Procédés cryptographiques pour le vote électronique

Orestis Ioannou

February 23, 2015

Ce document a pour but de vous exposer l'étude des différentes questions et problèmes auxquels nous faisons face, dans le domaine de la cryptographie, pour le vote électronique. Le rôle de cette étude était de se familiariser aux procédés cryptographiques utilisées pour affronter les problèmes liés au vote électronique, en utilisant les courbes elliptiques, du côté de la théorie ainsi que de la pratique. Cette étude a donné lieu à des modules qui peuvent servir pour chiffrer, déchiffrer mais aussi construire des preuves à connaissance nulle dans le domaine des courbes elliptiques. En outre, un système basique de vote sous forme d'une application web a été créé pour illustrer ces fonctionnalités.

Mots clés: Cryptographie; Vote électronique; Courbes elliptiques;

1 Introduction

Dans le cadre du second semestre de Master 1 Informatique à l'université Lyon 1 Claude Bernard, j'ai effectué un Travail d'Etude et de Recherche (TER). Cette étude, d'une durée de 6 semaines, avait pour thème les procédés cryptographiques dans le vote électronique et mon encadrant était Monsieur Fabien Laguillaumie, professeur à l'Institut de Science Financière et d'Assurances (ISFA) de l'université Claude Bernard.

L'objectif était d'étudier les solutions apportées aux questions et aux problèmes du vote électronique dans le domaine de la cryptographie et plus précisément en utilisant les courbes elliptiques. Entre autres, ce TER a permis d'étudier les questions de la confidentialité, vérifiabilité, incoercibilité notamment par le système de chiffrement ElGamal en utilisant des courbes elliptiques, les mix-nets et les preuves à connaissance nulle.

2 Contexte

Une des tâches les plus importante d'un gouvernement démocratique est la planification et l'exécution d'une élection qui désignera son successeur. C'est sans doute une tâche assez compliquée vue les contraintes strictes à respecter. Tout au long de l'histoire moderne nous avons reconnu des problèmes de résultats douteux, des pertes des bulletins, des votes des personnes non authentifiées (mortes aussi), de la technologie qui a échoué et des tentatives de fraude.

Cependant le vote n'est pas un processus moderne; nous connaissons déjà son existence depuis la Grèce ancienne ou les hommes votaient sur les "ostraca" pour ostraciser une personne de la communauté pendant 10 ans. Durant les années qui ont suivi la procédure est parvenue à plusieurs changements pour arriver aujourd'hui aux bulletins papiers. Il n'est donc pas surprenant que nous nous sommes intéressés maintenant au vote électronique. Il existe plusieurs protocoles sur internet mais cette étude utilise l'exemple du protocole conçu par M. Ben Adida grâce à sa bonne réception et son implémentation sous le nom de Helios.

2.1 Helios

Helios [1] est une plate-forme open-source pour créer des élections sur internet qui propose une solution aux questions de la confidentialité, vérifiabilité et incoercibilité. Il est basé sur la thèse de Ben Adida [2] et il est implémenté par Ben Adida, Olivier de Marneffe, Olivier Pereira avec les conseils de Jim Adler, Lawrence Lessig, Josh Benaloh, Andy Neff et Dan Wallach. Il compte aujourd'hui plus de 100.000 votes et il a vu sa plate-forme utilisée par plusieurs universités notamment par le Catholique de Louvain pour ses élections présidentielles.

Parmi les points forts de Helios les plus distincts sont:

1. la possibilité pour un électeur de suivre son vote et de s'assurer de sa bonne prise en compte par le serveur. Ceci signifie que personne ne peut modifier les votes, mêmes pas les administrateurs.

2. l'addition homomorphe (les votes ne sont pas déchiffrés un par un mais le seul déchiffrement effectué est à l'issue du vote)

3. l'utilisation de plusieurs clés pour effectuer le déchiffrement. Ceci permet de s'assurer qu'un administrateur ne peut pas déchiffrer le résultat ou un vote tout seul.

Helios utilise comme langage de programmation le Python pour le coté serveur et notamment Django comme web framework ainsi que du JavaScript pour le coté client. Ce projet, donc, est l'étude de plusieurs procédés utilisés dans cette plate-forme, et ses implémentations à l'aide des courbes elliptiques. Helios quant à lui implémente ElGamal dans les corps finis.

3 Étude théorique

Le but de cette section est de se familiariser avec les différents problèmes que nous rencontrons dans le vote électronique ainsi que la partie théorique de ses solutions.

3.1 Pourquoi voter en ligne est dure?

Le vote électronique est un problème qui définit plusieurs contraintes. D'une part nous avons besoin de la verifiabilité pour qu'un électeur puisse être sur que sa préférence a été bien prise en compte et d'autre part de la confidentialité pour assurer que une personne tierce n'est pas capable de savoir la vote d'un électeur. Les contraintes sont encore plus nombreuses quand on doit tenir compte que, pour des raisons de incoercibilité, l'électeur ne doit pas être capable de prouver à une personne tierce sa préférence ou bien qu'uniquement les personnes admissibles ont le droit de voter.

3.2 Cryptographie sur les courbes elliptiques

Le système de Helios, ainsi que d'autres systèmes comme le GNU Privacy Guard[3], utilise le cryptosystème ElGamal. C'est un système asymétrique probabiliste qui repose sur le problème du logarithme discret. Toutes les opérations sont faites dans un groupe multiplicatif $\mathbb{Z}/p\mathbb{Z}$, représenté par un générateur g. Casser l'algorithme ElGamal est dans la plupart des cas au moins aussi difficile que de calculer le logarithme discret.

