fondé sur le protocole CAKE publié par un groupe de chercheurs auquel appartient notre tutrice Mélissa Rossi.

Pour rendre ce protocole amont effectivement implémentable, nous avons résolu plusieurs problèmes théoriques. Pour empêcher les attaques *offline*, il était en particulier nécessaire d'encoder et de chiffrer du matériel cryptographique de manière à ce que le chiffré soit indiscernable d'une chaîne tirée aléatoirement. Nous avons aussi proposé un algorithme de chiffrement symétrique qui vérifie les propriétés de sécurité particulières de l'*ideal cipher*.

Une fois ces difficultés théoriques résolues, nous avons développé puis publié deux implémentations de CAKE en Python et en C, la première à des fins de démonstration algorithmique et la seconde pour estimer les performances plus précisément. Sur cette base, nous avons réalisé un démonstrateur physique fondé sur une *Raspberry Pi*. Celui-ci nous a permis de conclure notre PSC par des études de performance.

! Les implémentations que nous avons réalisées sont publiquement disponibles à l'adresse suivante :

`https://github.com/pq-pake`

Remerciements

Nous tenons tout d'abord à remercier chaleureusement Mélissa Rossi, notre tutrice tout au long de ce projet. En tant que membre de l'équipe ayant publié le protocole amont CAKE, son expertise et son avenance vis-à-vis de nos nombreuses questions et sollicitations ont été d'une aide précieuse.

Nous souhaitons également remercier Mme Rossi et ses collègues de l'ANSSI pour leur accueil dans leurs bureaux en janvier dernier et pour les échanges enrichissants que nous avons eus avec eux lors de notre visite. Ceux-ci nous ont grandement aidés à résoudre le problème majeur de notre PSC et à avancer dans notre réflexion.

Nous remercions également M. Schaeffer, notre encadrant pédagogique au sein du LIX, pour ses conseils et son suivi de notre projet.

Enfin, nous remercions Agnès Février, responsable scolarité 2A, et Fatima Pires Frances, assistante du département de l'enseignement de l'informatique, qui ont rendu ce PSC possible en assurant son suivi administratif et en organisant la soutenance.

Table des matières

1 État de l'art sur les PAKE

1.1 L'authentification lors d'un échange de clés : infrastructure à clés publiques ou mot de passe

La cryptographie, c'est-à-dire le fait de chiffrer des données et communications afin de les rendre illisibles pour les tiers non autorisés, est une composante essentielle de la sécurité informatique moderne. Elle est aujourd'hui utilisée par pratiquement tous les sites web, serveurs de messagerie et applications de communication pour se prémunir des interceptions réseau, et les appareils (ordinateurs, téléphones) vendus sont quasiment tous chiffrés par défaut.

Le cas classique est celui de la **cryptographie symétrique**, qui regroupe les protocoles où le chiffrement et le déchiffrement se font par la même clé secrète. L'algorithme le plus communément utilisé est l'*Advanced Encryption Standard* (AES), publié en 2000, et il est en pratique utilisé pour quasiment toutes les communications aujourd'hui.

Son prérequis est néanmoins que les deux personnes qui communiquent (appelons-les Alice et Bob comme on le fait classiquement) partagent une clé secrète commune. Il faut donc également définir un protocole d'**échange de clé**. En pratique, l'idée est en général d'utiliser du **chiffrement asymétrique**, où la clé de chiffrement (qu'on appelle la clé publique) est différente de la clé de déchiffrement (généralement appelée clé secrète ou clé privée). Si Alice veut communiquer avec Bob, elle génère une clé symétrique, la chiffre avec la clé publique de Bob avant de la lui transmettre. Bob n'a plus qu'à déchiffrer la clé symétrique avec sa clé privée, et ils peuvent l'utiliser pour s'échanger des messages.

Ce schéma requiert cependant qu'Alice soit sûre que la clé publique dont elle dispose soit celle de Bob, sous peine que quelqu'un se fasse passer pour lui. Cela peut être garanti par une infrastructure à clés publiques (PKI) : c'est-à-dire une autorité de confiance (en pratique une chaîne d'autorités de confiance) qui fournit les clés publiques à utiliser. Les sites web utilisent des certificats SSL fournis par des autorités de certification garantissant que le site web visité correspond bien au nom de domaine affiché.

Pour autant, la gestion d'une PKI est relativement complexe, car elle requiert des capacités de calcul et de communication (accès à Internet) qui ne sont pas réunies dans tous les cas de figure. On peut citer par exemple les objets connectés, les cartes à puce ou les passeports biométriques. Dans ces cas, il est nécessaire de trouver un autre moyen de s'assurer de l'identité de l'autre partie. C'est le rôle d'un échange de clé authentifié par mot de passe (PAKE).

Un **échange de clé authentifié par mot de passe (PAKE)** est un protocole permettant de s'accorder sur une clé de session, au cours duquel les deux parties s'authentifient mutuellement via un mot de passe commun. Elles s'assurent ainsi que leur interlocuteur est celui qu'il prétend être. De plus, aucune des parties ne révèle le mot de passe au cours de l'échange. [Boyko *et al.*, 2000]

Les PAKE sont utilisés pour la protection des données sensibles des puces des passeports de tous les pays européens et de la nouvelle carte d'identité française. [Ministère de l'Intérieur et des Outre-mer, 2021]

Pour rendre les PAKE utilisables en pratique, on doit s'autoriser des **mots de passe de faible entropie**[a] (i.e. des mots de passe faibles), et s'assurer qu'il soit malgré tout difficile pour un attaquant de les obtenir.
Un attaquant voulant obtenir le mot de passe utiliserait en général une attaque par dictionnaire,

où il teste toutes les combinaisons possibles (ou les plus probables) jusqu'à trouver la bonne. Pour qu'une telle attaque soit fructueuse, il doit disposer d'un **test d'arrêt** lui permettant de savoir si le mot de passe qu'il a essayé est le bon.
On peut en ce sens distinguer deux types d'attaques :

— Les attaques *online*, c'est-à-dire en présence du composant à protéger. Un test d'arrêt simple est de tenter un échange de clés avec le mot de passe proposé, puis d'envoyer un message avec la clé de session obtenue et de voir si la réponse est cohérente. Comme un tel test d'arrêt ne peut pas être empêché, le nombre de tentatives *online* doit être limité, ce qui est faisable par la lenteur même des composants, ou l'implémentation de mécanismes de blocage après un certain nombre de tentatives infructueuses.

— Les attaques *offline*, à partir de communications interceptées. Il n'est alors pas possible de limiter le nombre de tentatives, et il est donc nécessaire de garantir l'**absence de test d'arrêt**. Autrement dit, les communications interceptées ne doivent donner aucune indication à un attaquant quant au fait de savoir si le mot de passe qu'il a essayé est le bon. Typiquement, il doit être impossible d'obtenir à la fois un message en clair et sa version chiffrée avec le mot de passe, et la donnée du mot de passe ne doit pas rendre certains chiffrés impossibles ou plus ou moins probables.

La sécurité du PAKE se définit donc par l'absence de test d'arrêt *offline*.

a. Si un mot de passe est généré uniformément parmi N possibilités, il a une entropie de $\log_2(N)$. Par exemple, un PIN à 6 chiffres a une entropie d'environ 20 bits, ce qui est très faible.

1.2 La nécessité de développer des algorithmes post-quantiques

Tous ces mécanismes de cryptographie sont robustes et, malgré des évolutions dues principalement à la puissance de calcul croissante des ordinateurs, sont utilisés depuis une vingtaine d'années. Leur robustesse s'appuie sur des problèmes mathématiques pour lesquels aucun algorithme de résolution en temps polynomial n'est connu, comme le calcul du logarithme discret ou la factorisation d'entiers en facteurs premiers.

Cependant, l'émergence possible d'ordinateurs quantiques avec un grand nombre de qubits efficaces est susceptible de changer la donne, en permettant l'exécution d'algorithmes non utilisables sur un ordinateur classique. Si AES n'est pas menacé (il suffit de doubler la taille des clés pour résister à l'algorithme de Grover), les algorithmes de chiffrement asymétrique (et donc de PAKE) sont quant à eux cassés par les algorithmes comme celui de Shor. Il est donc nécessaire de trouver des alternatives post-quantiques à ces algorithmes, c'est-à-dire des algorithmes de chiffrement **exécutés sur un ordinateur classique**, mais résistant aussi bien aux attaques par des ordinateurs classiques que quantiques. [Shor, 1997][ANSSI, 2022][Jaques *et al.*, 2020]

Le concept de PAKE existe depuis les années 1990, et un certain nombre de protocoles ont été proposés, avec des degrés de preuve différents. Les plus utilisés dans l'industrie sont EKE et OEKE. Ils sont basés sur le problème du logarithme discret, et sont donc cassés par les ordinateurs quantiques. [Boyko *et al.*, 2000][Bellovin et Merritt, 1992]

1.3 Le protocole amont CAKE

Pour se préparer à l'arrivée d'ordinateurs quantiques plus puissants, il est donc nécessaire de construire un PAKE résistant aux attaques par des ordinateurs classiques et quantiques, dit **post-quantique**.

L'objet de ce PSC est d'implémenter un PAKE post-quantique, et plus précisément les protocoles CAKE et OCAKE, proposés en 2023 par une équipe comprenant Mme Mélissa Rossi, notre tutrice, chercheuse à l'ANSSI. [Beguinet *et al.*, 2023]

La démarche suivie dans l'article est de montrer qu'il est possible de convertir un *Key Encapsulation Mechanism* en PAKE, sous réserve qu'il vérifie un certain nombre de bonnes propriétés (comme l'indistinguabilité des clés publiques et des chiffrés échangés par rapport à un tirage aléatoire).

Un *Key Encapsulation Mechanism* (KEM) est un système permettant à deux interlocuteurs (Alice et Bob) de s'accorder sur une clé symétrique. C'est la donnée de trois algorithmes $(\texttt{KeyGen}, \texttt{Encaps}, \texttt{Decaps})$.

- Alice utilise `KeyGen` pour générer une paire de clés asymétriques $(\texttt{pk}, \texttt{sk})$. `pk` désigne ici la clé publique et `sk` désigne la clé privée.
- Bob reçoit `pk` et appelle la fonction `Encaps`. Celle-ci prend en entrée la clé `pk`, génère une clé symétrique K, ainsi qu'un chiffré c de K obtenu avec la clé publique `pk`. On dit que la clé K est **encapsulée**. Bob garde K et envoie c à Alice.
- Alice reçoit c et utilise `Decaps`, qui prend en entrée sa clé privée `sk` et le chiffré c, et renvoie K'. La clé est dite **décapsulée**. Le KEM est correct si $K = K'$ avec écrasante probabilité. Ainsi, Alice et Bob disposent de la même clé symétrique K.

Il faut noter que les KEM ne fournissent **pas d'authentification**, c'est-à-dire qu'ils ne permettent pas de s'assurer que l'interlocuteur est bien celui qu'il prétend être. Pour cela, il faut ajouter une étape supplémentaire, ce qui est le but du PAKE.

Le fonctionnement de CAKE est schématisé dans la figure 1 (et celui de OCAKE dans la figure 2). Dans un premier temps, Alice génère une paire de clés $(\texttt{pk}, \texttt{sk})$, puis envoie la clé publique chiffrée à Bob à l'aide d'un identifiant de session et du mot de passe. Bob la déchiffre, puis utilise `KEM.Encaps` à partir de `pk` pour obtenir une clé symétrique K et son chiffré c. Il envoie à Alice c, toujours chiffré avec identifiant de session et mot de passe, ce qu'Alice peut déchiffrer. Elle peut ainsi utiliser `sk` pour obtenir K, et la clé finale symétrique, dite clé de session et notée `SK`, est dérivée des clés désormais communes (en particulier K).

Dans les figures 1 et 2 :

- `NomAlice` et `NomBob` sont des noms uniques donnés aux deux interlocuteurs.
- `pw` est un mot de passe (considéré de faible entropie) partagé entre les deux interlocuteurs.
- `KEM.KeyGen`, `KEM.Encaps` et `KEM.Decaps` sont les fonctions de génération de clé, d'encapsulation et de décapsulation du *Key Encapsulation Mechanism* (KEM).
- $(\texttt{E}_1, \texttt{D}_1)$, $(\texttt{E}_2, \texttt{D}_2)$ $(\texttt{E}, \texttt{D})$ sont des fonctions de chiffrement et de déchiffrement symétrique utilisant une clé de chiffrement.
- `hash` est une fonction de hachage cryptographique (pour nous, SHA-256).
- $||$ est l'opérateur de concaténation.

On peut comprendre intuitivement pourquoi et à quelles conditions ce schéma est sécurisé. Dans un KEM, seuls c et `pk` transitent sur le réseau, ce qui empêche de déduire directement $K = K'$ à partir de données interceptées. Une attaque fonctionnelle ne peut être qu'une attaque de l'« **homme du milieu** », où un attaquant se fait passer pour Bob auprès d'Alice et inversement. L'authentification par mot de passe doit rendre de telles attaques difficiles.

