Étude et implémentation d'un schéma de chiffrement homomorphe

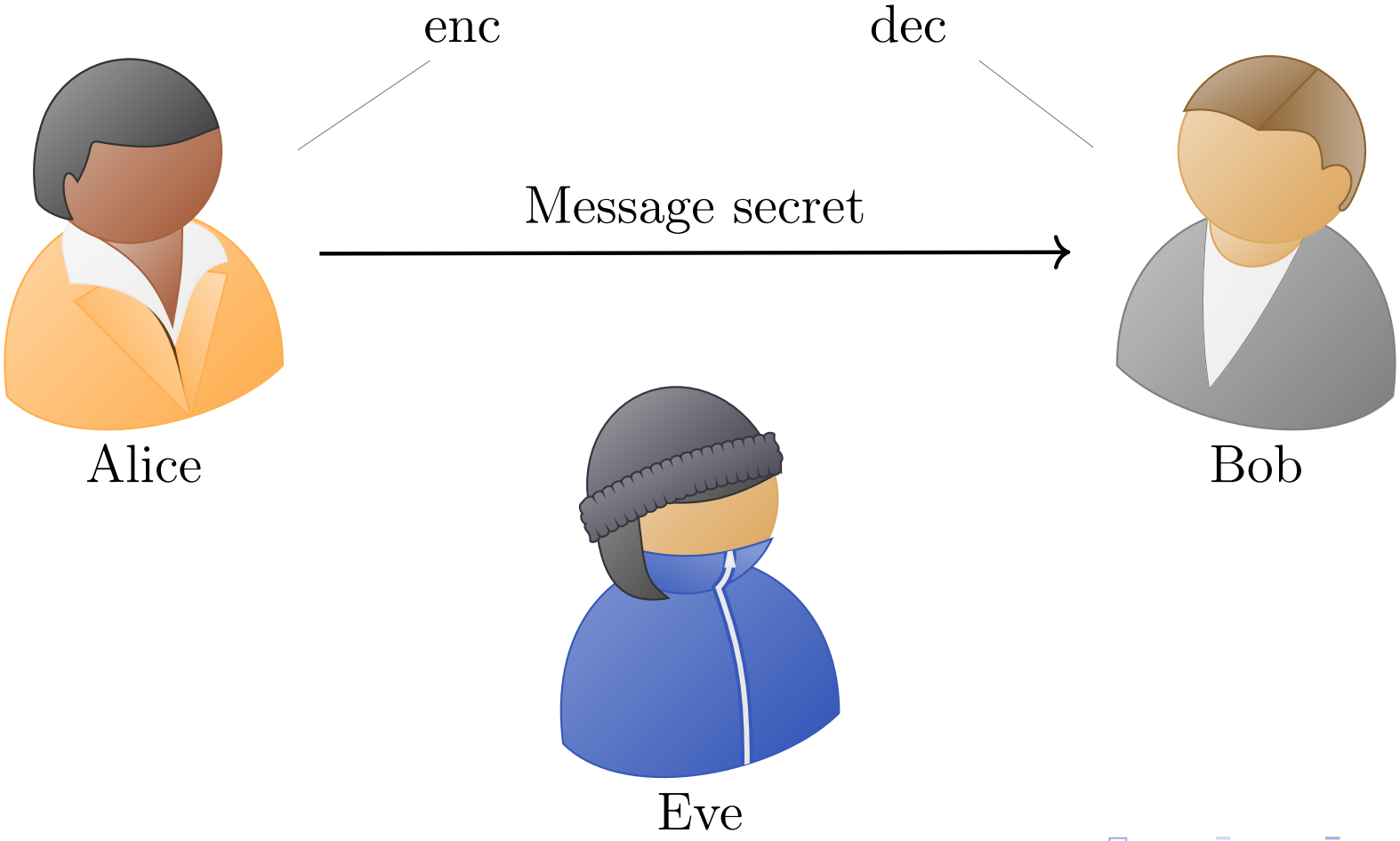
Milan Gonzalez-Thauvin

Thème : Transport

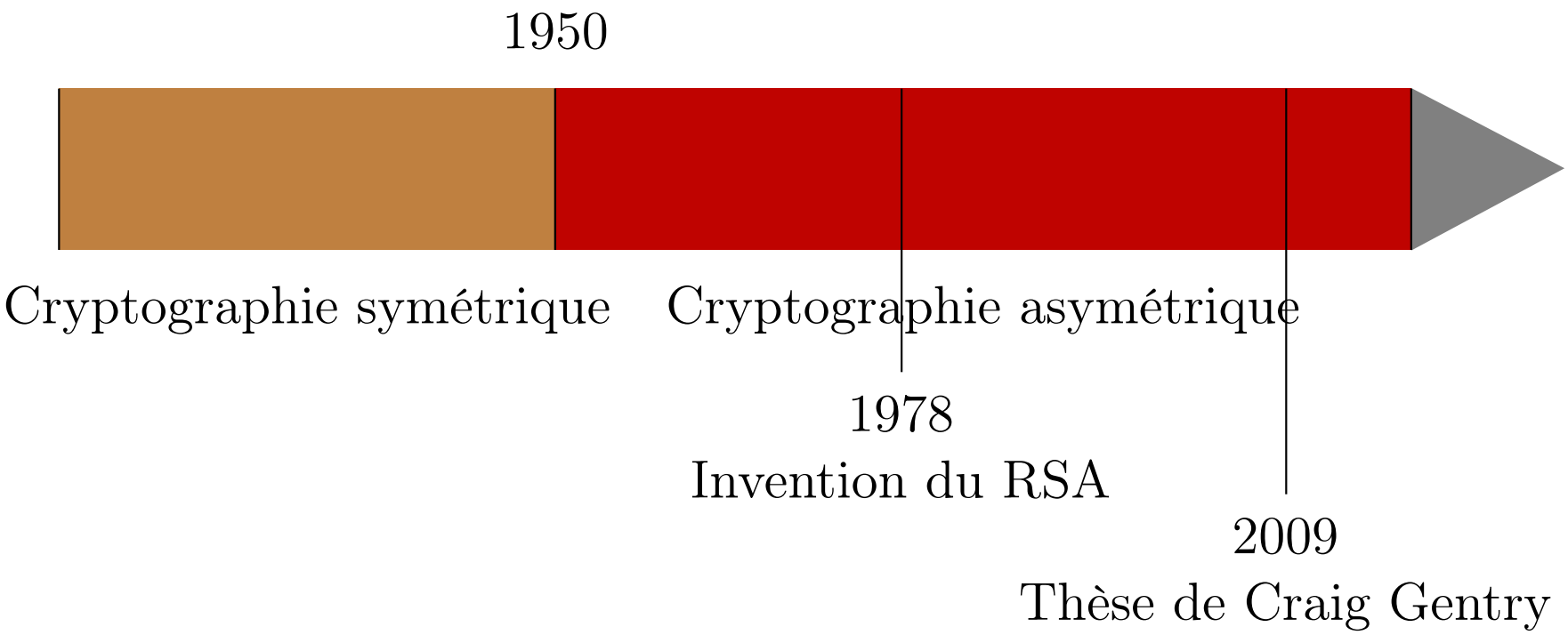
TIPE session 2019