Il existe aussi une façon de formaliser le problème du logarithme discret à l'aide de courbes elliptiques. Celles-ci sont un cas particulier des courbes algébriques, qui possèdent des propriétés d'addition géométriques assez intéressantes pour le domaine de la cryptographie. Plusieurs formes de courbes elliptiques ont été étudiés les dernières années. Parmi les plus connues sont:

- La forme de Weierstrass: $y^2 = x^3 + ax + b$
- La forme de Montgomery: $y^2 = x^3 + ax^2 + x$
- La forme de Edwards: $x^2 + y^2 = 1 + dx^2y^2$

Chacune des formes a ses propres lois d'additions. Par exemple pour ajouter deux points quelconques P et Q sur une courbe de Weierstrass il suffit de tracer une tangente qui passe par et P et Q, trouver le nouveau point (R) d'intersection avec la courbe, et calculer son inverse. Vous trouverez en annexe l'expression mathématique de la loi d'addition pour les 3 courbes. Sur la figure ci-dessous vous pouvez voir une illustration de ce processus.

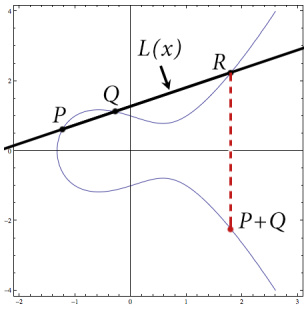

Figure 1: Addition de deux points sur une courbe de la forme Weierstrass

On utilise ce loi pour formaliser le problème du logarithme discret de la manière suivante: nous avons un corps fini $\mathbb{F}q$ avec q un nombre premier, E une courbe elliptique sur $\mathbb{F}q$, P le point générateur et p l'ordre du groupe. La clé secrète est un nombre aléatoire k et la clé publique $Q = kP$. Ainsi pour chiffrer un point M sur la courbe nous tirons aléatoire un nombre r et nous calculons $R = rP$ et $S = M + rQ$. Le chiffré est donc le couple (R,S). Ensuite pour déchiffrer il suffit que le porteur de la clé secrète calcule $S - kR = M + rkP - krP$. Il est donc évident que nous avons besoin de deux algorithmes pour réaliser ce cryptosystème, l'addition de deux points et la multiplication d'un point par un nombre entier. La multiplication d'un point par un entier se fait par l'algorithme "double and add" [13].

Ce cryptosystème a des propriétés assez intéressantes qui sont utiles pour proposer des solutions aux problèmes du vote électronique. La première étant l'homomorphisme additif qui nous permet d'additionner deux chiffrés et récupérer au déchiffrement l'addition de deux messages et la deuxième le re-chiffrement qui est utile pour anonymiser les votes.

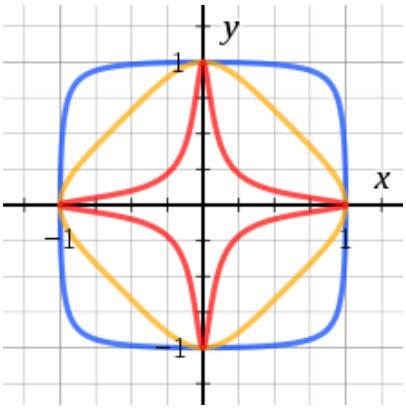

Figure 2: Trois courbes sous la forme d'Edwards

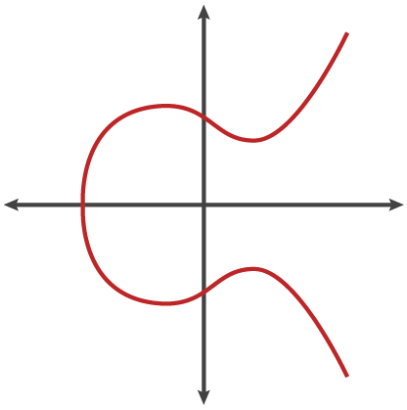

Figure 3: Une courbe de la forme Weierstrass

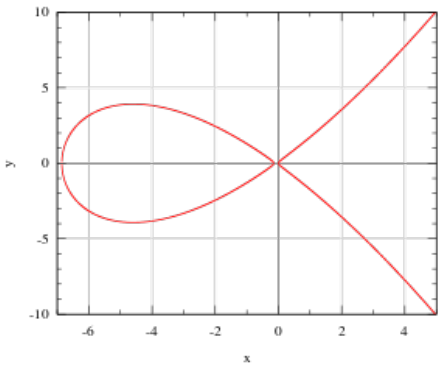

Figure 4: Une courbe de Montgomery

3.2.1 Addition homomorphe

Considérons une élection de type référendum, nous décidons d'attribuer au candidat 1 l'élément neutre de l'addition $\mathcal{O}$ de notre courbe elliptique et au candidat 2 un point X. Imaginons maintenant deux chiffrées, $c_1 = (R_1, S_1) = (r_1P, \mathcal{O} + r_1Q)$ et $c_2 = (R_2, S_2) = (r_2P, X + r_2Q)$. Si l'on ajoute les deux chiffrées on récupère $(r_1P + r_2P, \mathcal{O} + r_1Q + X + r_2Q)$ et puis au déchiffrement $\mathcal{O} + r_1Q + X + r_2Q - r_1Q - r_2Q = \mathcal{O} + X$. Puisque N est l'élément neutre $N + X = X$ alors nous savons que le candidat 2 a reçu 1 vote sur 2. Si l'on applique ce processus pour une liste de votes de longueur N nous récupérons au déchiffrement nX. Il suffit donc d'identifier ce n pour savoir le résultat de cette élection. Ceci nous permet de trouver le candidat avec le plus des votes sans avoir besoin de déchiffrer les votes. Par contre ce système définit une contrainte; puisque il n'existe pas d'algorithme rapide pour trouver n ce nombre il doit être petit, sinon on serait amener à résoudre le problème du logarithme discret.