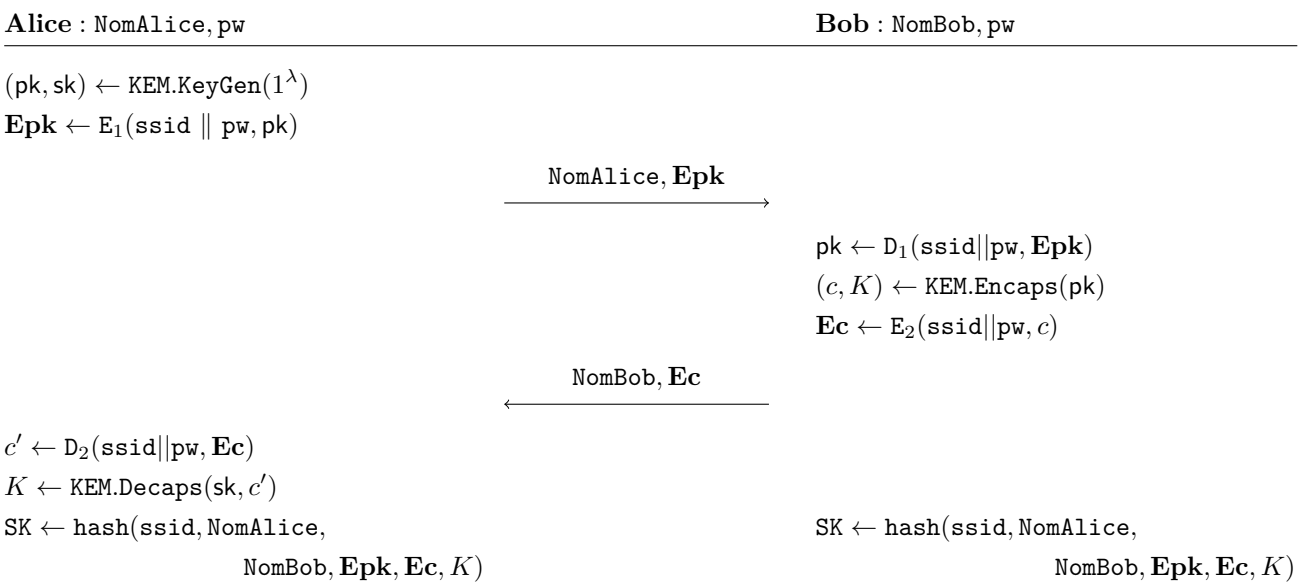

Figure 1 – Fonctionnement du protocole CAKE

! Pour éviter les attaques de l'homme du milieu, on chiffre pk et c avec une clé symétrique dérivée du mot de passe pw, et l'on transmet ainsi **Epk** et **Ec** à la place sur le réseau. Il faut alors s'assurer que l'attaquant ne puisse pas, à partir de (**Epk**, **Ec**), obtenir un test d'arrêt sur pw lui permettant de le retrouver par attaque *offline*, et ainsi mener par la suite une attaque de l'homme du milieu avec succès.
Cela requiert plusieurs conditions :

— Il ne doit pas être possible de distinguer la clé publique pk d'un tirage aléatoire (*fuzziness* du KEM), et il doit en être de même pour le chiffré c (*anonymity* du KEM).

— De même, le chiffrement qui s'applique à pk et c doit ressembler à une permutation aléatoire, autrement dit (E_1, D_1) et (E_2, D_2) doivent être des *ideal ciphers*, notion introduite formellement en partie 2.1.

L'article [Beguinet *et al.*, 2023] montre que **CRYSTALS-Kyber**, un KEM post-quantique en voie d'être standardisé par le *National Institute of Standards and Technology* (NIST) américain, vérifie la *fuzziness* et l'*anonymity*. Nous choisissons donc Kyber comme KEM.
Tout l'enjeu est alors de trouver des *ideal ciphers* convenables.

En effet, le protocole proposé dans [Beguinet *et al.*, 2023] est encore très « amont », et ces questions pratiques d'implémentation ne sont pas abordées dans l'article. En particulier, l'encodage pratique de la clé publique générée par **CRYSTALS-Kyber** réduit sa *fuzziness*, nous avons donc dû revoir son encodage dans la partie 3.2. De plus, les couples (E_1, D_1), (E_2, D_2) (E, D) doivent vérifier les propriétés de l'*ideal cipher*. Nous devions donc trouver des fonctions cryptographiques convenables. La partie 2 détaille la solution que nous proposons à ce problème.

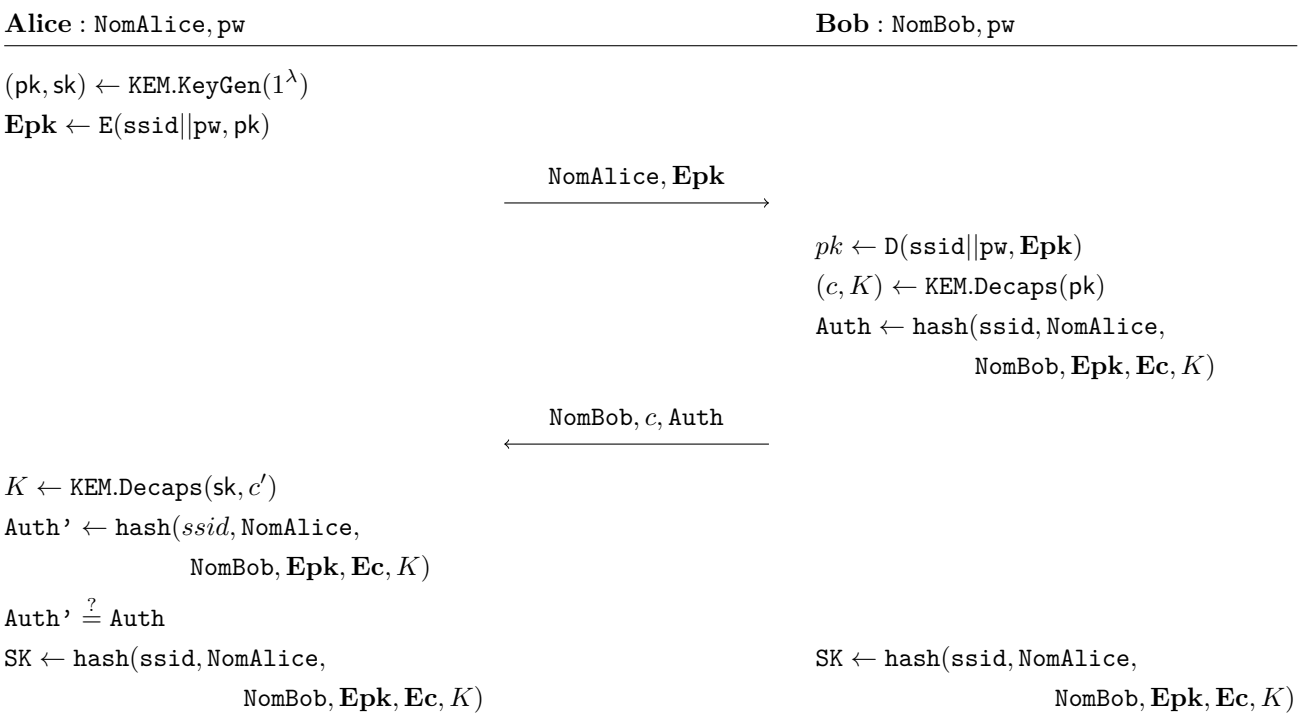

Figure 2 – Fonctionnement du protocole OCAKE

1.4 Organisation de notre travail

Au début de l'année, nous avons jalonné notre projet en trois étapes principales, ce qui nous a permis de garder une vision claire de notre progression et de nous organiser en conséquence. Ces jalons étaient les suivants :

— Tout d'abord, approfondir nos compétences en cryptographie afin de nous approprier les algorithmes CAKE et OCAKE.

— Ensuite, identifier et résoudre les difficultés théoriques liées à l'implémentation de ces algorithmes, en particulier la conception d'un protocole de chiffrement symétrique répondant aux exigences de la preuve développée dans [Beguinet *et al.*, 2023]. Nous souhaitions en outre concevoir un protocole « hybride » alliant des chiffrements pré-quantiques et post-quantiques.

— Enfin, implémenter ces protocoles dans le langage C et créer un démonstrateur simulant un cas d'utilisation réelle de notre travail (passeports biométriques ou cartes d'identité par exemple).

Pendant le déroulement de notre projet, nous travaillions ensemble le mercredi. Des réunions bihebdomadaires avec notre tutrice en visioconférence nous permettaient de faire un point sur l'avancement de notre projet. Nous avons également utilisé un dépôt Github pour gérer notre code et les différents livrables.

Venant de filières différentes, nous avions des centres d'intérêt différents : certains étaient plus intéressés par la théorie et d'autres par les aspects plus concrets. Nous avons donc réparti les tâches en fonction de nos compétences et de nos envies.

2 Le problème du chiffrement symétrique : l'*ideal cipher*

Lors de l'exécution du protocole CAKE, nous avons vu que nous devons disposer de deux *ideal ciphers* $(\mathtt{E}_1, \mathtt{D}_1)$ et $(\mathtt{E}_2, \mathtt{D}_2)$ pour chiffrer la clé publique $\mathtt{pk}$ de Kyber et le *ciphertext* c. La recherche de fonctions convenables a constitué une partie importante de notre recherche. Il était en effet nécessaire non seulement de trouver un algorithme convenable, mais aussi d'encoder différemment la clé publique (par rapport à la spécification de Kyber).

! La preuve du protocole CAKE n'est pas valide si $(\mathtt{E}_1, \mathtt{D}_1)$ et $(\mathtt{E}_2, \mathtt{D}_2)$ ne sont pas des *ideal ciphers*.

Nous détaillons dans cette partie en quoi les algorithmes classiques comme AES ne conviennent pas, et la solution que nous avons retenue.

2.1 Description du problème et modèle de l'*ideal cipher*

Chiffrer un sous-groupe. Soit q un nombre premier. Notons $\mathbb{F}_q$ l'unique corps fini de cardinal q (à isomorphisme près) et $\mathbb{F}_q[X]$ l'ensemble des polynômes à coefficients dans $\mathbb{F}_q$. Posons $\mathcal{R}_q = \mathbb{F}_q[X]/(X^n + 1)$ le groupe-quotient de $\mathbb{F}_q[X]$ par l'idéal de $\mathbb{F}_q[X]$ engendré par $X^n + 1$. Les clés publiques de Kyber sont alors des éléments de l'ensemble $\mathbb{G} = \mathcal{R}_q^k$. Les paramètres habituels utilisés dans le protocole Kyber sont $q = 3329$, $n = 256$ et $k = 4$. Notons que $\mathbb{G}$ est fini de cardinal q^{nk}, donc nous identifierons ici $\mathbb{G}$ à $[\![0, q^{nk} - 1]\!]$.

Ainsi, le problème est de construire un *ideal cipher* de $[\![0, q^{nk} - 1]\!]$. Formellement, il s'agit de construire une famille de bijections $E_k : \mathbb{G} \to F_k$ où $k \in \mathcal{K}$ est la clé de chiffrement et F_k est le domaine des messages chiffrés, vérifiant la propriété suivante :

La famille (E_k) constitue un *ideal cipher* lorsqu'on peut considérer que chaque clé $k \in \mathcal{K}$ définit une permutation aléatoire.

Concrètement, cela signifie que l'interception d'un chiffré $E_k(m)$ pour $m \in \mathbb{G}$ et $k \in \mathcal{K}$ ne doit donner aucune information sur la clé k.

Or un message $m \in \mathbb{G}$ est stocké sur $l = 11982$ bits, mais remarquons qu'il appartiendra toujours à $[\![0, q^{nk} - 1]\!]$ (dont le logarithme en base 2 est $\approx 11981,7$), ce qui laisse donc un espace de bits inutilisé. Cependant, presque tous les chiffrements que nous pouvons construire prennent en entrée - et renvoient - une chaîne de l bits et constituent donc des permutations de $[\![0, 2^l - 1]\!]$. Il n'en existe pas *a priori* sur $[\![0, q^{nk} - 1]\!]$. Cela signifie que l'ensemble-image $E_k(\mathbb{G})$ varie en fonction de $k \in \mathcal{K}$.

Or, lors de l'interception d'un message chiffré c, l'attaquant est certain que la clé k utilisée appartient à l'ensemble $\{k' \in \mathcal{K} \mid c \in E_{k'}(\mathbb{G})\}$, ce qui permettrait à un attaquant d'obtenir des informations sur la clé de chiffrement si les ensembles-images $E_k(\mathbb{G})$ variaient en fonction de k. C'est pourquoi il est crucial que l'ensemble-image $E_k(\mathbb{G})$ soit le même pour toutes les clés $k \in \mathcal{K}$.

[Beguinet *et al.*, 2023] propose une solution *ad hoc* pour résoudre cette difficulté. À partir d'un *ideal cipher* $E_k : [\![0, 2^l - 1]\!] \to [\![0, 2^l - 1]\!]$, nous pouvons construire un *ideal cipher* $E'_k : \mathbb{G} \to \mathbb{G}$ de la manière suivante : on itère le chiffrement E_k en partant de m, c'est-à-dire qu'on calcule successivement $E_k(m), E_k(E_k(m)), \ldots$ jusqu'à obtenir une chaîne de bits correspondant à un entier strictement inférieur à 3329^{1024}. Le déchiffrement fonctionne de la même manière : on itère E_k^{-1} jusqu'à ce qu'on obtienne une valeur sous ce seuil. La probabilité que la valeur obtenue après une itération convienne est

$$p = \frac{3329^{1024}}{2^{11982}} \approx 0,81$$

et le nombre d'itérations suit donc la loi géométrique $\mathcal{G}(p)$, dont l'espérance est $p^{-1} \approx 1,23$.

Éviter les collisions. Donnons-nous une clé k, un message m, et posons $c := E_k(m)$. Une collision est alors un couple $(k', m') \in \mathcal{K} \times \mathbb{G}$ vérifiant $k \neq k'$ et $E_k(m) = E_{k'}(m')$. Ce problème admet une solution triviale : il suffit de choisir une clé $k' \neq k$, et de poser $m' = E_{k'}^{-1}(c)$. La preuve développée dans [Beguinet *et al.*, 2023] impose une condition supplémentaire sur l'*ideal cipher* : il doit être statistiquement impossible de trouver des collisions sans recourir à la fonction de déchiffrement, comme décrit ci-dessus.