Plan de l'exposé

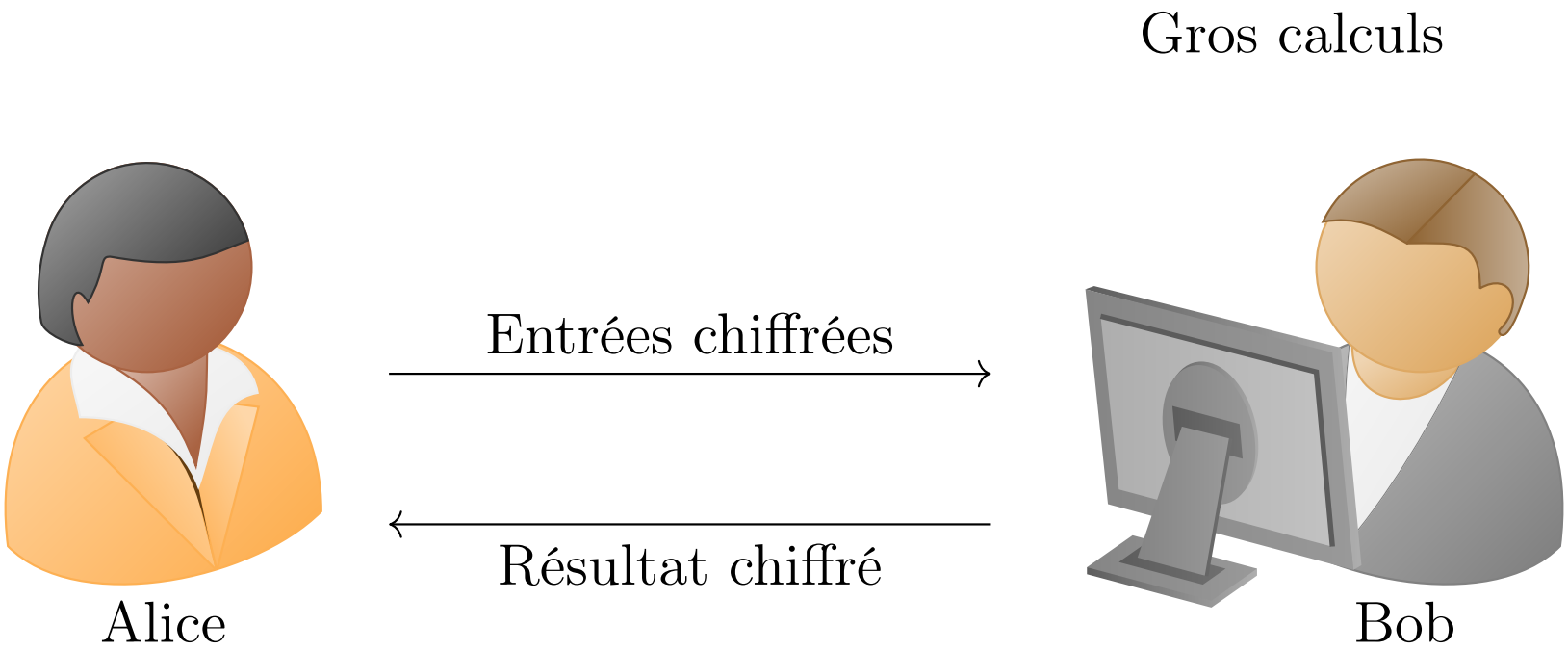
Introduction



Histoire


1950
Cryptographie symétrique
Cryptographie asymétrique
1978
Invention du RSA
2009
Thèse de Craig Gentry

La cryptographie homomorphe



La cryptographie homomorphe

Définition : Calculer sur des chiffrés

Si m est un mot **clair**, $c = \mathrm{enc}(m)$ son **chiffré**, et f une fonction, f est **compatible** pour le schéma si