3.2.2 Re-chiffrement

Le re-chiffrement est utile pour anonymiser les votes afin de rendre impossible à une personne de remonter à la personne qui a voté sachant un chiffré. Soit un chiffré $c = (rP, X + rQ)$ le re-chiffrement se fait de

cette manière: $c_2 = (rP + r_2P, X + rQ + r_2Q) = (P(r + r_2), X + Q(r + r_2))$ r_2 etant un nombre aléatoire tiré uniformément modulo q. Nous verrons au chapitre (3.4.3) comment montrer l'équivalence de deux chiffrées.

3.2.3 Pourquoi utiliser les courbes elliptiques?

La raison d'utiliser des courbes elliptiques pour les procédés cryptographiques est la longueur des clés pour atteindre le même niveau de sécurité que les autres crypto-systèmes comme celui de RSA ou d'AES. Les meilleurs algorithmes à nos jours qui résolvent le problème du logarithme discret dans les courbes elliptiques sont le Baby Step Giant Step [4] et le Rho de Pollard [5] ayant une complexité de $O(\sqrt{n})$ avec n l'ordre du groupe. Par conséquence pour avoir une sécurité de 128 bits (le standard aujourd'hui) il faut une longueur de clé au moins 256 bits, alors que pour la même sécurité avec RSA il faut 4096 bits.

Pour exprimer donc un point sur la courbe il nous faut 256 bits pour la coordonnée x ainsi qu'un bit de plus (0 ou 1) pour savoir lequel de deux points sur la courbe on attend. On note que sachant un x il existe deux solutions, une positive et une négative, quand on ressoude l'équation quadratique de la courbe. Avec un bit on peut savoir lequel des deux on cherche. Cette propriété des courbes elliptiques entraîne des performances bien meilleures que les autres algorithmes.

3.3 Mix network

Un réseau de mélange, mix-network, est un ensemble de nœud de routage qui permettent d'anonymiser de l'information sans y avoir accès ni à son origine ni à sa destination finale. Ils ont été introduit en informatique par David Chaum en 1981 [7] et un exemple d'utilisation le plus connu est le projet TOR [6].

L'utilisation d'un tel protocole est utile pour les élections afin d'anonymiser les votes. L'anonymisation des votes est utile afin de permettre la publication ouverte de la liste des bulletins et donner la possibilité d'auditer les résultats à toute personne qui a accès à la clé secrète.

Dans sa thèse [2], Ben Adida présente à plusieurs protocoles de mix network pour enfin choisir celui de Sako - Killian / Benaloh. La principe de ce protocole est que chaque nœud permute et re-chiffre les votes qu'il reçoit et les envoie au prochain nœud. Il suffit donc d'au moins un nœud honnête pour assurer l'impossibilité de tracer les votes. Comme nous avons vu au (3.2.2) pour re-chiffrer nous avons besoin que de la clé publique donc nous garantissons aussi que les nœuds n'aurons pas accès à l'information chiffrée.

Cependant pour réaliser ce protocole il nous faut deux preuves, une preuve de la bonne permutation (celle-ci ne sera pas implémentée dans notre protocole, faute de temps) et une preuve de re-chiffrement. Vous trouverez plus des détails au chapitre suivant.

3.4 Preuves à connaissance nulle

Dans le domaine de la cryptographie les preuves à connaissance nulle sont des preuves utilisées pour que une entité, le "prouver", prouve à une autre entité, le "verifier", qu'une proposition est correcte sans toutefois révéler une autre information que la véracité de la proposition. Ces preuves sont souvent construites en trois cycles; premièrement le prouver envoie un "commitment", le verifier réponde avec un

"challenge" et finalement le prouver renvoie la "réponse" pour que le verifier puisse la vérifier. Dans ce type des preuves nous distinguons trois propriétés principales:

1. Consistance. Si les deux entités suivent le protocole, le verifier acceptera toujours la preuve.
2. Solidité. Un prouver malhonnête ne peut pas convaincre un vérifier honnête que la proposition est vraie et ceci avec une forte probabilité
3. Aucun apport d'information: le verifier ainsi que toute personne tierce qui suit les echanges pendant la preuve n'apprennent de la part du prouver rien de plus que la véracité de la proposition. Cette propriété reste valable même si le verifier ne suit pas le protocole

Alors ce type de preuves sont utiles dans le vote électronique puisque un électeur doit être capable de vérifier la bonne prise en compte de son vote (que le système a bien chiffré sa préférence), les mix networks doivent être capable de prouver leur honnêteté lors du re-chiffrement ou la permutation ainsi que le système doit montrer publiquement que l'addition homomorphe est bien déroulé.

3.4.1 Preuve de Chaum Pedersen

Une preuve indispensable pour réaliser ces preuves est la preuve de Chaum et Pedersen pour une hypothèse d'un tuple Decisional Diffie–Hellman (DDH). L'hypothèse signifie que $\forall w, t, u \in \mathbb{Z}_q$ il est calculatoirement difficile de distinguer les deux tri plets (tP, wP, wtP) et (tP, wP, uP).