Cette contrainte écarte plusieurs pistes comme le chiffrement par flux (*stream cipher*) : ce protocole consiste à se donner un générateur pseudo-aléatoire $G_l : \mathcal{K} \to [\![0, 2^l - 1]\!]$, et à renvoyer $c := m \oplus G_l(k)$, le $\oplus$ désignant ici l'opération XOR bit-à-bit (équivalente à l'addition dans le groupe $\mathbb{Z}/2\mathbb{Z}$). Le déchiffrement consiste alors à calculer $c \oplus G_l(k)$. Ce protocole ne peut convenir car étant donné c, il est très simple de générer une collision : par exemple on tire aléatoirement une clé $k' \neq k$ et on pose $m' := c \oplus G_l(k')$. On aura alors $c' = m' \oplus G_l(k') = (c \oplus G_l(k')) \oplus G_l(k') = c$.

2.2 Plusieurs pistes infructueuses pour l'*ideal cipher*

La grande majorité des algorithmes de chiffrement (non exclus par la section précédente) comme l'AES sont des chiffrements par blocs, c'est-à-dire des algorithmes qui divisent un message en blocs de taille fixée. Généralement, ces tailles sont des puissances de 2 comme 32, 64, 128, etc. Pour chiffrer un message constitué de plusieurs blocs, l'algorithme emploie des modes d'opération pour combiner les différents blocs. Dans cette section, nous allons montrer que les modes usuels ne conviennent pas pour notre problème.

Dans toute la suite, on notera $m = B_1...B_n$ le découpage du message en ses différents blocs, E_k le chiffrement par blocs associé à la clé $k \in \mathcal{K}$ et D_k l'algorithme de déchiffrement.

2.2.1 L'*electronic codebook block* (ECB)

Ce mode consiste à chiffrer chaque bloc individuellement avec la même clé : le chiffré vaut donc $c := E_k(B_1)...E_k(B_n)$. Cette méthode est peu recommandée puisque si un message comporte deux blocs identiques (par exemple $B_s = B_t$), alors les s-ème et t-ème blocs du chiffré seront aussi identiques. Ce mode ne peut donc pas être identifié à un *ideal cipher* - où le chiffré doit être assimilable à une chaine aléatoire - et de manière plus générale, est à éviter.

2.2.2 Le *cipher block chaining* (CBC)

Ce mode consiste à combiner chaque bloc avec le chiffré du bloc précédent. Plus précisément, on se donne d'abord un vecteur d'initialisation IV : il s'agit d'un bloc généré de manière pseudo-aléatoire. On pose alors $C_0 = IV$. Puis pour tout $1 \leqslant i \leqslant n$, on pose $C_i = E_k(B_i \oplus C_{i-1})$. On renvoie alors $c := C_0...C_n$ (voir l'illustration ci-dessous).

Pour le déchiffrement, on remarque que $B_i = D_k(C_i) \oplus C_{i-1}$ pour tout $1 \leqslant i \leqslant n$. Cette formule permet ainsi de calculer le i-ème bloc du message clair à partir des $(i-1)$-ème et i-ème blocs chiffrés.

Il s'agit du mode le plus utilisé aujourd'hui. Cependant, il ne peut correspondre à un *ideal cipher* :

— Remarquons d'abord que l'égalité des premiers blocs de deux messages est conservée après chiffrement (pourvu que les vecteurs d'initialisation soient égaux).

— En outre, si l'on modifie des blocs situés à la fin du chiffré (disons $C_i, C_{i+1}, ..., C_n$), alors les premiers blocs du message clair correspondant à ce chiffré, ici $B_1, ..., B_{i-1}$, ne seront pas affectés par cette modification.

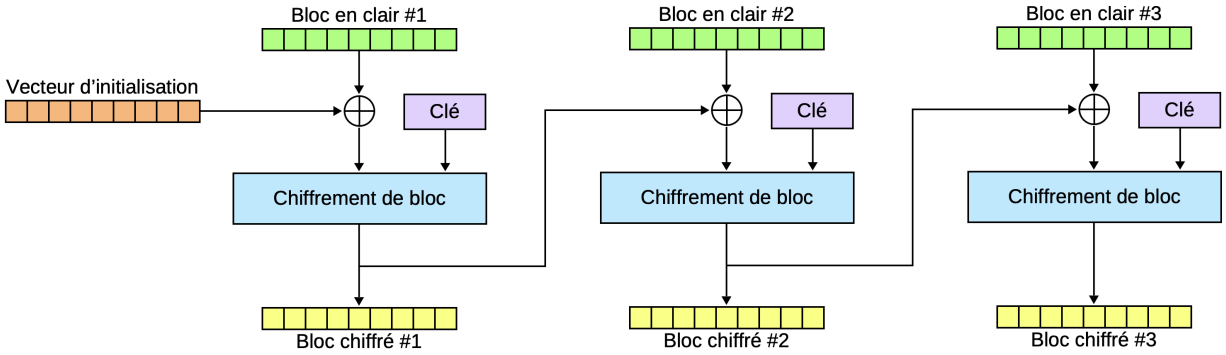

Figure 3 – Illustration graphique du *cipher block chaining*[1]

Le mode CBC ne fournit donc pas une permutation aléatoire.

2.2.3 Le *cipher feedback* (CFB)

Ce mode est très proche du *cipher block chaining*. On pose toujours $C_0 = IV$, mais pour tout $1 \leqslant i \leqslant n$, on pose $C_i = E_k(C_{i-1}) \oplus B_i$ (voir l'illustration ci-dessus). Pour le déchiffrement, on utilise l'égalité $B_i = E_k(C_{i-1}) \oplus C_i$. Remarquons que le déchiffrement n'utilise que la fonction E_k et ne nécessite pas d'appel à la fonction D_k. Cela peut constituer un avantage pour les chiffrements par blocs comme AES où les deux fonctions sont différentes. Cependant, il ne convient pas pour la même raison que le mode CBC (égalité des premiers blocs chiffrés en cas d'égalité des blocs en clair).

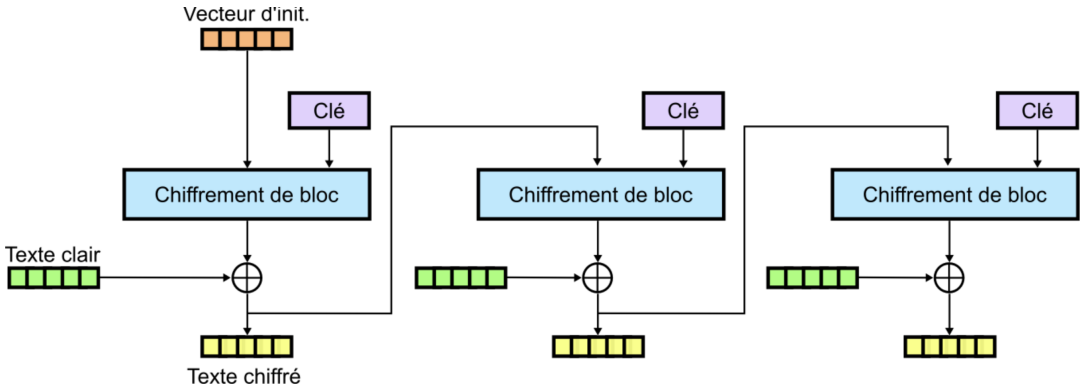

Figure 4 – Illustration graphique du *cipher feedback*[2]

2.2.4 L'*output feedback* (OFB)

Ce mode consiste à construire un chiffrement par flux à partir du chiffrement par blocs.

On se donne comme précédemment un vecteur d'initialisation IV. On définit ensuite la suite I_i récursivement par $I_0 = IV$ et $I_i = E_k(I_{i-1})$ pour tout $1 \leqslant i \leqslant n$. On pose alors $C_i := B_i \oplus I_i$ pour tout $1 \leqslant i \leqslant n$ et on renvoie $c := C_0 ... C_n$ (voir l'illustration ci-dessous).

1. Bmoine, CC BY-SA 4.0 <https ://creativecommons.org/licenses/by-sa/4.0>, via Wikimedia Commons
2. Original téléversé par Dake sur Wikipédia français., CC BY 1.0 <https ://creativecommons.org/licenses/by/1.0>, via Wikimedia Commons

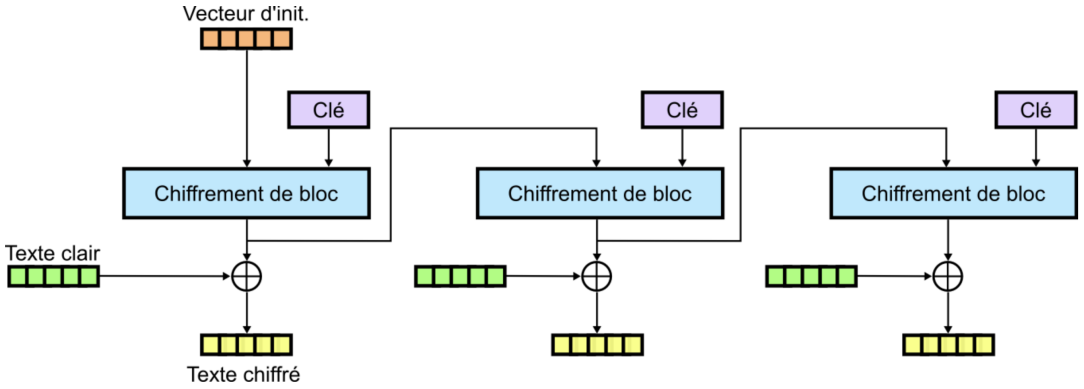

Figure 5 – Illustration graphique de l'*output feedback*[3]

Comme vu à la section 2.1, le chiffrement par flux ne peut convenir pour résoudre notre problème. Ce mode est donc à écarter ici.

2.2.5 Le chiffrement basé sur un compteur (CTR)

Comme le mode OFB, il s'agit de contruire un chiffrement par flux à partir du chiffrement par blocs. Comme précédemment, on dispose d'un vecteur d'initialisation IV. On définit alors $C_0 := IV$ et pour $1 \leqslant i \leqslant n$, on pose $C_i := B_i \oplus E_k(IV \mid\mid i)$ (où $\mid\mid$ désigne la concaténation). Par rapport au mode OFB, il a l'avantage d'être parallélisable puisque le calcul de chaque bloc C_i se fait indépendamment des autres. En revanche, il ne constitue pas un *ideal cipher* pour la même raison qu'OFB.

En somme, nous avons été amenés à écarter tous les (principaux) modes d'opération des chiffrements par blocs.

2.3 Échange avec des cryptographes de l'ANSSI

Dans le cadre de notre travail sur le protocole CAKE, nous avons eu la chance de pouvoir rencontrer une dizaine de cryptographes travaillant avec Mme Rossi à l'ANSSI. Cette rencontre nous a permis de discuter avec eux de nos problématiques principales, au premier rang desquels le problème du choix de l'algorithme de chiffrement symétrique, qui doit vérifier des propriétés très précises et inhabituelles.

Ce rendez-vous a été pour nous l'occasion de synthétiser les connaissances que nous avions acquises depuis quatre mois sur le sujet, de définir clairement les points bloquants, et de lister les pistes de recherche que nous avons envisagé avec notre tutrice.

Comme nous avions pu l'espérer, les échanges avec les cryptographes de l'ANSSI ont été très riches. Ils nous ont permis de mieux comprendre les enjeux de notre projet sous des aspects que nous n'avions pas envisagés. En effet l'équipe de Mme Rossi est composée de chercheurs de domaines très variés (chiffrement symétrique, post-quantique, sécurité matérielle, etc.) qui ont donc abordé notre présentation sous des angles différents. Nous avons ainsi revu avec eux les propriétés nécessaires pour un *ideal cipher*, discuter de la pertinence des différentes pistes que nous avions envisagées comme les réseaux de Feistel, les *half-ideal ciphers* et les *wide-block ciphers*. Voir ces experts de leurs domaines respectifs tenter de trouver des failles dans le protocole CAKE à l'aune de leur spécialité a été une

3. Original téléversé par Dake sur Wikipédia français., CC BY 1.0 <https ://creativecommons.org/licenses/by/1.0>, via Wikimedia Commons

expérience très enrichissante. Nous avons particulièrement apprécié leur pédagogie, leur disponibilité et leur bienveillance.

Après de nombreuses recherches de notre côté, ainsi que plusieurs allers-retours avec ces chercheurs les mois suivants, nous avons conclu que la meilleure solution pour notre protocole était d'utiliser un réseau de Feistel.

Nous sommes ainsi repartis de cette rencontre avec une meilleure compréhension des enjeux de notre projet et une idée claire de notre plan d'action pour les mois à venir. Nous avons aussi pu revoir l'échéancier de notre projet avec notre tutrice et valider avec elle notre feuille de route.

2.4 La solution retenue : le chiffrement de Feistel

2.4.1 Les propriétés intéressantes du chiffrement de Feistel

À la suite de nos recherches et de nos échanges avec les cryptographes de l'ANSSI, nous avons déterminé que la meilleure solution de chiffrement symétrique pour CAKE est le chiffrement par réseau de Feistel. En effet, des avancées très récentes proposent des constructions basées sur les réseaux de Feistel qui peuvent constituer un *ideal cipher* avec un nombre de tours suffisant et d'autres hypothèses sous-jacentes détaillées ci-dessous. Cette propriété est essentielle à la preuve de sécurité du protocole CAKE.

Un article publié en 2014 propose une construction pour laquelle 14 tours de Feistel suffisent pour obtenir un *ideal cipher*, et montre par ailleurs que 5 tours ne peuvent jamais suffire [Coron *et al.*, 2016]. Un autre article améliore ce résultat en réduisant le nombre de tours à 10. [Dachman-Soled *et al.*, 2015]. Nous avons choisi de retenir un chiffrement de Feistel à 14 tours pour notre implémentation.

2.4.2 Principe du chiffrement de Feistel

Le chiffrement de Feistel consiste à itérer à plusieurs reprises (14 pour nous) une opération sur le message à chiffrer. À chaque itération, le message est divisé en deux blocs de taille égale. L'un des deux blocs (celui de droite, r_i) est modifié par une fonction F_{k_i} qui dépend de la clé k et du tour i. Le résultat obtenu est alors combiné avec l'autre bloc (celui de gauche, q_i) par une opération XOR. Les deux blocs sont ensuite permutés pour former le message chiffré de l'itération suivante, comme représenté dans le schéma ci-dessous.