$$\mathrm{dec}(f(c)) = f(m)$$

En pratique

Somme et **produit** car **polynôme**

La cryptographie homomorphe

Définition : Cryptographie partiellement homomorphe

On peut effectuer des additions **et/ou** des multiplications en nombre **fini**

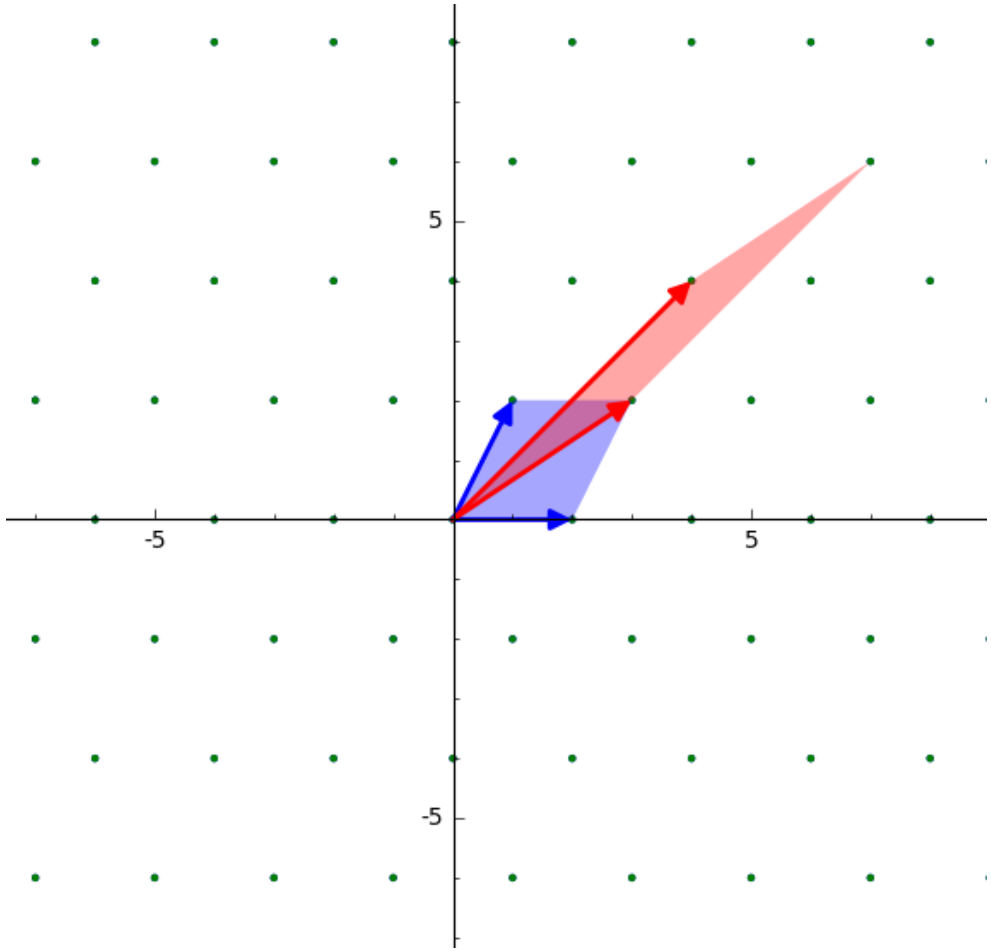
Définition : Cryptographie complètement homomorphe

On peut effectuer des additions **et** des multiplications en nombre **infini**

Pourquoi fait on la distinction ?

La cryptographie homomorphe se base sur du **bruit**.

La cryptographie homomorphe



État de l'art

Avant la Thèse de Craig Gentry

Cryptographie **partiellement homomorphe**

Thèse de Craig Gentry (2009)

Astuce : **bootstrap**

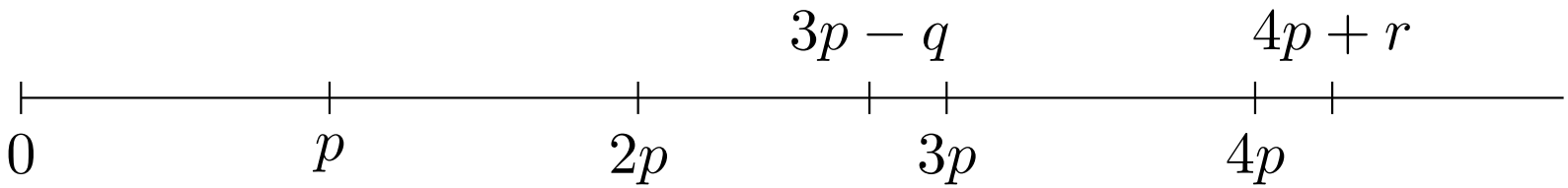
Depuis la thèse de Craig Gentry

Beaucoup de schémas **théoriques**
Aucune application concrète car **aucun n'est assez performant**

Le schéma de Jung Hee Cheon et Damien Stehlé : Fully Homomophic Encryption over the Integers Revisited

2016

Le schéma de Jung Hee Cheon et Damien Stehlé



Avec $0 < q \ll p$ et $0 < r \ll p$

Chiffrement sur les entiers

Problème du **PGCD approché**

Le schéma de Jung Hee Cheon et Damien Stehlé

Génération des clefs

clef **privée** : $sk = p$
clef **publique** : $pk = x_1, \ldots, x_\tau / \forall i \in [1, \tau], x_i \leftarrow pq_i + r_i$ avec x_0 le plus grand et $\frac{x_1}{2}$ impair.