Chaum et Pedersen utilise cette hypothèse dans sa preuve: Soit P le point générateur, $Q = kP$ une clé publique, et $c = (R, S)$ un chiffré. Si le c correspond au chiffrement de $\mathcal{O}$ (élément neutre) nous avons $R = rP$ et $S = rxP$ tandis que si c'est un chiffré de 1 il correspondra à $R = rP$ et $S = P + rkP$. Le prouver qui a l'aléa utilisé pour chiffrer est capable, à l'aide du protocole de Chaum Pedersen, de prouver que le tuple (P, Q, W, U) avec $W = R$ et $U = S$ pour le chiffré de 0 et $W = R$ et $U = S - P$ pour le chiffré de 1 est un tuple DDH valide. Ce protocole est établi sur 3 cycles. Pour le premier le prouver envoie le couple (commitment) ($a = sP$, $b = sQ$) avec s un nombre aléatoire. Le vérifier lui renvoie un aléatoire v et enfin le prouver lui renvoie $t = s + vr$. Le verifier vérifie alors deux égalités:

- $tP == a + qW$. Ceci est vraie car $tP = (s + vr)P$ et $a + qW = sP + vrP = (s + vr)P$
- $tQ == b + qU$. Aussi vraie car $tQ = (s + vr)kP$ et $b + qU = skP + vrkP = (s + vr)kP$

3.4.2 Preuve de disjonction

Cette preuve nous est utile pour arriver à montrer que nous avons bien chiffré soit 0 soit 1. Nous appelons ceci une preuve de disjonction $(c_1 \vee c_2)$ [8] [22].

Supposons que l'électeur vote pour le candidat 0. Il dispose d'un chiffré $c = (R, S)$ et il veut prouver son vote. La preuve de disjonction se déroule en 3 cycles de la manière suivante:

1. Le prouver suit la première étape de la preuve de Chaum Pedersen (3.4.1) pour le vote de 0 et il calcule $a_1 = sP$. Pour le chiffré de 1 le "commitment" est $a_2 = Pt_2 - U_2 v_2$ avec $s_2, v_2 \xrightarrow{r} \mathbb{Z}_q$ et U_2 le chiffré de 1. Alors cette fois le "commitment" est le tuple (a_1, a_2).
2. Le vérifier envoie un "challenge" v tiré aléatoirement sur $\mathbb{Z}_q$.

3. Pour finaliser la preuve le prouver calcule $v_1 = v - v_2$ et réponde avec le tuple (v_1, v_2, t_1, t_2) avec t_1 calculé de la même façon que (3.4.1). Le verifier a besoin de vérifier 3 preuves:

   - $v == v_1 + v_2$
   - $t_1 P == a_1 + v_1 U_1$. Cette preuve est identique à la première preuve de (3.4.1)
   - $t_2 P == a_2 + v_2 U_2$. Ceci est vrai car $a_2 + v_2 U_2 = t_2 P - U_2 v_2 + v_2 U_2 = t_2 P$.

On note ici que la preuve pour le chiffré de 1 est simulé sans que celui existe et calculé explicitement pour que cela fonctionne. Ceci nous permet de prouver $c_1 \vee c_2$ sans que le verifier puisse savoir lequel de deux chiffrés on connaît véritablement. Il est possible de simuler une preuve "Zero Knowledge", justement parce qu'elle est "Zero Knowledge". Le fait que cette preuve ne divulgue pas d'informations signifie qu'il est possible de simuler les échanges entre le prouveur et le vérifier sans connaître le secret du prouveur (en la "simulant").

3.4.3 Preuve de re-chiffrement

Considérons maintenant un chiffré $c_1 = (R_1, S_1)$ et $c_2 = (R_2, S_2)$ son re-chiffrement comme nous avons vu au (3.2.2). Pour prouver d'une manière "zero-knowledge" que ceci est correcte on construit le tuple DDH suivant: (P, Q, W, U) ou P est le point générateur, Q la clé publique, $W = R_2 - R_1$ et $U = S_2 - S_1$. Ensuite nous avons qu'à suivre le protocole décrite au (3.4.1) pour le vérifier.

4 Étude pratique

Cette recherche a abouti à une étude pratique et notamment à la création d'un module des courbes elliptiques qui peut servir aux projets qui utilisent les courbes elliptiques pour la partie cryptographique, et d'une simple application web pour illustrer le processus d'une élection afin de mesurer la performance du système et voir dans la pratique les problèmes que nous pouvons rencontrer lors d'une implémentation d'un protocole cryptographique.

Vous pouvez consultez le code source [9] sous la plate-forme Github.

4.1 Module des courbes elliptiques

Au total ce module compte environ 1000 lignes de code Python reparties de la manière suivante:

- curve.py 357 lignes de code. Implémentation de trois types de courbes elliptiques avec leur propres lois d'addition et de multiplication.
- elGamal.py 199 lignes de code. Fonctions pour générer des clés, chiffrer, déchiffrer et ajouter de manière homomorphe.
- texts.py 25 lignes de code. Les classes de textes en claire et des chiffrées.
- zkp.py 83 lignes de code. Les preuves à connaissance nulle.

- tools.py 310 lignes de code. Plusieurs fonctions notamment utilisés pour le modulo inverse, la racine carrée modulaire, Baby Step Giant Step et Rho de Pollard (fonctions pour résoudre un logarithme discret) et d'autres différents algorithmes utilisées dans le module.

Ce travail pourrait se rendre utile pour les personnes qui utilisent les courbes elliptiques dans leurs projet. Il implémente les plus importantes fonctionnalités qu'un projet ait besoin, parfois même optimisées. Les algorithmes mathématiques utiles pour les courbes ont été vérifiés avec le logiciel Sage [10] un logiciel libre qui utilise python pour créer un outil mathématique visant d'offrir une solution libre comme alternative à Matlab, Maple et Mathematica.