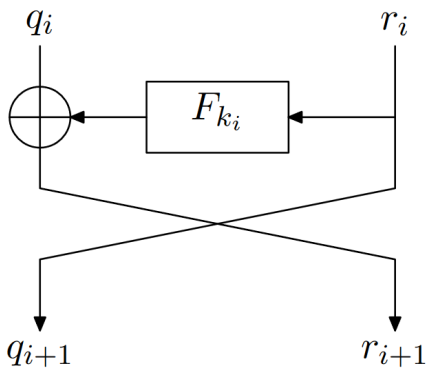

Figure 6 – Un tour du chiffrement de Feistel

La fonction F_{k_i} que nous avons choisie est égale à la fonction $m \mapsto \texttt{half\_hash}(k, m, i)$ décrite dans la section 3.1. Elle est construite en se basant sur la fonction de hachage SHA-512. Notons qu'en

itérant ce processus 2 fois on obtient $q_{i+2} = q_i \oplus \texttt{half\_hash}(\texttt{symmetric\_key}, r_i, i)$ et $r_{i+2} = r_i \oplus \texttt{half\_hash}(\texttt{symmetric\_key}, q_{i+2}, i+1)$. Nous avons utilisé ces formules car elles évitent de permuter bloc de gauche et bloc de droite, ce que nous avons trouvé plus lisible pour le code. On obtient donc l'algorithme de chiffrement suivant :

Feistel.encrypt(symmetric_key, message)

```
q ← message[0 : len(message) / 2]
r ← message[len(message) / 2 : len(message)]
for i ∈ ⟦1, NB_ROUNDS/2⟧ do
    new_left ← left ⊕ half_hash(symmetric_key, right, 2 * i − 1)
    new_right ← right ⊕ half_hash(symmetric_key, new_left, 2 * i)
    right ← new_right
    left ← new_left
end for
return left||right
```

Pour le déchiffrement, on remarque que $r_i = r_{i+2} \oplus \texttt{half\_hash}(\texttt{symmetric\_key}, q_{i+2}, i+1)$ et $q_i = q_{i+2} \oplus \texttt{half\_hash}(\texttt{symmetric\_key}, r_i, i)$.

Feistel.decrypt(symmetric_key, cipher)

```
q ← cipher[0 : len(cipher) / 2]
r ← cipher[len(cipher) / 2 : len(cipher)]
for i ∈ ⟦1, NB_ROUNDS/2⟧ do
    new_right ← right ⊕ half_hash(symmetric_key, left, NB_ROUNDS + 2 − 2 * i)
    new_left ← left ⊕ half_hash(symmetric_key, new_right, NB_ROUNDS + 1 − 2 * i)
    right ← new_right
    left ← new_left
end for
return left||right
```

3 La résolution de difficultés théoriques

Au-delà de la problématique de l'*ideal cipher* qui a constitué un enjeu essentiel du PSC, nous avons rencontré et résolu d'autres difficultés théoriques de moindre envergure, que nous avons regroupées dans cette partie.

Ces difficultés concernent essentiellement l'implémentation effective de l'*ideal cipher* via le chiffrement de Feistel telle que décrite dans la partie précédente. Il était en effet nécessaire de générer des *hashs* de taille arbitraire (3.1) et de réencoder la clé publique de `Kyber` (3.2) pour que l'*ideal cipher* soit utilisable. En plus de cela, nous avons travaillé sur la résistance du protocole aux attaques temporelles sur la clé publique (3.3) et le *ciphertext* (3.4), ainsi que sur l'hybridation avec un algorithme pré-quantique (3.5).

3.1 La génération de *hashs* de taille arbitraire

Le chiffrement de Feistel repose sur une *fonction de hachage sécurisée*. Nous avons choisi d'utiliser pour cela la fonction SHA-512 (notée `sha512`) standardisée par le NIST.

> Une fonction de hachage sécurisée est une fonction associant des valeurs de taille fixe à des données de taille quelconque et qui soit résistante aux collisions et aux images réciproques (autrement dit, il n'est pas possible d'obtenir la pré-image d'une valeur plus facilement que par force brute).

D'après [Coron *et al.*, 2016], notre algorithme de chiffrement est alors un *ideal cipher* si la fonction de hachage se comporte comme un oracle aléatoire (*random oracle*), c'est-à-dire si elle renvoie un *hash* indiscernable d'une chaîne tirée aléatoirement.

`sha512` renvoie un bloc de 512 bits, alors que nous avons besoin d'obtenir des valeurs de hachage (*hashs*) de taille arbitraire (plus exactement la moitié de celle du message chiffré) lors des tours de Feistel. Par exemple, pour chiffrer la partie polynomiale de la clé publique `pk` de Kyber, il faut chiffrer 11982 bits, donc obtenir des *hashs* de 5991 bits.

La solution que nous avons retenue, pour obtenir un *hash* de ℓ bits, est de calculer $\lceil \ell/512 \rceil$ blocs de 512 bits, puis de les concaténer et de tronquer le résultat à ℓ bits. Pour obtenir des blocs différents, le i-ème message est concaténé avec i avant d'appliquer `sha512`. Cette méthode est illustrée par la figure 7 dans le cas $\ell = 5991$. Notons qu'il est courant de supposer que `sha512` est un *random oracle*. Puisque la fonction `hash_half` renvoie une concaténation de chaînes générées par `sha512`, on peut légitimement supposer qu'elle constitue un *random oracle*, et donc que notre chiffrement constitue un *ideal cipher*.

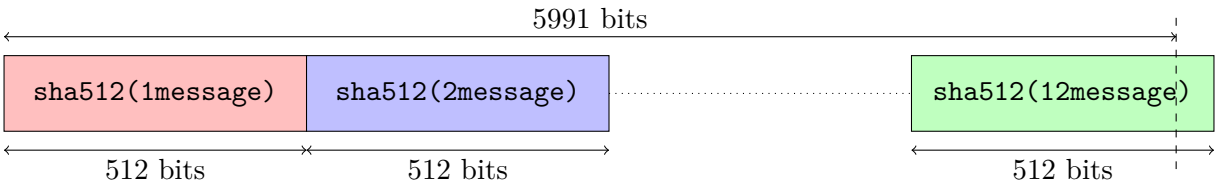

Figure 7 – Génération d'un *hash* de `message` faisant 5991 bits via `sha512`

L'algorithme que nous avons utilisé dans nos implémentations est donc le suivant, en tenant compte du fait que la longueur du résultat était toujours égale à celle de l'entrée (qui correspond à la moitié de la taille du message chiffré). Par ailleurs, le message se voit adjoindre une clé de chiffrement

`symmetric_key` par le fonctionnement de Feistel [4], ainsi que le numéro du tour (car les fonctions F_{k_i} doivent être indépendantes les unes des autres pour la preuve de [Coron *et al.*, 2016]).

`hash_half(symmetric_key, message, round)`

```
result ← bits()
for i ∈ ⟦1, ⌈ℓ/512⌉⟧ do
    result ← result ‖ sha512(i ‖ message ‖ symmetric_key ‖ round)
end for
return result[: ℓ]
```

3.2 L'encodage compact de la clé publique

Les deux échanges ayant lieu sur le réseau et devant être protégés par le modèle de l'*ideal cipher* sont les échanges de **Epk** et de **Ec**. La solution que constitue le chiffrement de Feistel est valide pour chiffrer l'ensemble des chaines de bits d'une taille paire fixée dans lui-même. Cependant, les propriétés exigées par CAKE (*fuzziness* et *anonymity*) demandent de plus que les données à chiffrer soient encodées de manière à prendre toutes les valeurs de l'espace de départ ([Beguinet *et al.*, 2023]), comme expliqué dans la section 2.1. [5]

En ce sens, la constitution de la clé publique `pk` est problématique. Elle est constituée d'une graine de 32 octets (256 bits convenablement uniformes) et de 1024 coefficients compris entre 0 et 3328. La spécification officielle propose d'encoder chacun de ces coefficients sur 12 bits (puisque $\log_2(3329) \approx 11,7$) et d'y adjoindre la graine, ce qui donne au total $12 \times 1024 + 256 = 12544$ bits, soit 1568 octets. Si l'on appliquait directement Feistel, toutes les valeurs de l'espace de départ (et donc d'arrivée) ne seraient pas possibles, ce qui permettrait à un attaquant d'obtenir des informations sur le mot de passe `pw` par la donnée de **Epk**.

Pour pallier ce problème, nous avons développé un « encodage compact » des coefficients polynomiaux. On les sépare d'abord de la graine, à laquelle on peut appliquer Feistel normalement.

Le polynôme peut prendre 3329^{1024} valeurs différentes, ce qui requiert un encodage de $\approx 11981,7$ bits. Au lieu de concaténer les encodages des coefficients $c_0, \ldots, c_{1023}$, on stocke l'encodage de l'entier $S = \sum_{i=0}^{1023} c_i 3329^i$ (autrement dit on voit les coefficients comme des chiffres en base 3329). On obtient la fonction `encode` transformant l'encodage donné par la spécification en notre encodage compact, et sa réciproque `decode`. Nous les reproduisons ci-dessous en pseudo-code simplifié.

`encode(message)`

```
S ← 0
for i ∈ ⟦0, 1023⟧ do
    c_i ← Σ_{j=0}^{11} message_{12i+j} 2^j
    S ← S + 3329^i c_i
end for
return S.to_bits()
```

4. Cette clé est directement dérivée du mot de passe `pw` et de variables publiques liées à la session.
5. En plus du fait que ces valeurs soient indiscernables d'un tirage aléatoire, ce que vérifie Kyber.

`decode(encoded_message)`

$S \leftarrow$ `encoded_message.to_int()`
for $i \in [\![0, 1023]\!]$ **do**
  $c_i \leftarrow S \pmod{3329}$
  $S \leftarrow \lfloor S/3329 \rfloor$
end for
`message` $\leftarrow$ `bits()`
for $i \in [\![0, 1023]\!]$ **do**
  $\texttt{message}_{12i}, \ldots, \texttt{message}_{12i+11} \leftarrow c_i$`.to_bits()`
end for
return `message`

La partie polynomiale ainsi encodée est stockée sur 11982 bits, auxquels on pourrait théoriquement appliquer Feistel, mais la situation n'est toujours pas satisfaisante. En effet, les valeurs de S ne sont comprises qu'entre 0 et $3329^{1024} - 1$ (dont le logarithme en base 2 est $\approx 11981,7$), laissant un espace de bits inutilisé.

La solution retenue est celle suggérée dans [Beguinet *et al.*, 2023] et décrite formellement dans la partie 2.1. Pour rappel. il s'agit d'itérere chiffrement de Feistel plusieurs fois, jusqu'à obtenir une chaîne de bits correspond à un entier strictement inférieur à 3329^{1024}. On a vu que la probabilité que la valeur obtenue après une itération convienne est $p \approx 0,81$.

De cette manière, l'espace des chiffrés est le même que l'espace de départ, et en particulier il ne dépend pas du mot de passe `pw` et ne permet donc pas de déduire des informations sur celui-ci.

3.3 Protection contre les attaques temporelles

La méthode présentée précédemment permet d'obtenir un *ideal cipher* pour le chiffrement de la clé publique. Elle présente néanmoins le défaut d'avoir un nombre d'itérations dépendant directement du mot de passe, ce qui ouvre la voie à des **attaques temporelles**.

! Les attaques temporelles (ou *timing attacks*) sont une famille d'attaques qui exploitent le temps de réponse de l'appareil à une requête pour obtenir des informations sur le mot de passe.
Par exemple, si l'attaquant prend la place d'Alice dans le protocole et qu'il peut choisir **Epk**, il peut observer de combien d'itérations Bob a besoin pour déchiffrer son entrée, et voir quels mots de passe correspondent. En répétant ses observations avec des **Epk** différents, et étant donné la faible entropie du mot de passe, il semble concevable qu'il le trouve en temps raisonnable.

Pour rendre ces attaques impossibles, nous avons choisi de rendre le nombre d'itérations N fixe. Ces itérations sont calculées quoi qu'il en soit, mais seul le premier résultat correspondant à un entier strictement inférieur à 3329^{1024} est pris en compte, ce qui signifie que les itérations suivantes sont faites « dans le vide ».

Si aucune des N itérations ne donne un résultat convenable, le chiffrement échoue. La probabilité que cela arrive doit être infime afin de garantir le bon fonctionnement du protocole et d'empêcher les attaques visant à provoquer cette erreur (ce qui serait équivalent à une attaque temporelle, révélant que la combinaison $(\texttt{pw}, \textbf{Epk})$ requiert plus de N itérations). Nous avons fixé pour objectif que cette probabilité soit sous 2^{-128}, qui est la probabilité d'erreur tolérée pour Kyber dans sa spécification

([Avanzi *et al.*, 2021]). On veut donc :

$$-N \log_2(1-p) > 128 \iff N \geqslant 54,$$

et on choisit ainsi $N = 54$ (figure 8).