Paramètres principaux

λ : Paramètre de sécurité
η : Nombre de bits de p
γ : Nombre de bits des x_i
ρ : Nombre de bits du bruit initial
τ : Nombre d'éléments dans la clef publique

Fonctions de base

Chiffrer

Soit $S \subset \{1, 2, \cdots, \tau\}$

$$c = \left[\sum_{i \in S} x_i + \left\lfloor \frac{x_1}{2} \right\rceil m \right]_{x_0}$$

Déchiffrer

$$m = \left[\left\lfloor \frac{2c}{p} \right\rceil \right]_2$$

Fonctions de base

Addition

$$c_{add} = [c_1 + c_2]_{x_0}$$

Multiplication

Procédé complexe

Implémentation (Python 3)

État de l'implémentation

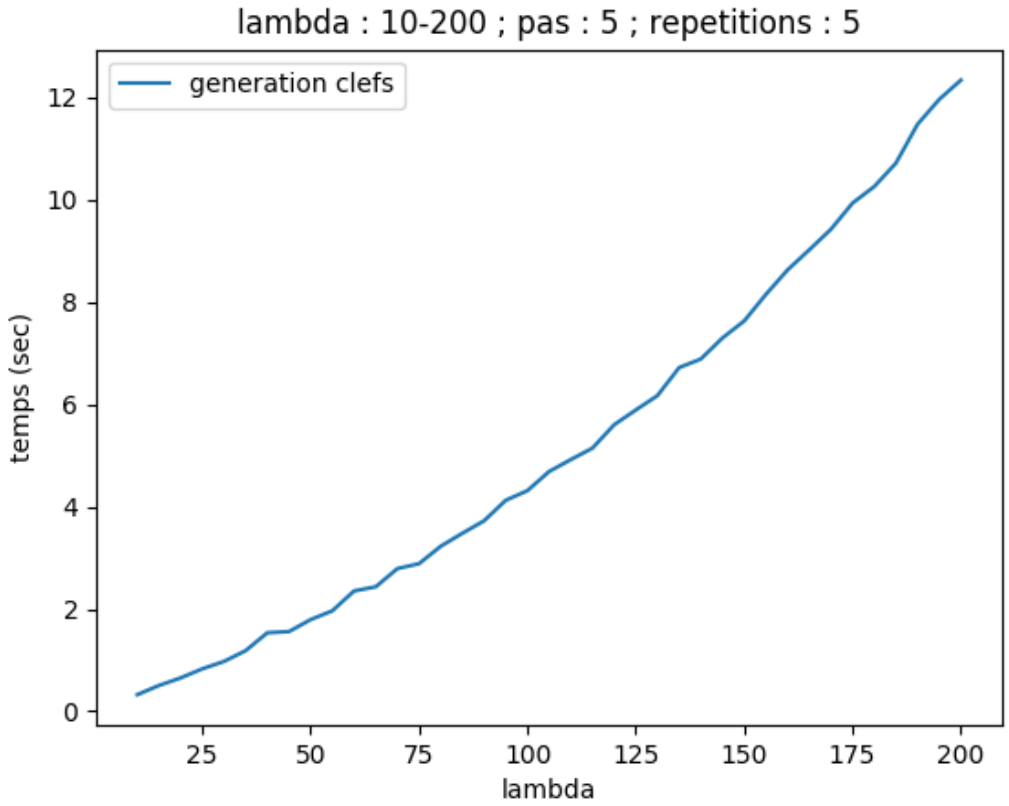
- Création des clefs : ✓
- Chiffrement : ✓
- Déchiffrement : ✓
- Somme : ✓
- Produit : ✓
- Bootstrap : Échec