Remarque 4.1. *Les preuves décrites au* (3.4) *sont toutes interactives. L'heuristique de Fiat-Shamir [15] nous permet de les rendre en non interactive. Dans ce cas le verifier au lieu d'envoyer un aléa utilise une fonction cryptographique de hachage. Dans le module, la fonction utilisée est la SHA512 [14] et vous pouvez voir cette transformation dans l'annexe et la fonction* ***chaum_pedersen_ddh****.*

4.2 Création d'une application web

Une fois la plupart des fonctionnalités utiles pour un processus d'élection ont été implémentées l'étape suivante était de mettre en place une plate-forme pour simuler des référendums. Voici une figure qui décrit ce processus:

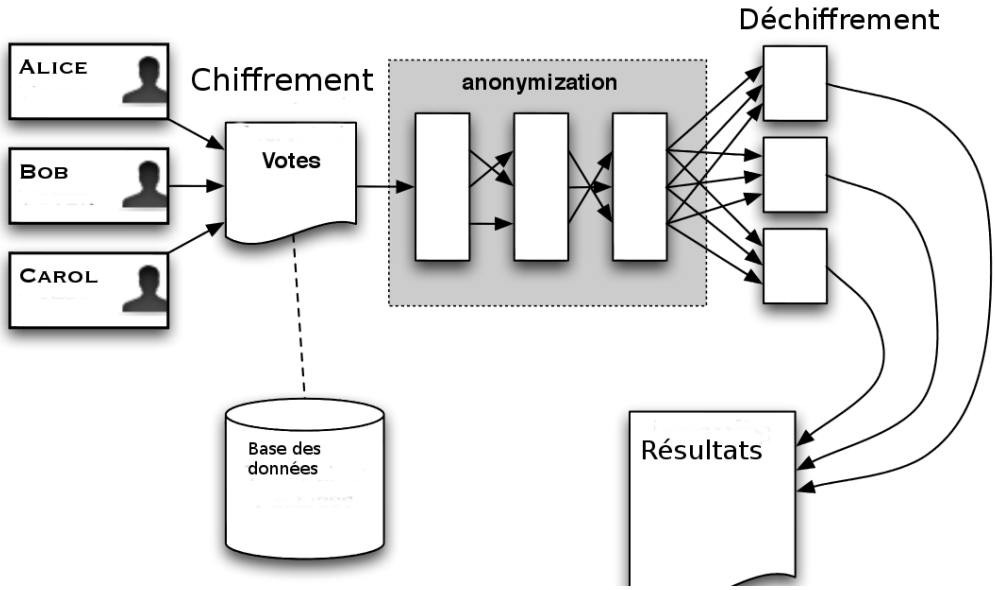

Figure 5: Processus d'une élection

Remarque 4.2. *Sur la figure ci-dessus on observe les grandes lignes d'une élection. Les électeurs font leurs votes et on les enregistrent dans une base des données. Une fois toutes les votes sont enregistrés on procède à la partie de l'anonymisation et finalement au déchiffrement (sur la figure le déchiffrement utilise la principe d'une clé partagée).*

4.2.1 Choix de conception

A l'aide du site SafeCurves [17] qui mesure la sécurité de différentes courbes connues, la courbe Curve2213 [16] a été choisit. Elle offre une sécurité de 128 bits.

Pour réaliser ce processus certains choix d'implémentation ont été fait. Premièrement les bulletins des votes sont enregistrées dans des listes python et sérialiser dans de fichiers à l'aide du module **pickle** de python [18]. Ce choix se base sur le besoin d'avoir des déroulements de référendums simultanées et pour éviter d'alourdir le projet avec une base des données vue que le seul besoin est de inscrire un vote et de récupérer la liste.

En ce qui concerne les mix networks et leur communication avec le serveur, nous utilisons la principe de WebSockets qui offre une communication web bidirectionnelle, client / serveur. Ainsi nous avons un serveur qui fonctionne avec le framework **Tornado** [11] et un client créé avec python.

Pour affronter les problèmes d'écoute clandestine sur le réseau de télécommunication entre un électeur et le serveur d'enregistrement il faudrait que le client puisse chiffrer et prouver son chiffrement dans le côté client. La première solution, la plus évidente, serait d'utiliser du JavaScript pour chiffrer au côté client le vote. Malheureusement le temps nécessaire pour re-implémenter les courbes et le chiffrement sous JavaScript ne nous permet d'avancer avec cette solution ainsi nous avons choisit de offrir à l'électeur un exécutable au moment du vote pour chiffrer son vote et l'envoyer au serveur.

Enfin pour la partie front-end de la partie web le micro-framework **Flask** [12] a été utilisé notamment pour sa similitude et la possibilité de le mettre en place avec Tornado.

4.2.2 Côté serveur

Tornado est un framework web évolutif et non-bloquant sous Python et une librairie de réseau asynchrone. En utilisant une entrée/sortie non bloquante il peut ouvrir plusieurs dizaines milliers des connections ce qui le fait idéal les Web Sockets et les applications qui ouvrent des connexions de longe durée avec le client.

Comme vous pouvez voir en annexe le serveur Web Socket est assez simple, ayant les fonctions nécessaires pour envoyer et recevoir des messages ainsi qu'une méthode appelée par la boucle principale qui permet à envoyer des messages aux clients (dans ce cas les mix-networks).