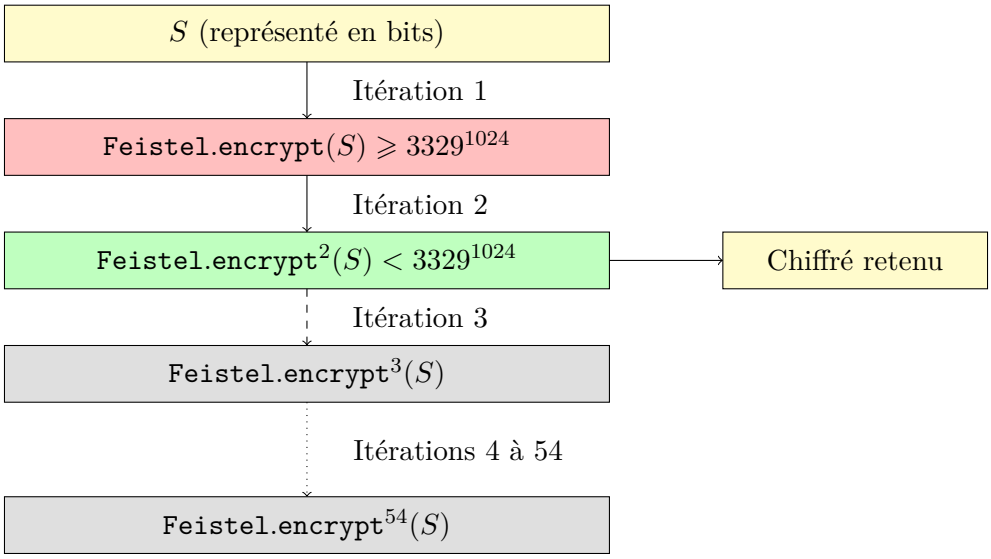

Figure 8 – Les $N = 54$ itérations sont toujours exécutées, mais seules quelques-unes (ici les deux premières) jouent en pratique un rôle dans la détermination du chiffré.

La fonction complète chiffrant une clé publique est donc la suivante, en pseudo-code. Rappelons que `symmetric_key` est directement dérivée du mot de passe `pw` à l'entropie supposée faible, ce qui justifie les efforts faits pour éviter les attaques sur cette clé.

`encrypt_pk(symmetric_key, pk)`

$v, s \leftarrow$ `pk`$[: -256], $`pk`$[-256 :]$ ▷ Séparation du polynôme v et de la graine s
$v \leftarrow$ `encode`(v) ▷ Utilisation de l'encodage compact
$e \leftarrow$ `None` ▷ e contiendra le chiffré du polynôme une fois trouvé.
for $\_ \in [\![0, 53]\!]$ **do**
  $v \leftarrow$ `Feistel.encrypt`$($`symmetric_key`$, v)$
  $b \leftarrow v$`.to_int`$() < 3329^{1024}$ and $e =$ `None`
  $e \leftarrow (1-b) \times e + b \times v$
end for
if $e =$ `None` **then** ▷ Les 54 itérations n'ont pas donné de résultat convenable.
  `Error: encryption could not be performed.`
end if
return Epk $= e \parallel$ `Feistel.encrypt`$($`symmetric_key`$, s)$

Et on en déduit immédiatement sa réciproque `decrypt_pk`.

`decrypt_pk`(`symmetric_key`, **Epk**)

$s \leftarrow$ `Feistel.decrypt`(`symmetric_key`, **Epk**$[-256:]$) ▷ La graine est immédiate.
$e \leftarrow$ **Epk**$[:-256]$
$v \leftarrow$ `None` ▷ v contiendra le polynôme déchiffré une fois trouvé.
for $\_ \in [\![0, 53]\!]$ **do**
  $e \leftarrow$ `Feistel.decrypt`(`symmetric_key`, e)
  $b \leftarrow e$.`to_int`$() < 3329^{1024}$ and $v =$ `None`
  $v \leftarrow (1-b) \times v + b \times e$
end for
if $v =$ `None` **then** ▷ Ne devrait jamais arriver puisque le chiffrement a eu lieu.
  `Error: decryption could not be performed.`
end if
return `pk` $=$ `decode`$(v) \parallel s$ ▷ Encodage compact

Le bloc
  $b \leftarrow v$.`to_int`$() < 3329^{1024}$ and $e =$ `None`
  $e \leftarrow (1-b) \times e + b \times v$
est fonctionnellement équivalent à
  if v.`to_int`$() < 3329^{1024}$ and $e =$ `None` **then**
    $e \leftarrow v$
  end if
et on peut faire la même analogie dans `decrypt_pk`.
Toutefois, l'introduction d'un booléen permet de garantir que les mêmes opérations sont effectuées pour chacune des 54 itérations, que la condition soit vraie ou fausse. Sans cela, la dernière itération « réelle » pourrait être repérée dans les traces de consommation de puissance comme celle ayant causé une affectation en plus, ce qui rendrait inutile la protection implémentée contre les attaques temporelles.

De cette manière, nous avons bien construit un *ideal cipher* convenable pour la clé publique de Kyber et résistant aux attaques temporelles. En prenant `symmetric_key` $=$ `sha512`(`ssid` $\parallel$ `pw`), le couple $(\mathtt{E}_1, \mathtt{D}_1)$ de la figure 1 est directement donné par la formule ci-dessus.

$$\mathtt{E}_1(\mathtt{ssid} \parallel \mathtt{pw}, \mathtt{pk}) = \mathtt{encrypt\_pk}(\mathtt{sha512}(\mathtt{ssid} \parallel \mathtt{pw}), \mathtt{pk})$$

$$\mathtt{D}_1(\mathtt{ssid} \parallel \mathtt{pw}, \mathbf{Epk}) = \mathtt{decrypt\_pk}(\mathtt{sha512}(\mathtt{ssid} \parallel \mathtt{pw}), \mathbf{Epk})$$

La clé publique chiffrée $\mathbf{Epk} = \mathtt{E}_1(\mathtt{ssid} \parallel \mathtt{pw}, \mathtt{pk}) = \mathtt{encrypt\_pk}(\mathtt{symmetric\_key}, \mathtt{pk})$ peut alors être transmise de manière sûre sur le réseau.

3.4 Le cas du *ciphertext*

De la même manière que pour la clé publique, le chiffré (ou *ciphertext*) de `Kyber` doit être chiffré par un *ideal cipher*. La tâche est cependant bien plus aisée ici grâce à la manière dont ce chiffré est encodé dans `Kyber`.

En effet, `Kyber` existe en version non compressée et en version compressée. Dans la version non compressée, c est la donnée de deux polynômes, le premier à $4n = 1024$ coefficients et le second à

$n = 256$, tous compris entre 0 et $q - 1 = 3328$. Si cette version était utilisée, nous devrions encoder ces polynômes comme nous l'avons fait, avec un encodage compact suivi d'un nombre fixé d'itérations de Feistel.

Cependant, dans la version compressée, les coefficients polynomiaux sont compressés de façon à être compris entre 0 et $2047 = 2^{11} - 1$ pour le premier polynôme et entre 0 et $31 = 2^5 - 1$ pour le second. [Avanzi *et al.*, 2021] Il s'agit d'une compression avec perte, mais qui, par construction de `Kyber`, n'a qu'un impact négligeable sur la correction du protocole. Elle a été ajoutée pour en améliorer les performances.

L'article [Beguinet *et al.*, 2023] prouve l'*anonymity* de `Kyber` sur la version non compressée, et celle-ci est toujours valide en version compressée puisque la clé compressée résulte du troncage de la clé non compressée, elle-même assimilable à un tirage aléatoire. Les implémentations de `Kyber` sur lesquelles nous nous appuyons utilisent la version compressée.

Ainsi, le *ciphertext* que nous devons chiffrer est assimilable à un tirage aléatoire dans $\{0, \dots, 2^{11 \cdot 1024 + 5 \cdot 256} - 1\} = \{0, \dots, 2^{12544} - 1\}$, encodé sur 12544 bits, et nous pouvons donc directement lui appliquer Feistel puisque ce nombre est pair.

On définira donc l'*ideal cipher* $(\mathtt{E}_2, \mathtt{D}_2)$ comme suit :

$$\mathtt{E}_2(\mathtt{ssid} \parallel \mathtt{pw}, c) = \mathtt{Feistel.encrypt}(\mathtt{sha512}(\mathtt{ssid} \parallel \mathtt{pw}), c)$$

$$\mathtt{D}_2(\mathtt{ssid} \parallel \mathtt{pw}, \mathbf{Ec}) = \mathtt{Feistel.decrypt}(\mathtt{sha512}(\mathtt{ssid} \parallel \mathtt{pw}), \mathbf{Ec})$$

3.5 Hybridation avec un algorithme pré-quantique

L'algorithme fourni par l'équipe de Mélissa Rossi implémenté avec notre *ideal cipher* permet d'obtenir un PAKE qui est sécurisé au sens de la preuve réalisée dans [Beguinet *et al.*, 2023] et que l'on suppose résistant aux attaques quantiques. Néanmoins, les algorithmes post-quantiques sont tous relativement jeunes, par comparaison à la forte maturité des algorithmes « classiques ». Pour cette raison, l'ANSSI recommande ([ANSSI, 2022]) aujourd'hui d'utiliser de l'hybridation, c'est-à-dire de combiner chiffrement post-quantique et pré-quantique, afin d'éviter toute régression de sécurité imputable à des failles non découvertes dans les algorithmes post-quantiques. Cette recommandation, partagée par d'autres agences équivalentes comme le BSI allemand, vaut jusqu'à ce que la phase 3 du schéma de transition de l'ANSSI soit atteinte, c'est-à-dire au moins jusqu'en 2030. Le prix en performance est raisonnable car les algorithmes pré-quantiques sont généralement moins complexes que les algorithmes post-quantiques.

Dans le protocole CAKE, cette recommandation concerne le recours au KEM `Kyber`, mais pas aux *ideal ciphers* $(\mathtt{E}_1, \mathtt{D}_1)$ et $(\mathtt{E}_2, \mathtt{D}_2)$ qui relèvent de la cryptographie classique. Nous avons ainsi proposé une approche hybride pour la partie du protocole relevant du KEM.

L'idée est de remplacer Kyber par un KEM « hybride » dans l'algorithme, qui conjuguerait Kyber et un KEM pré-quantique. Il est alors important de conserver les propriétés de Kyber essentielles à la preuve de CAKE, en particulier la *fuzziness* et l'*anonymity*. Pour cela, on s'appuie sur le KEM ElGamal ([Elgamal, 1985]), fondé sur le problème de Diffie-Hellman, qui vérifie ces propriétés d'après [Beguinet *et al.*, 2023]. Notons `Kyber` et `EG` les KEM associés. On note `KEM` le KEM hybride que l'on construit.

La génération des clés consiste à utiliser les fonctions `KeyGen` des KEM pré et post-quantiques. La clé privée est donc simplement le couple $\mathsf{sk} = (\mathsf{sk}_{\mathtt{EG}}, \mathsf{sk}_{\mathtt{Kyber}})$. La clé publique est également un couple de clés, mais comme on souhaite la rendre indiscernable d'un tirage aléatoire, on l'encodera comme

la concaténation $\mathsf{pk} = \mathsf{pk}_{\texttt{EG}} \parallel \mathsf{pk}_{\texttt{Kyber}}$, qui est réversible puisque les tailles des clés sont connues. On pourra donc écrire par la suite $\mathsf{pk}_{\texttt{EG}} \parallel \mathsf{pk}_{\texttt{Kyber}} \leftarrow \mathsf{pk}$. `KeyGen` génère ainsi $(\mathsf{pk}, \mathsf{sk})$:

$\texttt{KEM}.\mathrm{KeyGen}(1^\lambda)$

1 : $(\mathsf{pk}_{\texttt{EG}}, \mathsf{sk}_{\texttt{EG}}) \leftarrow \texttt{EG.KeyGen}(1^\lambda)$
2 : $(\mathsf{pk}_{\texttt{Kyber}}, \mathsf{sk}_{\texttt{Kyber}}) \leftarrow \texttt{Kyber.KeyGen}(1^\lambda)$
3 : $\mathsf{sk} \leftarrow (\mathsf{sk}_{\texttt{EG}}, \mathsf{sk}_{\texttt{Kyber}})$
4 : $\mathsf{pk} \leftarrow \mathsf{pk}_{\texttt{EG}} \parallel \mathsf{pk}_{\texttt{Kyber}}$
5 : **return** $(\mathsf{pk}, \mathsf{sk})$

La fonction `Encaps` combine celles des KEM existants. On utilise la méthode présentée dans [Giacon *et al.*, 2018] pour assembler les deux KEM. On concatène les chiffrés (envoyés par la suite sur le réseau) comme on l'a fait pour les clés publiques afin de préserver l'anonymité.

$\texttt{KEM}.\mathrm{Encaps}(\mathsf{pk})$

1 : $\mathsf{pk}_{\texttt{EG}} \parallel \mathsf{pk}_{\texttt{Kyber}} \leftarrow \mathsf{pk}$
2 : $(c_{\texttt{EG}}, K_{\texttt{EG}}) \leftarrow \texttt{EG.Encaps}(\mathsf{pk}_{\texttt{EG}})$
3 : $(c_{\texttt{Kyber}}, K_{\texttt{Kyber}}) \leftarrow \texttt{Kyber.Encaps}(\mathsf{pk}_{\texttt{Kyber}})$
4 : $K \leftarrow W(K_{\texttt{EG}}, K_{\texttt{Kyber}})$
5 : $c \leftarrow c_{\texttt{EG}} \parallel c_{\texttt{Kyber}}$
6 : **return** (c, K)

Enfin, le déchiffrement réassemble les méthodes existantes :

$\texttt{KEM}.\mathrm{Decaps}(\mathsf{sk}, c)$

1 : $(\mathsf{sk}_{\texttt{EG}}, \mathsf{pk}_{\texttt{Kyber}}) \leftarrow \mathsf{sk}$
2 : $c_{\texttt{EG}} \parallel c_{\texttt{Kyber}} \leftarrow c$
3 : $K_{\texttt{EG}} = \texttt{EG.Decaps}(\mathsf{sk}_{\texttt{EG}}, c_{\texttt{EG}})$
4 : $K_{\texttt{Kyber}} = \texttt{Kyber.Decaps}(\mathsf{sk}_{\texttt{Kyber}}, c_{\texttt{Kyber}})$
5 : **return** $K \leftarrow W(K_{\texttt{EG}}, K_{\texttt{Kyber}})$

Les clés publiques et les chiffrés générés par ce KEM sont ainsi indistinguables d'un tirage uniforme comme concaténation de chaînes qui le sont. Le protocole CAKE peut donc être employé avec ce KEM.