Paramètres de base

Ceux proposés par l'article

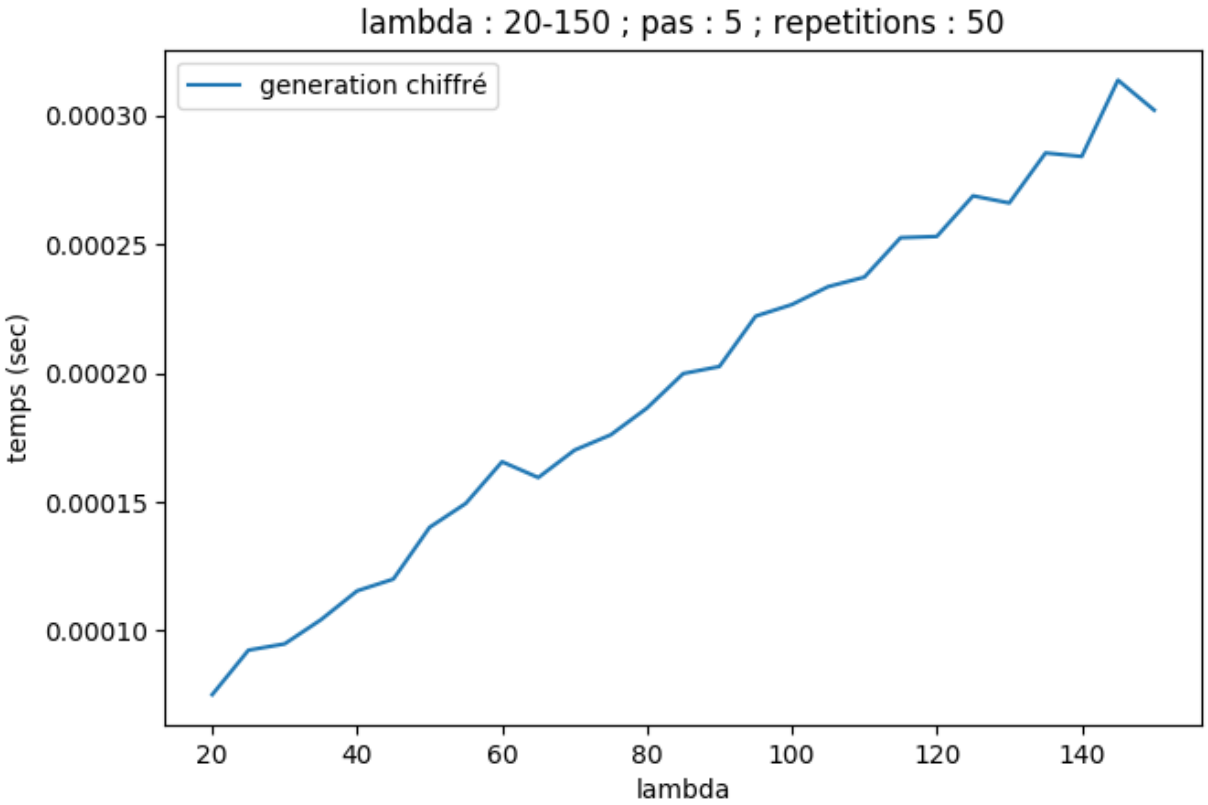
Constatation des performances

Génération des clefs



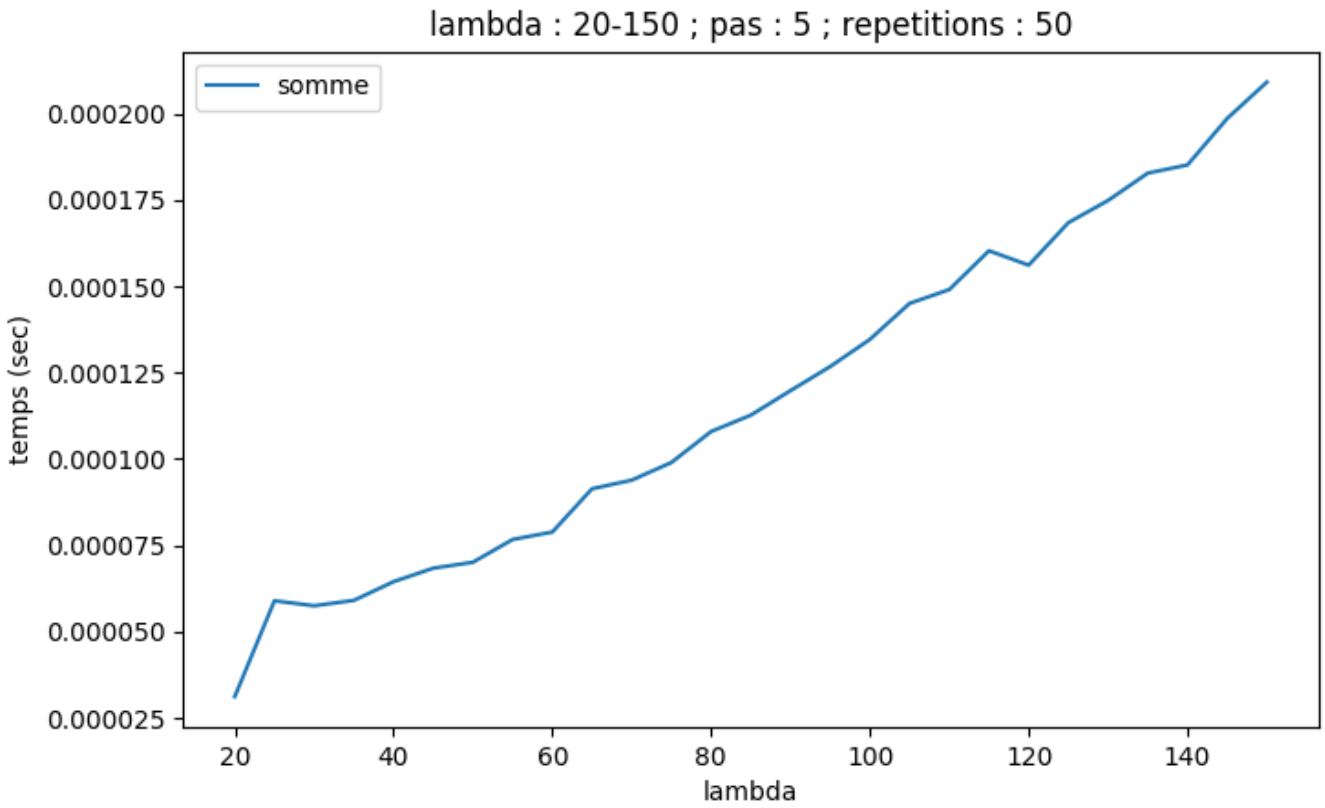
Constatation des performances

Génération des chiffrés



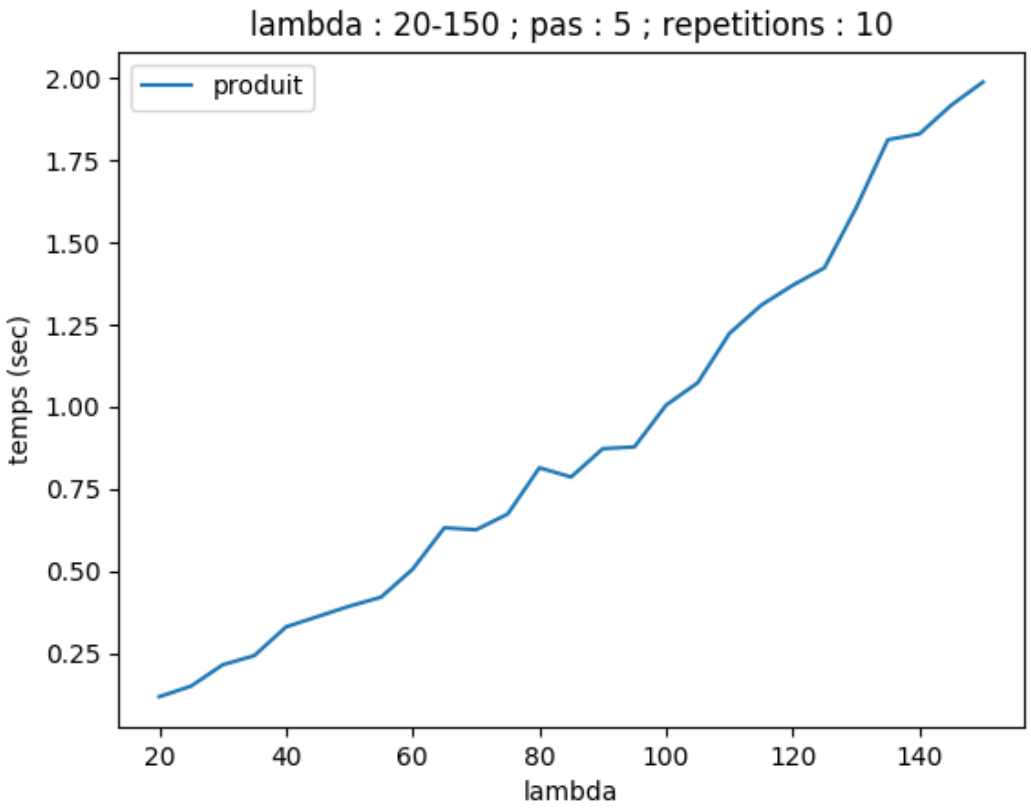
Constatation des performances

Somme



Constatation des performances

Produit



Constatation des performances

Nombre d'opérations

Nombre d'opérations

Dérisoire $\rightarrow$ 4 ou 5 multiplications bit à bit avant erreur

Le bootstrap

Échec

En cause :

- **Schéma trop théorique** et peu adapté à une implémentation
- **Évaluation homomorphique** des fonctions de chiffrement et déchiffrement **trop gourmande en calculs** et donc incompatible avec des paramètres pertinents pour un ordinateur

Améliorations

Code et Surcouche

But

- Simplifier l'**utilisation** du schéma
- Faciliter son **debugage** et son amélioration
- Permettre à d'**autres personnes** d'utiliser le schéma

Moyen

Tout un **écosystème** (notamment **opérations** bit à bit, **Classes** Python, fonctions regroupant les **opérations de base**)