Tornado a sa propre implémentation des templates et du routage pour le web cependant **Flask** est beaucoup plus simple pour les petites applications comme il utilise les templates **Jinja2** [19] et son protocole de routage de URI RESTful est plus simple. L'idée derrière ce programme est la génération de templates selon la requête URI et la complétion avec des fichiers **markdown**. Vous pouvez trouver plus d'informations sur ces caractéristiques dans l'annexe et plus précisément le fichier **flask_tornado.py**.

4.2.3 Côté client

Les mix-networks sont de simples interfaces qui sont toujours en attente d'une instruction du serveur. Ils utilisent une boucle infinie, font du "parsing" en continue les messages du serveur et répondent selon les besoin. Dans le projet le fichier "client_mixnet" compte 61 lignes de code. Dans cette première version du projet on utilise un seul client mix network pour simplifier. Il suffit de quelques modifications du code

pour installer plus des clients notamment pour spécifier le routage.

Quelques figures sont à votre disposition dans l'annexe qui illustrent ces fonctionnalités ainsi que des fichiers des traces entre le serveur et le client mix network.

5 Questions ouvertes

Ce projet fut une première étude du sujet du vote électronique avec les courbes elliptiques et il est loin d'être fini. On note que la partie théorique laisse des questions ouvertes sur la possibilité de créer des élections avec plus de deux candidats ainsi qu'à étudier les preuves de permutation (utile pour le mix network), la preuve d'addition homomorphe et enfin la possibilité qu'un utilisateur puisse tracer son vote toute au long de l'élection (une fonctionnalité implémenté dans Helios). Une piste pour traiter le problème de plusieurs candidats est étudié par Sergi Siso Godia [20] qui serait cohérente avec le stade actuel de la plate-forme réalisé.

En ce qui concerne la partie pratique, l'application web reste assez basique et en l'état elle ne peut pas être utilisée publiquement. Notamment la création d'un module JavaScript semble indispensable mais d'autres améliorations sont possibles dans les fonctions de création des élections, de vote ou de l'addition homomorphe. Enfin la mise en place d'un système d'authentification pour contrôler qui est éligible de voter est aussi impérative.

6 Conclusion

Durant ce projet j'ai eu l'occasion de me confondre à plusieurs problèmes dans le domaine de la cryptographie à la fois théoriques et d'autres pratiques. La possibilité de traiter un sujet moderne comme le vote électronique m'a beaucoup passionné. L'opportunité de me familiariser avec les courbes elliptiques, les preuves à connaissance nulle et les mix networks, qui sont des sujets modernes et actuellement à la recherche semble inestimable. Le projet est ma première réel expérience avec le monde de la cryptographie et il m'a permis de approfondir mes connaissances dans ce sujet et confirmer mon intérêt de poursuivre dans cette direction.

7 Annexe

7.1 Loi du groupe

7.1.1 Courbe de Weierstrass

La forme de Weierstrass: $y^2 = x^3 + ax + b$

- Addition $P(x_1, y_1) + Q(x_2, y_2)$:
  $x = (y_2 - y_1)^2/(x_2 - x_1)^2 - x_1 - x_2$
  $y = (2x_1 + x_2)(y_2 - y_1)/(x_2 - x_1) - (y_2 - y_1)^3/(x_2 - x_1)^3 - y_1$

- Multiplication $2P$
  $x = (3x_1^2 + a)^2/(2y_1)^2 - x_1 - x_1$
  $y = (2x_1 + x_1)(3x_1^2 + a)/(2y_1) - (3x_1^2 + a)^3/(2y_1)^3 - y_1$

7.1.2 Courbe de Montgomery

La forme de Montgomery: $y^2 = x^3 + ax^2 + x$

- Addition $P(x_1, y_1) + Q(x_2, y_2)$:
  $x = (y_2 - y_1)^2/(x_2 - x_1)^2 - a - x_1 - x_2$
  $y = (2x_1 + x_2 + a)(y_2 - y_1)/(x_2 - x_1) - (y_2 - y_1)^3/(x_2 - x_1)^3 - y_1$

- Multiplication $2P$
  $x = (3x_1^2 + 2ax_1 + 1)^2/(2 * y_1)^2 - a - x_1 - x_1$
  $y = (2x_1 + x_1 + a)(3x_1^2 + 2ax_1 + 1)/(2 * y_1) - (3x_1^2 + 2ax_1 + 1)^3/(2 * y_1)^3 - y_1$

7.1.3 Courbe de Edwards

La forme de Edwards: $x^2 + y^2 = 1 + dx^2y^2$

- Addition $P(x_1, y_1) + Q(x_2, y_2)$:
  $x = (x_1y_2 + y_1x_2)/(1 + dx_1x_2y_1y_2)$
  $y = (y_1y_2 - x_1x_2)/(1 - dx_1x_2y_1y_2)$

- Multiplication $(2P)$
  $x = (x_1y_1 + y_1x_1)/(1 + dx_1x_1y_1y_1)$
  $y = (y_1y_1 - x_1x_1)/(1 - dx_1x_1y_1y_1)$