La fonction $W : (K_{\texttt{EG}}, K_{\texttt{Kyber}}) \mapsto K$ est appelée *KEM combiner*. Une solution décrite dans [Petcher et Campagna, 2023] est de prendre une valeur hachée des clés, c'est-à-dire :

$$W(K_{\texttt{EG}}, K_{\texttt{Kyber}}) = \texttt{sha256}(K_{\texttt{EG}} \parallel K_{\texttt{Kyber}}).$$

Il est à noter que si cette méthode renforce la sécurité du PAKE vis-à-vis des failles touchant les KEM (qui permettraient d'acquérir sk par exemple), la surface d'attaque est potentiellement augmentée en cas de défaillance de la *fuzziness* ou de l'*anonymity* d'un des KEM. En effet, il suffirait qu'une partie de la chaîne échangée soit discernable d'un tirage aléatoire pour que des informations sur la clé symétrique (et donc `pw`) soient déductibles. Ce compromis nous semble raisonnable dans la mesure où les démonstrations de ces propriétés font appel à des preuves plus simples dans des modèles moins complexes.

Nous n'avons pas eu le temps d'ajouter l'hybridation à nos implémentations en C et en Python, mais nous aimerions le faire dans les prochains jours.

4 Implémentation Python

Notre implémentation de CAKE et OCAKE en Python est disponible publiquement à l'adresse : `https://github.com/pq-pake`.

4.1 Python comme preuve de concept, C pour aller vers des applications industrielles

Nous avons choisi d'utiliser deux langages de programmation différents pour notre projet. Les premiers prototypes et essais ont été réalisés en Python. Ce langage permet une programmation rapide et efficace, et est particulièrement adapté pour les preuves de concept. De plus, étant orienté objet, il permet une bonne abstraction qui rend le code plus lisible et permet au lecteur d'adapter facilement le code à tout langage.

Cependant, Python est un langage interprété, ce qui le rend plus lent que les langages compilés. De plus, Python ne peut pas être exécuté (facilement) sur les cartes à puce, comme celles des cartes d'identité. C'est pourquoi nous avons décidé de créer une seconde implémentation en C. Ce langage a l'avantage d'être de bas niveau, ce qui permet de contrôler plus finement les ressources de la carte à puce ou de l'ordinateur.

! L'implémentation Python n'est prévue qu'à des fins de démonstration algorithmique, contrairement à l'implémentation C qui se rapproche de ce à quoi ressembleraient des applications industrielles futures.
De fait, l'implémentation Python dépend de la bibliothèque kyber-py, qui est présentée comme se limitant à une preuve de concept. En particulier, elle n'est pas résistante aux attaques temporelles.

4.2 Spécification de l'API Python

4.2.1 Utilisation de l'API

Notre implémentation de CAKE et OCAKE en Python est destinée à être utilisée par des développeurs souhaitant intégrer CAKE et OCAKE dans leurs applications. L'utilisateur de notre API pourra obtenir une *clé de session*, c'est-à-dire un secret de 256 bits partagé par les deux ordinateurs, qu'il peut ensuite utiliser comme clé de chiffrement symétrique pour sécuriser ses communications (avec AES par exemple).

Pour obtenir cette clé de session, l'utilisateur pourra recourir à la classe **AliceCake** sur un des appareils, et à **BobCake** sur l'autre (ou **AliceOCake** et **BobOCake** s'il souhaite plutôt utiliser le protocole OCAKE).

L'utilisation de notre API CAKE est décrite dans la figure 9. L'utilisation de OCAKE est similaire, mais nécessite d'envoyer `bob.auth_verifier` à Alice en même temps que `bob.encrypted_ciphertext` et `bob.name`.

4.2.2 Structure de l'implémentation

L'utilisateur final utilise uniquement les classes `AliceCake, BobCake, AliceOCake` et `BobOCake`. Cependant, pour éviter la duplication de code et pour améliorer sa maintenabilité, notre implémentation est structurée comme le montre la figure 10. Chaque classe `Interlocutor` implémente les méthodes communes aux classes `AliceCake` et `BobCake` (ou `AliceOCake` et `BobOCake`). Par exemple, la génération de la clé symétrique finale, la clé de session, est implémentée dans la classe

Alice		Bob

`alice = Alice(`
` int(0).to_bytes(),`
` b'183754')`
`alice.generate_keypair()`
		`bob = Bob(`
		` int(0).to_bytes(),`
		` b'183754')`

	`alice.encrypted_public_key, alice.name` →	

		`bob.generate_symmetric_key(`
		` alice.encrypted_public_key,`
		` alice.name)`

	← `bob.encrypted_ciphertext, bob.name`	

`alice.decrypt_ciphertext(`
` bob.encrypted_ciphertext,`
` bob.name)`		

`alice.session_key`		`bob.session_key`

Figure 9 – Utilisation pratique de notre API Python entre deux appareils : Alice et Bob. Les flèches représentent l'envoi de données entre les deux appareils. (Dans le cas de CAKE)

`Interlocutor`. La génération du `auth_verifier`, spécifique à OCAKE, est implémentée dans la classe `InterlocutorOCake`.

```python
class AliceCake(InterlocutorCake):
    def __init__(
        self,
        session_id: bytes,
        password: bytes,
        name: bytes = "Alice".encode("utf-8"),
        debug: bool = True,
    ) -> None:
```

Les classes `AliceCake`, `BobCake`, `AliceOCake` et `BobOCake` sont instanciées par l'utilisateur final avec les arguments ci-dessus. Le `session_id` est un compteur qui permet de garantir l'unicité de la clé de session, il doit être incrémenté à chaque nouvelle session. Le `password` est le mot de passe partagé entre Alice et Bob. Le `name` est le nom propre de l'interlocuteur. Enfin, le paramètre `debug` permet d'afficher des informations de débogage si nécessaire.

Le chiffrement symétrique que nous avons choisi (le chiffrement de Feistel, dont le fonctionnement est détaillé dans la partie 2.4) est implémenté dans le fichier `ideal_cipher/feistel.py`. Il est important de noter que l'utilisation de `ideal_cipher.feistel.encrypt` et `ideal_cipher.feistel.decrypt` directement dans CAKE n'aurait pas été sécurisée.

En effet, avant de chiffrer la clé publique générée par `Kyber.KeyGen`, il est nécessaire de l'encoder en utilisant la méthode que nous avons mise au point dans la partie 3.2. Cette

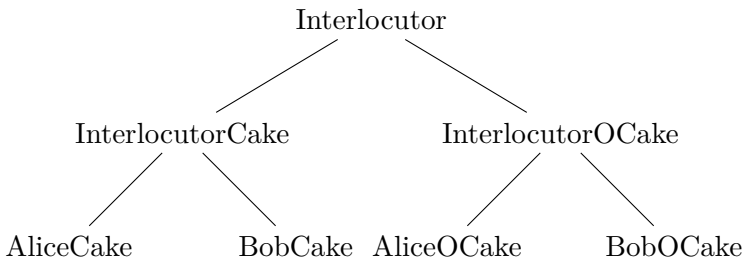

Figure 10 – Structure des classes permettant l'implémentation de CAKE et OCAKE en Python.

méthode est implémentée dans la classe `ideal_cipher.compact_encoder.CompactEncoder`. La clé publique peut alors être chiffrée en utilisant la méthode `ideal_cipher.public_key.encrypt(public_key: bytes, symmetric_key: bytes) -> bitarray`.

Nous avons décidé d'utiliser la bibliothèque `bitarray` pour manipuler les bits dans l'implémentation de notre chiffrement de Feistel. En effet, le type natif Python `bytes` ne permet que de manipuler des octets (groupes de 8 bits). Or, notre chiffrement de Feistel ainsi que notre encodage manipulent directement les bits.

Nous avons mis au point plusieurs tests unitaires pour nous assurer du bon fonctionnement de notre implémentation. Ces tests sont disponibles dans le fichier `tests.py` et peuvent être exécutés en suivant les instructions du fichier `README.md`.

5 Implémentation C et démonstrateur

5.1 Problématiques rencontrées

5.1.1 Choix du langage de programmation

Comme expliqué précédemment, nous avons effectué notre première implémentation avec le langage Python. Celui-ci présente de nombreux et précieux avantages lors de la création du prototype car sa syntaxe et son haut niveau d'abstraction nous ont permis de nous focaliser dans un premier temps sur les difficultés cryptographiques. Cependant, pour des applications embarquées, Python est sous-optimal. En effet, Python est notoirement lent et gourmand en mémoire. C'est un langage interprété et non compilé et donc, en réalité, lancer un programme Python revient à lancer un interpréteur qui se charge lui-même de lire et d'exécuter le fichier fourni. Ainsi une implémentation écrite purement avec ce langage est moins optimisée qu'un langage compilé. L'expérience montre que pour des applications intensives en ressources CPU, un programme écrit en Python est généralement plus lent que son équivalent en C de plusieurs ordres de grandeur.

Outre le fait d'appréhender et d'utiliser correctement les protocoles PAKE, un des objectifs de ce projet est de proposer une implémentation performante et robuste de ces protocoles qui puisse convenir à des usages industriels. Cela inclut par exemple une utilisation dans des documents d'identité comme les cartes nationales d'identité ou les passeports, ou encore dans des lecteurs de cartes utilisés par les autorités pour lire les informations contenues par ces documents. Sur la suggestion de notre tutrice, notre intérêt s'est porté d'abord sur les circuits logiques programmables (FPGA pour *Field Programmable Gate Array*) qui utilisent le langage VHDL. Cependant, développer sur cette plateforme nécessite un apprentissage spécifique, ainsi que du matériel que nous ne possédions pas et avec lequel nous n'étions pas familier. Plus problématique encore, une implémentation en VHDL ne serait vraiment utile que pour le matériel précité et serait difficilement adaptable à un usage sur d'autres plateformes. C'est pourquoi nous avons rapidement abandonné cette possibilité pour nous concentrer sur une implémentation en langage C.

En effet, celui-ci est un langage de programmation de bas niveau qui permet de contrôler finement les ressources de la machine. Il est compilé et non interprété, ce qui signifie que le code source est transformé en langage machine par un compilateur avant d'être exécuté. On obtient alors des performances bien supérieures à celles de Python. Le langage C est par ailleurs très répandu et très utilisé dans l'industrie et ce malgré la montée en popularité d'autres alternatives visant le bas niveau comme C++ ou plus récemment Rust. Cela signifie que le langage C est bien documenté, qu'il existe de nombreux outils pour le développer et le déboguer et que de nombreux développeurs en sont familiers. Une implémentation dans ce langage peut donc être plus facilement partagée, maintenue et distribuée.

Cela ne rend pas la tâche facile pour autant. En effet, C est un langage de bas niveau qui nécessite de gérer manuellement la mémoire et les ressources de la machine. Cela fait de lui une source infinie de bugs [6], de failles de sécurité [7], de fuites de mémoire [8] et de comportements inattendus [9]. C'est pourquoi une vigilance particulière est nécessaire lors du développement en C. Cela signifie que le code doit être

6. Problèmes courants en C et C++, liste compilée par le *static analyzer* PVS-Studio : `https://pvs-studio.com/en/docs/warnings/`

7. Par exemple la faille OpenSSL Heartbleed en 2014, `https://www.cert.ssi.gouv.fr/actualite/CERTFR-2014-ACT-015/`

8. Par exemple il y a quelques semaines sur les produits de sécurité Palo Alto : `https://www.cert.ssi.gouv.fr/avis/CERTFR-2024-AVI-0295/`

9. Une liste compilé par J. Regehr de manipulations d'entiers non définies dans les différentes spécifications mais utilisées par des bibliothèques cryptographiques populaires : `https://blog.regehr.org/archives/1054`

testé et validé de manière rigoureuse pour s'assurer de sa fiabilité. Le code doit également être écrit de manière claire et lisible pour faciliter sa maintenance et son évolution.

Enfin, certains membres du groupe avaient déjà une expérience en C, ou l'ont acquise dans le cadre du MODAL INF473X à l'École. Cela a entériné notre choix de langage pour l'implémentation finale des protocoles PAKE.

Cependant les choix ne se sont pas arrêtés au choix du langage. Cette décision a été, bien au contraire, dimensionnante pour les types de problèmes auxquels nous nous sommes confrontés. Nous expliquons ces problèmes dans les paragraphes suivants.

5.1.2 Gestion de la mémoire

La gestion de la mémoire est fondamentalement différente entre C et Python. En Python, la mémoire est gérée automatiquement par l'interpréteur par le biais d'un *garbage collector*. Cela signifie que l'interpréteur alloue et libère automatiquement la mémoire nécessaire pour les objets créés par le programme. Cependant, cela signifie que l'on ne peut pas contrôler finement la mémoire et que l'on ne peut pas savoir exactement quand celle-ci est allouée ou libérée. Cela se fait évidemment au détriment des performances. En C en revanche, la gestion de la mémoire est à la charge du programmeur. Il doit manuellement déclarer les variables, les allouer et les libérer explicitement lorsqu'il n'en a plus besoin. On obtient un contrôle fin de la mémoire et on sait exactement quand la mémoire est allouée et libérée. Cette gestion plus fine de la mémoire (d'aucun dirait « plus laborieuse ») nous donne un meilleur contrôle sur les performances du programme et sur la gestion des ressources de la machine. Cependant, cela signifie également que le programmeur doit être plus vigilant et plus rigoureux dans la gestion de la mémoire pour éviter les fuites ou les corruptions de mémoire.

La gestion de la mémoire, en particulier de celle allouée sur le tas, est une source très importante de problèmes car le langage ne donne pas de garantie sur la validité des accès mémoires (problèmes de type *use after free*, *double free*, etc). Cela peut mener à des bugs très difficiles à trouver et à corriger. Allouer la mémoire sur la pile est aussi *de facto* plus rapide que sur le tas. C'est pourquoi, lorsque la taille des données est connue à la compilation, nous avons privilégié l'allocation sur la pile. C'est le cas par exemple pour les clés secrètes et les clés publiques qui sont de taille fixe et connue à la compilation. En revanche, lorsque la taille des données est inconnue à la compilation, nous avons utilisé l'allocation sur le tas comme pour les messages échangés entre les deux parties du protocole qui peuvent être de taille variable (elle dépend, entre autres, de la taille des noms des deux parties).