Code et Surcouche

Résultat

Un code clair, concis, et facile d'utilisation

Code

```
1 lambdaa, nb_operations = 50, 1000
2
3 alice, bob = genererCS(lambdaa, nb_operations)
4 a = bob.chiffrerentier(5)
5 b = bob.chiffrerentier(4)
6 c = (a*b)
7 print(alice.dechiffrerentier(c))
```

Paramètres

Idée

Trouver les paramètres pour rendre le schéma :

- Plus **rapide**
- Plus **fiable** (possibilité d'effectuer plus d'opération)

pour un même paramètre de sécurité λ

Moyen

Fonctions permettant de tester et comparer les performances du schéma en faisant varier certains paramètres

Paramètres

Paramètres pertinents

$\rho = \lambda$
$\eta = \lfloor \rho + \log_2(\lambda) \rfloor$
$\gamma = \lfloor 1,05 * \eta \rfloor$
$\tau = \gamma + 2 * \lambda + 2$

Résultat

- **Sécurité** $\searrow$

+ **Fiabilité** $\nearrow$

Génération des clefs

Cause

- Nombre d'opérations faible (même après le changement de paramètres)
- Qualité des clefs aléatoire (dont certaines obsolètes)

Idée

Fonction qui génère de nombreuses paires de clefs jusqu'à en trouver une satisfaisante

Génération des clefs

Code

```
def genererCS(lambdaa,maxi):
    clef_invalide = True
    while clef_invalide:
        alice = Prive(lambdaa)
        bob = Publique(alice.publier())
        p, pc, j, prod_correct = 1, 1, 0, true
        while prod_correct:
            t = random.randint(0,1)
            p *= t
            pc *= bob.chiffrerbit(t)
            j += 1
            prod_correct = (p==alice.dechiffrerbit(pc))
            if j > maxi : return alice, bob
```

Génération des clefs

Résultat

- **Temps de génération des clefs** ↗

\+ **Fiabilité** ↗
\+ **Sécurité** ↗

Conclusion

Succès

- Schéma **partiellement homomorphe**
- Relativement **sûr** même si loin des normes actuelles
- **Fiable** dans la limite d'un bruit raisonnable

Échec

- Pas de **Bootstrap**
- Non utilisable à grande échelle car trop **lent** (plusieurs secondes pour chaque multiplication)

Conclusion

Utilisations possibles

Conditions :

- Peu de multiplications
- Des calculs dont on peut majorer la complexité

Prédilection :

- **Vote électronique**