7.2 Tornado - Flask

```
'''
Websocket server
'''
class WSHandler(websocket.WebSocketHandler):
    clients = []
    def open(self):
        WSHandler.clients.append(self)
        print 'connection opened...'
        #self.write_message("Mix")

    def on_message(self, message):
        print 'received:', message
        if message == "Waiting":
            print "sending"
            self.write_message(json.dumps(Alice.Pk.__dict__))
            retrieved = tools.retrieve_list("test")
            self.write_message(json.dumps([cipher.__dict__ for cipher in retrieved]))
        else:
            votes = json.loads(message, object_hook = cipher_decoder)
            tally = Alice.tallying(votes)
            print "searching"
            res = Alice.find_solution(Alice.decrypt(tally))
            print "\nThe candidate 1 has, ",res," votes out of ",len(votes),"\n"
            tools.write_result(res,votes,tools.retrieve_list("test"))
            #tornado.ioloop.IOLoop.instance().add_callback(result,res)

    @classmethod
    def ask_to_mix(cls):
        print "Writing to client"
        for client in cls.clients:
            client.write_message("Mix")

'''
Routing URLS
'''
@app.route('/vote', methods=('GET', 'POST'))
def vote():
    vote1 = Alice.basePoint
    vote2 = Coord(-1,-1)
    form = SimpleForm()
    form.certification.choices = [(str(vote1),'cand1'),(str(vote2),'cand2')]
    if form.validate_on_submit():
        vote = tools.parse_vote(request.form.get('certification'))
        print vote == vote1
        print "retrieving list and encrypting vote"
        retrieved = tools.retrieve_list("test")
        retrieved.append(Alice.encrypt(vote)[0])
        #print "lets see",Alice.decrypt(retrieved[-1]) is Alice.basePoint
        print "saving into file"
        tools.save_list(retrieved,"test")
        return "<h1>Success</h1>"
    #print render_template('vote.html', form=form),form
    return render_template('vote.html', form=form)

```

```
@app.route('/<path:path>/')
def page(path):
    #print pages.get(path)
    page = pages.get_or_404(path)
    #print "res",page
    return render_template('page.html', page=page)
```

7.3 Chaum Pedersen

```
def chaum_pedersen_ddh(curve, Pk, w, u, r):
    '''
    Proover sends (a,b)=(s*g,s*y) s random
    Verifier sends random c
    Proover sends t=(s+cr)
    Verifier check: g*t == a+w*c and y*t == b+u*c
    '''
    y = Pk.point
    g = Pk.basePoint
    s = random.randint(1, curve.order)
    a = curve.double_and_add(g, s)
    b = curve.double_and_add(y, s)
    hash_object = hashlib.sha512(str(a.x))
    hex_dig = hash_object.hexdigest()
    c = long(hex_dig, 16) % curve.order
    t = (s + c * r) % curve.order
    left_hand = curve.double_and_add(g, t)
    right_hand = curve.add(a, curve.double_and_add(w, c))
    left_hand2 = curve.double_and_add(y, t)
    right_hand2 = curve.add(b, curve.double_and_add(u, c))
    if((left_hand == right_hand) and (left_hand2 == right_hand2)):
        print "DDH verified"
        return True
    else:
        print "Proof not verified"
        return False
```

7.4 Chiffrement - Déchiffrement

```
def encrypt(self, P):
        ''' Encrypt a point P '''
        assert self.curve.is_valid(
            P) is True or P is not Coord(-1,-1), "Trying to encrypt point not on curve"
        r = random.randint(1, self.curve.order)
        R = self.curve.double_and_add(self.basePoint, r)
        __ = self.curve.double_and_add(self.Pk.point, r)
        cipher = self.curve.add(P, __)
        return texts.CipherText([R, cipher]), r

    def decrypt(self, Cipher):
        ''' Decrypt a cipher text '''
        R, c = Cipher.getCipher()
        S = self.curve.double_and_add(R, self.Sk.k)
        invPoint = self.curve.inverse(S)
        plaintext = self.curve.add(invPoint, c)
        return plaintext
```