Après avoir pris ces précautions nécessaires, il nous est possible de profiter de cet accès direct à la mémoire et de manipuler les données de manière plus efficace. Cela nous a été particulièrement utile pour les opérations d'*ideal cipher*, en particulier les fonctions de chiffrement et de déchiffrement de Feistel mais aussi les fonctions d'encodage et de décodage de la clé publique. En effet, ces fonctions nécessitent de manipuler des données de taille importante et de les transformer de manière précise. Ainsi en Python, nous devons nous résoudre à ce genre d'expressions, élégantes et concises mais peu efficaces en termes de performances :

```
def encode(self, message: bytes) -> bitarray:
  # ...
  for i in range(self.nb_coefficients):
      coefficients[i] = sum(
          bits[i * self.coefficient_bits + j] << j
          for j in range(self.coefficient_bits)
      )
```

```python
    # ...

def decode(self, message: bitarray) -> bytes:
    # ...
    associated_sum = sum(message[-i - 1] << i for i in range(len(message)))
    # ...
```

En C, nous avons pu écrire des fonctions plus véloces, en particulier en évitant les boucles et en utilisant des opérations bit à bit pour manipuler les données. Par exemple, voici les fonctions d'encodage et de décodage de l'*ideal cipher* en C :

```c
void pake_ic_encode(const uint8_t* input, uint8_t* output) {
  // ...
  for (int i = 0; i < COEFF_HALF_SIZE; i++) {
    coefficients[2 * i] =
        (uint16_t)input[3 * i] | ((uint16_t)(input[3 * i + 1] & 0x0F) << 8);
    coefficients[2 * i + 1] =
        (uint16_t)(input[3 * i + 1] >> 4) | (uint16_t)input[3 * i + 2] << 4;
  }
  // ...
}

void pake_ic_decode(const uint8_t* input, uint8_t* output) {
  // ...
  for (int i = 0; i < COEFF_HALF_SIZE; i++) {
    output[3 * i] = coefficients[2 * i] & 0xFF;
    output[3 * i + 1] =
        (coefficients[2 * i] >> 8) | ((coefficients[2 * i + 1] & 0x0F) << 4);
    output[3 * i + 2] = (coefficients[2 * i + 1] >> 4) & 0xFF;
  }
  // ...
}
```

On s'est ainsi rendu compte que les opérations que nous faisions en Python sur les bits consécutifs peuvent se réécrire assez simplement en C en manipulant 16 ou 24 bits à la fois, ce qui correspond à deux ou trois octets consécutifs.

Plus généralement, les types d'abstraction comme `bitarray` que nous avons utilisés en Python pour rendre le code plus lisible et plus facile à maintenir ne sont pas toujours les plus performants. Ces types ne sont aussi évidemment pas disponibles en C, où tout est représenté par des entiers ou des tableaux d'entiers. Cela nous a obligé à repenser certaines parties du code et une grande partie du travail d'adaptation du protocole de Python à C a consisté à réécrire ces fonctions dans un langage offrant moins d'abstraction.

5.1.3 Choix de l'implémentation de Kyber

Dans notre implémentation Python, nous avons utilisé la bibliothèque kyber-py pour les opérations CRYSTALS-Kyber. Cette bibliothèque est une implémentation purement en Python et n'a

pas été conçue pour être utilisée dans un contexte de production. Pour notre implémentation en C, nous avons dû chercher une bibliothèque C proposant les opérations cryptographiques nécessaires tout en fournissant certaines garanties sur la robustesse de l'implémentation. Nos recherches nous ont finalement mené à choisir le projet PQClean[Kannwischer *et al.*, 2022]. Celui-ci propose des implémentations autonomes (*standalone*), éprouvées et propres de nombreux algorithmes post-quantiques à l'étude par le NIST dans le cadre de la standardisation future de la cryptographie post-quantique. Ceux-ci comprennent entre autres Kyber1024, saveur de la bibliothèque qui nous importe dans notre cas. Le projet met également l'emphase sur la qualité du code et la facilité d'utilisation des API mises à disposition, ce qui s'est révélé salutaire pour notre implémentation. Enfin, la réactivité des mainteneurs du projet, leur transparence sur l'historique des problèmes (de sécurité ou pas) résolus et leur réputation nous ont conforté dans notre choix.

Contrairement à d'autres bibliothèques comme GMP ou OpenSSL, l'utilisateur n'a pas à fournir lui-même la bibliothèque au moment de la compilation. En effet, étant donné que le projet PQClean est peu répandu, nous avons choisi de ne pas en faire une dépendance externe mais plutôt de l'inclure directement dans nos fichiers bibliothèques.

5.1.4 Manipulation de grands entiers

La manipulation de grands entiers a été un autre sujet délicat qui a émergé du passage de Python à C. En effet, Python supporte nativement les entiers de taille arbitraire. Bien que celui-ci représente la plupart des entiers sur une taille fixe (classiquement 32 bits) il passe (de façon transparente pour le développeur) à un stockage de taille arbitraire dès qu'ils dépassent la valeur maximale représentable. C'est là un des nombreux mécanismes que Python gère automatiquement pour le développeur mais que ne fait pas C. En effet, les entiers sont ici représentés en mémoire par des types de taille variable, allant de 8 bits (`unsigned char`) à 64 bits (`unsigned long long`). Mais C ne supporte pas nativement la manipulation d'entiers de taille arbitraire.

Cette fonctionnalité est primordiale pour l'encodage de la clé publique au tout début du protocole. À cet égard, il aurait été plutôt mal avisé de développer nous-même une bibliothèque de manipulation de grands entiers car c'est un problème courant qui est déjà pris en charge par plusieurs bibliothèques éprouvées par le temps et leur incorporation dans de nombreux projets. De plus, coder nous-même de telles fonctions alourdirait inévitablement le code final du projet et le rendrait mécaniquement moins facilement maintenable, ce qui n'est pas l'objectif recherché par notre PSC.

C'est ainsi que nous avons choisi d'utiliser la bibliothèque GMP (pour *GNU Multiple Precision*). C'est une bibliothèque libre dévelopée par le projet GNU dans le cadre du système d'exploitation GNU/Linux. La bibliothèque offre la possibilité de faire de nombreuses manipulations arithmétiques entre grands nombres (entiers, fractions, etc.) et nombres standards (entiers, flottants) et permet également d'importer et d'exporter les entiers manipulés.

Il est tout de même important de garder en tête que, bien que GMP soit une bibliothèque très performante et très bien maintenue, utiliser des entiers de taille arbitraire reste une opération coûteuse en temps de calcul et en mémoire. Il a donc été nécessaire d'utiliser cette bibliothèque avec parcimonie et de ne pas l'utiliser pour des opérations qui pourraient être faites avec des entiers de taille fixe. Ainsi, seules les fonctions `pake_ic_encode`, `pake_ic_decode` et `pake_ic_value_is_not_in_range` utilisent GMP pour manipuler les entiers de taille libre, tout en veillant à réutiliser les entiers déjà alloués, éviter des opérations inutiles et passer aux entiers standards dès que possible.

5.1.5 Fonctions de hachage

Les protocoles PAKE nécessitent des oracles aléatoires qui, en pratique, sont approximés par des fonctions de hachage cryptographiques. En pratique, nous avons donc besoin d'utiliser les algorithmes SHA-256 et SHA-512. Pour les mêmes raisons que pour la manipulation de grands entiers, il n'est pas recommandé de coder nous-mêmes ces fonctions de hachage. En effet, ces fonctions sont des primitives cryptographiques qui sont utilisées dans de nombreux protocoles et qui ont été étudiées et éprouvées par de nombreux cryptanalystes. Le risque de coder nous-mêmes ces fonctions est de créer des failles de sécurité qui pourraient être exploitées par un attaquant pour compromettre le protocole. Alternativement, si une faille est découverte dans une bibliothèque de hachage, il est possible de la corriger et de mettre à jour la bibliothèque pour corriger le problème.

C'est pourquoi nous avons choisi d'utiliser la bibliothèque OpenSSL pour les fonctions de hachage. C'est la bibliothèque de référence pour un grand nombre de fonctions cryptographiques et elle est utilisée dans de nombreux projets et par de nombreux développeurs. Ce choix se fait donc de façon assez naturelle.

5.1.6 Politique de gestion des dépendances

Dans les trois derniers paragraphes, nous avons présenté des problématiques nécessitant l'ajout de dépendances à notre projet. Les trois bibliothèques présentées — PQClean, GMP et OpenSSL — sont les trois seules dépendances de notre projet. Il est important de noter qu'une dépendance peut être externe ou internalisée. Il a donc fallu choisir pour chaque dépendance si elle devait être externalisée ou internalisée.

Une dépendance externe est une dépendance qui n'est pas fournie avec le projet et qui doit être fournie par l'utilisateur final au moment de la compilation de son propre code. C'est le comportement par défaut des dépendances en C et elle présente plusieurs avantages. Tout d'abord, cela permet aux développeurs de ces bibliothèques de publier des nouvelles itérations de leur code (par exemple, des correctifs de sécurité) sans que nous ayons nous-mêmes à redistribuer une nouvelle version de notre propre logiciel. Au demeurant, l'utilisateur peut choisir l'implémentation qui lui convient le mieux : tant que les fonctions ont la même signature, le *linker* peut résoudre la dépendance et compiler les binaires en un exécutable. Enfin cette technique nous aide à réduire la taille du binaire distribué, en mutualisant les fonctions déjà présentes dans la bibliothèque externe. Cette option est particulièrement intéressante pour des dépendances populaires, présentes dans la plupart des systèmes d'exploitation et facilement installables par l'utilisateur final. C'est le choix que nous avons fait pour GMP et OpenSSL.

Une dépendance internalisée, quant à elle, est fournie avec le projet et est compilée en même temps que le reste du code. Cela permet de spécifier une version particulière de la bibliothèque et de garantir que le code fonctionne avec cette version. Il en découle aussi que nous ne dépendons pas de la disponibilité de la bibliothèque sur le système de l'utilisateur final. C'est le choix que nous avons retenu pour PQClean car la bibliothèque est peu répandue et nous ne voulions pas que l'utilisateur final ait à installer une bibliothèque supplémentaire pour utiliser notre code, participant de fait à l'adoption de notre implémentation.

5.1.7 Convention de nommage

Il est utile de noter qu'il n'existe pas d'espace de nom en C. Cela signifie que toutes les fonctions, les variables et les types définis dans un fichier C sont visibles par tous les autres fichiers qui l'incluent.

Cette inclusion est même transitive lorsqu'elle est faite sur un fichier d'en-tête. Ainsi, afin de ne pas créer des conflits de noms entre les différentes bibliothèques et de ne pas polluer l'espace de noms global, il est d'usage de préfixer les noms des fonctions, des variables et des types avec un préfixe unique. Cela nous assure que les noms utilisés dans notre code ne sont pas déjà utilisés par une autre bibliothèque et que notre code interne ne sera pas utilisé par une autre bibliothèque. Cela permet également de faciliter la lecture du code en indiquant clairement quelles fonctions, variables et types appartiennent à notre code et lesquels appartiennent à une bibliothèque externe. C'est pourquoi nous avons choisi de préfixer les noms de nos fonctions et de nos types présents dans nos fichiers d'en-tête par `pake_` pour indiquer qu'ils appartiennent à notre implémentation des protocoles PAKE. Nos macros en en-tête sont, elles, préfixées par `PAKE_`.

5.2 Spécification de l'API C et publication

Lors de l'élaboration de la spécification de notre bibliothèque, un des enjeux principaux a été de rendre les protocoles de PAKE post-quantiques les plus transparents possibles pour l'utilisateur final. A priori, il n'a pas besoin de connaître les détails de l'implémentation et ne doit pas se soucier du contenu de la mémoire des agents ou des messages échangés. Cela ne signifie pas pour autant que ces détails doivent lui être cachés à tout prix, bien au contraire. En effet, la transparence de l'implémentation est un gage de sécurité, car elle permet à l'utilisateur de vérifier que les protocoles sont bien implémentés et qu'ils ne contiennent pas de failles de sécurité. Cependant, celui-ci ne doit pas être obligé de comprendre les détails de l'implémentation pour pouvoir l'utiliser.

Nous avons donc décidé de publier notre bibliothèque en open-source, sous licence MIT. Toutes les sources de notre bibliothèque sont disponibles sur notre dépôt GitHub à l'adresse suivante : `https://github.com/pq-pake`.

Notre bibliothèque s'utilise en générant des *agents* qui génèrent des *messages* à partir de leur état interne. Ces messages sont ensuite envoyés à l'autre agent, qui les traite pour mettre à jour son propre état interne. Les agents sont ensuite capables de générer des messages suivants, jusqu'à ce que le protocole soit terminé. Les agents peuvent enfin générer des clés symétriques à partir de la clé secrète partagée.

La transmission des messages entre les deux agents est à la charge de l'utilisateur, ce qui rend le protocole agnostique du protocole de communication utilisé.

5.2.1 Structure de la bibliothèque

La bibliothèque pq-pake est divisée en trois fichiers d'en-tête. Un premier fichier contient les définitions des constantes et les deux autres sont les API pour les protocoles CAKE et OCAKE respectivement.

Le fichier d'en-tête `pq-pake/constants.h` contient les constantes utilisées en interne par la bibliothèque. Il définit également la valeur de `PAKE_SHARED_SECRET_SIZE` qui est la taille de la clé secrète partagée par les agents et utilisable par l'utilisateur.

Les fichiers d'en-tête `pq-pake/cake.h` et `pq-pake/ocake.h` contiennent les définitions des structures utilisées pour les agents et les messages des protocoles CAKE et OCAKE respectivement. Ils contiennent également les fonctions permettant de manipuler ces structures.

5.2.2 Évolutions futures

La bibliothèque a été conçue de sorte que de nouveaux protocoles ou de nouvelles versions des protocoles existants puissent être facilement ajoutés. En effet, ces derniers sont implémentés de manière modulaire, ce qui permet de les ajouter ou de les modifier sans affecter le reste de la bibliothèque. De plus, chaque message généré commence par un champ `protocol` d'un octet qui permet d'identifier celui qui est utilisé. Ce champ est notamment utilisé pour vérifier que les messages reçus sont bien ceux attendus. Pour le moment, seuls deux protocoles (`PAKE_PROTO_CAKE_KYBER1024` et `PAKE_PROTO_OCAKE_KYBER1024`) sont enregistrés dans la bibliothèque.

5.3 Choix techniques pour le démonstrateur

Notre implémentation se doit de fonctionner sur des appareils très peu puissants : des cartes d'identité et des petits appareils électroniques. Pour s'assurer que notre implémentation est bien utilisable en conditions réelles, nous avons décidé de faire un démonstrateur.

Notre objectif est de créer une « carte d'identité » ainsi qu'un poste pour lire la carte (qui appartiendrait à la douane en pratique par exemple).

5.3.1 Le choix du matériel

Nous avons d'abord essayé d'utiliser une carte à puce *ATmega328 CH340* compatible Arduino. Nous avons finalement compris que cette carte n'avait pas assez de mémoire vive pour notre usage : elle n'a que 2 Ko de RAM, alors que la clé publique de *Kyber* fait à elle seule 1,6 Ko. Cet échec a permis de mettre en lumière une limite à l'utilisation des PAKE. Il faut cependant noter que les puces utilisées dans les cartes d'identités ou cartes bancaires possèdent plus de RAM et sont donc compatibles avec les PAKE.

Nous avons finalement décidé d'utiliser une *Raspberry Pi Zero 2 W* pour représenter la carte d'identité et un ordinateur classique pour le poste de douane. Ce modèle a l'avantage de posséder bien plus de mémoire RAM (512 Mo) ainsi que plusieurs antennes.

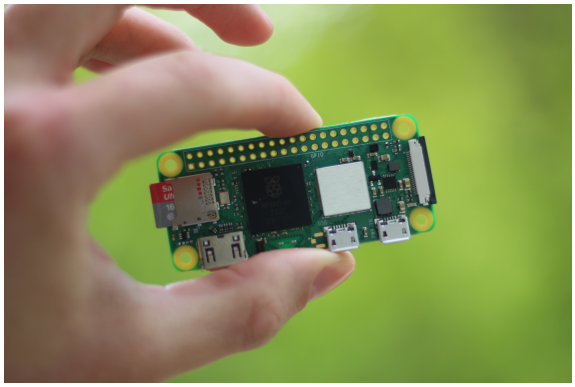

Figure 12 – Notre *Raspberry Pi Zero 2 W*, représentant une carte d'identité

5.3.2 Le choix du protocole de communication

Bien entendu, CAKE fonctionne indépendamment du protocole de communication. Les cartes d'identité utilisent le protocole NFC pour échanger avec le lecteur. Ce protocole a cependant été prévu en premier lieu pour de la lecture seule. Des solutions existent pour en faire un lecteur/receveur, mais

sont difficiles à mettre en œuvre. Pour notre implémentation, nous avions souhaité utiliser une antenne *NRF24L01* avec notre *ATmega328 CH340*. Cependant, puisque nous avons finalement dû passer à une *Raspberry Pi Zero 2 W*, nous avons décidé d'utiliser un autre protocole : le Bluetooth. La *Raspberry Pi* en contient déjà une, ce qui permet en outre de réduire le coût de ce démonstrateur.

Pour ne pas nous encombrer du matériel spécifique pendant la phase de débogage et de test, le code du démonstrateur fait abstraction du protocole de communication. Une macro permet, au moment de la compilation, de choisir entre une communication par socket Bluetooth ou par un socket TCP.

Cependant les deux protocoles de communication ont, en pratique, des limitations de taille de paquet différentes. En effet alors que TCP supporte des paquets de 65 ko ce qui est plus qu'assez pour nos besoins, le Bluetooth ne supporte que des paquets d'environ 1 Ko. Sachant que nos paquets ont une taille d'environ 5 Ko, la couche de transport a donc dû être adaptée pour prendre en compte ces différences et nous permettre de transmettre des messages de taille arbitraire.

5.3.3 Mise en pratique

Nous avons utilisé notre implémentation en C de CAKE sur le *Raspberry Pi* ainsi que sur notre ordinateur. L'échange de clé s'effectue convenablement, les deux appareils peuvent ensuite communiquer par chiffrement symétrique AES.

Plus précisément, nous avons choisi l'algorithme de chiffrement symétrique AES256 GCM. L'utilisation d'une clé de 256 bits est nécessaire pour maintenir 128 bits de sécurité, à cause de l'attaque post-quantique de l'algorithme de Grover. [Jaques *et al.*, 2020]

Nous avons décidé d'utiliser le mode GCM (*Galois Counter Mode*) de AES pour sa sécurité reconnue ainsi que l'authentification des messages qu'il permet. Ainsi, si le message chiffré est modifié par un attaquant, il sera refusé par notre programme.

Les paramètres initiaux de AES256 GCM sont choisis parmi les paramètres disponibles sur les agents CAKE. Ainsi la clé symétrique est naturellement le résultat de l'échange de clé authentifié et le vecteur d'initialisation correspond aux premiers octets de la clé publique. En effet le vecteur d'initialisation n'a pas de contraintes de sécurité particulière, mis à part le fait qu'il doit être unique pour chaque message chiffré. Ainsi, le vecteur d'initialisation est choisi de manière déterministe à partir de la clé publique.

5.4 Benchmarks

Nous avons réalisé des mesures de performance pour comparer les différentes implémentations de CAKE et OCAKE. Nous avons utilisé un ordinateur portable équipé d'un processeur *Intel Core i7-8850H* cadencé à 2.60 GHz et de 16 Go de RAM. Chaque mesure a été réalisée $n = 500$ fois.

Algorithme	Temps moyen	Temps médian	Min	Max	Écart-type
CAKE Python	1250 ms	1239 ms	1090 ms	2062 ms	103 ms
CAKE C	73.93 ms	73.58 ms	68.62 ms	89.85 ms	2.90 ms
OCAKE C	73.32 ms	72.33 ms	69.11 ms	96.20 ms	3.54 ms

Table 1 – Performance des différentes implémentations de CAKE et OCAKE. Les valeurs représentent le temps de calcul cumulé des deux agents $n = 500$.

On observe sans surprise que l'implémentation en C est bien plus rapide que l'implémentation en Python, d'un facteur 17 environ. On observe également que les deux protocoles ont des performances

très similaires, ce qui est attendu puisque les deux protocoles sont très proches en termes de complexité algorithmique.

Message chiffré	Temps moyen	Temps médian	Min	Max	Écart-type
Cipher text	1.23 ms	1.22 ms	1.19 ms	2.90 ms	0.08 ms
Public key	70.54 ms	70.84 ms	61.15 ms	101.61 ms	5.31 ms

Table 2 – Performance des deux fonctions de chiffrement/déchiffrement $n = 500$

Nous avons également mesuré les performances des fonctions de chiffrement et de déchiffrement utilisées par les protocoles. Pour rappel, le chiffrement du *ciphertext* consiste à faire un seul tour de chiffrement de Feistel tandis que le chiffrement de la *public key* nécessite de faire plusieurs passages par le réseau de Feistel car le résultat doit être inférieur à une certaine borne. Pour obtenir la *public key*, on fait donc systématiquement $k = 54$ tours de Feistel afin de nous protéger des attaques temporelles.

On constate alors que la majorité du temps de calcul des protocoles est consacré au chiffrement de la clé publique. On remarque d'ailleurs que ce temps est environ 54 fois plus long que le temps de chiffrement du *cipher text*, ce qui est cohérent avec le nombre d'itérations du réseau de Feistel. Cela signifie également que le temps de vérification de l'appartenance au bon intervalle est relativement limité. Ce calcul utilise les grands entiers de la bibliothèque GMP.

On peut aussi noter que ce temps de calcul est réparti très inégalement entre les deux agents. En effet, le temps de chiffrement de la clé publique est effectué par l'agent qui envoie la clé publique (par exemple, la carte d'identité), tandis que le temps de déchiffrement est effectué par l'agent qui reçoit la clé publique (par exemple le lecteur de carte). Cela signifie que le temps de calcul est plus important pour l'agent qui, en théorie, a les contraintes les plus fortes en termes de puissance de calcul disponible. Ce fait n'est pas forcément un problème en pratique, car il peut aider à limiter naturellement la fréquence à laquelle un attaquant peut demander la génération de nouvelles clés publiques à la puce. Cela permet d'éviter les attaques *online*.

Alice

```
cake_agent* alice =
 cake_create_alice(
 session_id,
 password,
 strlen(password),
 alice_name,
 strlen(alice_name)
 );
cake_create_message_round1(
 alice, &alice_message,
 &alice_message_size
);
```

alice_message →

Bob

```
cake_agent* bob =
 cake_create_bob(
 session_id,
 password,
 strlen(password),
 alice_name,
 strlen(alice_name)
 );
```

```
cake_create_message_round2(
 bob, alice_message
 &bob_message,
 &bob_message_size
);
```

← bob_message

```
cake_create_message_round3(
 alice, bob_message
);
const uint8_t* session_key =
 cake_get_shared_secret(alice);
```

```
const uint8_t* session_key =
 cake_get_shared_secret(bob);
```

Figure 11 – On utilise ici le protocole CAKE mais on pourrait indifféremment utiliser le protocole OCAKE.

Conclusion

Ce projet scientifique collectif nous a permis de découvrir en profondeur le domaine de la cryptographie post-quantique. Nous avons travaillé tant sur les aspects théoriques comme les preuves cryptographiques ou les réseaux de Feistel, que sur les aspects pratiques avec l'implémentation de nos protocoles en Python et en C ou les enjeux d'hybridation pour proposer des échanges de clé sécurisés résistants aux attaques par ordinateur quantique. Nous avons également réalisé un démonstrateur physique pour illustrer la faisabilité de notre protocole. Ce projet nous a ainsi permis de nous familiariser avec les méthodes de travail en recherche scientifique.

La cryptographie post-quantique reste un domaine de recherche en plein essor, et peu de standards ont été établis à ce jour. Le NIST travaille toujours sur l'élaboration de nouveaux standards cryptographiques pour le monde post-quantique. CRYSTALS-Kyber a été récemment standardisé et d'autres algorithmes sont à l'étude pour l'être aussi. Notre travail pourrait alors servir de base aux acteurs industriels pour l'implémentation de protocoles post-quantiques sécurisés sur des cartes d'identité ou des passeports.

Références

[ANSSI, 2022] ANSSI (2022). ANSSI views on the Post-Quantum Cryptography transition.

[Avanzi *et al.*, 2021] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J., Schwabe, P., Seiler, G. et Stehlé, D. (2021). CRYSTALS-kyber : Algorithm specifications and supporting documentation.

[Beguinet *et al.*, 2023] Beguinet, H., Chevalier, C., Pointcheval, D., Ricosset, T. et Rossi, M. (2023). GeT a CAKE : Generic Transformations from Key Encaspulation Mechanisms to Password Authenticated Key Exchanges. Publication info : Published elsewhere. 21st International Conference on Applied Cryptography and Network Security (2023).

[Bellovin et Merritt, 1992] Bellovin, S. et Merritt, M. (1992). Encrypted key exchange : password-based protocols secure against dictionary attacks. *In Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, Oakland, CA, USA. IEEE Comput. Soc. Press.

[Boyko *et al.*, 2000] Boyko, V., MacKenzie, P. et Patel, S. (2000). Provably Secure Password-Authenticated Key Exchange Using Diffie-Hellman. *In* Preneel, B., éditeur : *Advances in Cryptology — EUROCRYPT 2000*, Lecture Notes in Computer Science, pages 156–171, Berlin, Heidelberg. Springer.

[Coron *et al.*, 2016] Coron, J.-S., Holenstein, T., Künzler, R., Patarin, J., Seurin, Y. et Tessaro, S. (2016). How to Build an Ideal Cipher : The Indifferentiability of the Feistel Construction. *Journal of Cryptology*, 29(1):61–114.

[Dachman-Soled *et al.*, 2015] Dachman-Soled, D., Katz, J. et Thiruvengadam, A. (2015). 10-Round Feistel is Indifferentiable from an Ideal Cipher. Publication info : Preprint. MINOR revision.

[Elgamal, 1985] Elgamal, T. (1985). A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472. Conference Name : IEEE Transactions on Information Theory.

[Giacon *et al.*, 2018] Giacon, F., Heuer, F. et Poettering, B. (2018). KEM Combiners. Publication info : A minor revision of an IACR publication in PKC 2018.

[Jaques *et al.*, 2020] Jaques, S., Naehrig, M., Roetteler, M. et Virdia, F. (2020). Implementing Grover Oracles for Quantum Key Search on AES and LowMC. *In* Canteaut, A. et Ishai, Y., éditeurs : *Advances in Cryptology – EUROCRYPT 2020*, volume 12106, pages 280–310. Springer International Publishing, Cham.

[Kannwischer *et al.*, 2022] Kannwischer, M. J., Schwabe, P., Stebila, D. et Wiggers, T. (2022). Improving Software Quality in Cryptography Standardization Projects. Publication info : Published elsewhere. Security Standardization Research 2022.

[Ministère de l'Intérieur et des Outre-mer, 2021] Ministère de l'Intérieur et des Outre-mer (2021). La puce de la nouvelle carte nationale d'identité.

[Petcher et Campagna, 2023] Petcher, A. et Campagna, M. (2023). Security of Hybrid Key Establishment using concatenation.

[Shor, 1997] Shor, P. W. (1997). Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484–1509. arXiv :quant-ph/9508027.