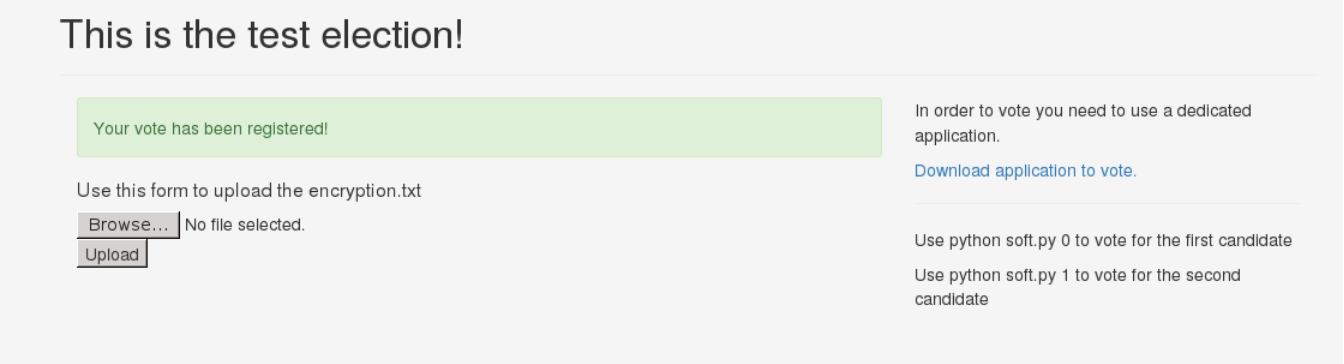

Figure 6: Voter dans une élection

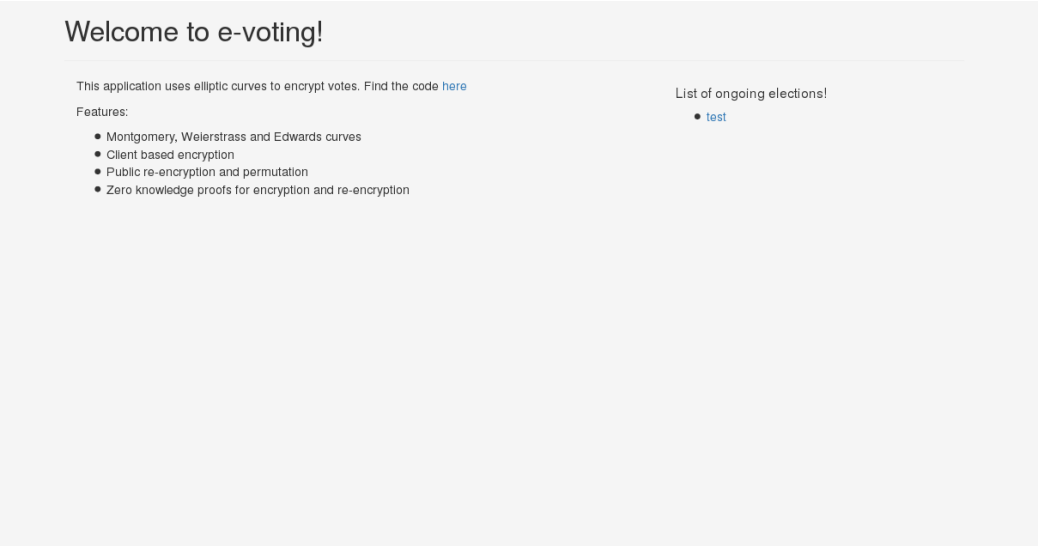

Figure 7: Index de l'application

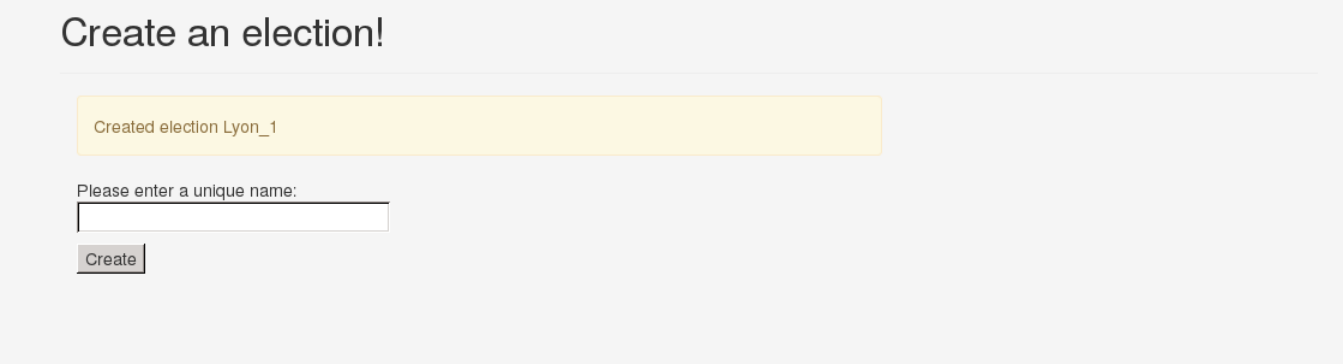

Figure 8: Créer d'une élection

References

[1] Helios voting http://vote.heliosvoting.org/.

[2] Thèse de Ben Adida http://assets.adida.net/research/phd-thesis.pdf

[3] Gnu Privacy Guard https://www.gnupg.org/

[4] Baby Step Giant Step https://en.wikipedia.org/wiki/Baby-step_giant-step

[5] Pollard Rho pour les logarithmes https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm_for_logarithms

[6] The Onion Routing https://www.torproject.org/

[7] Chaum, David L., Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms Commun. ACM Journal, 1981 http://doi.acm.org/10.1145/358549.358563

[8] AndrewClausen., Logical Composition of Zero-Knowledge Proofs http://www.cis.upenn.edu/~mkearns/teaching/Crypto/zkp-disj.pdf

[9] Code source https://github.com/oorestisime/elliptic_curves

[10] Sage http://sagemath.org/

[11] Tornado Web Server http://tornadoweb.org/

[12] Flask http://flask.pocoo.org/

[13] Algorithme "Double and add" https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication#Double-and-add

[14] Famille de fonctions de hachage SHA2 https://en.wikipedia.org/wiki/SHA-2

[15] Amos Fiat and Adi Shamir., How to Prove Yourself: Practical Solutions to Identification and Signature Problems. CRYPTO 1986: http://www.cs.rit.edu/~jjk8346/FiatShamir.pdf

[16] Diego F. Aranha et al., A note on high-security general-purpose elliptic curveshttp://eprint.iacr.org/2013/647.pdf

[17] Safe Curves http://safecurves.cr.yp.to/

[18] Pickle https://docs.python.org/2/library/pickle.html

[19] Jinja Templating System http://jinja.pocoo.org/docs/dev/

[20] Sergi Siso Godia., An electronic voting platform with elliptic curve cryptography http://repositori.udl.cat/bitstream/handle/10459.1/45765/Siso.pdf

[21] Explicit Formulas Datavase https://hyperelliptic.org/EFD/index.html

[22] Zero Knowledge proofs lecture http://cgi.di.uoa.gr/~aggelos/crypto/assets/7_zk_handout.pdf

[23] Lectures of Dr.Rivest on advanced cryptography http://courses.csail.mit.edu/6.897/spring04/materials.html

[24] Helios online documentation http://documentation.heliosvoting.org/.

[25] B.Adida, C. Andrew Neff, Ballot Casting Assurance https://www.usenix.org/legacy/events/evt06/tech/full_papers/adida/adida.pdf

[26] B.Adida, Helios: Web-based Open-Audit Voting https://www.usenix.org/legacy/events/sec08/tech/full_papers/adida/adida.pdf