

MEAE – Transparence et vérifiabilité V2

Spécifications v2.04

www.voxaly.com

SOMMAIRE

1	Introduction.....	6
2	Propriété intellectuelle	7
3	Belenios et personnalisation Voxaly.....	8
4	La cinématique de la phase de vote	9
4.1	Etapes du vote.....	9
4.2	Correspondance avec la description Belenios.....	11
4.3	Flux entre le navigateur et le SI VOTE.....	12
5	Structure des données	13
5.1	Éléments cryptographiques	13
5.2	Bureau de vote, Election, consulat et candidats.....	13
5.2.1	Loria Elections	13
5.2.2	Mise en œuvre Voxcore	14
5.3	Suffrage en clair SU_clair.....	15
5.4	Suffrage chiffré.....	15
5.4.1	Loria Réponse chiffrée	15
5.4.2	Loria Bulletin de vote.....	16
5.4.3	Mise en œuvre Voxcore	16
5.5	Accumulés.....	16
5.5.1	Loria Accumulation	16
5.5.2	Mise en œuvre Voxcore	16
5.6	Déchiffrés partiels.....	17
5.6.1	Loria déchiffrement partiel.....	17
5.6.2	Mise en œuvre Voxcore -modélisation.....	17
5.7	Résultats finaux.....	18
5.7.1	Loria Résultat de l'élection	18
5.7.2	Mise en œuvre voxcore	18
5.8	Cachet de signature du suffrage par le serveur.....	18
5.9	Conditions particulières Législatives 2022	18
5.10	Conditions particulières Législatives partielles 2023	19
6	Preuves utilisées et mise à disposition de tiers	20
6.1	Accusé de réception.....	20
6.2	Récépissé de vote	21
6.3	Preuve du cachet serveur de la preuve de dépôt du bulletin	22

6.4	Preuve de déchiffrement.....	22
7	Détails et pseudo_code des fonctions Voxaly.....	23
7.1	Fonction Vox_Schnorr_CreationKey().....	23
7.2	Fonction Vox_Schnorr_Création_Signature().....	23
7.3	Fonction Vox_Schnorr_Contrôle_Signature().....	24
7.4	Fonction Vox_EncodageCléPrivéeAssesseur().....	24
7.5	Fonction Vox_DécodageCléPrivéeAssesseur().....	25
7.6	Fonction Vox_ReférenceBulletin().....	25
7.7	Fonction Vox_Créationcachetserveur_Preuve_de_Vote().....	26
7.8	Fonction Vox_Controlcachetserveur().....	27
7.9	Fonction Vox_CleRib97().....	27
8	Détail et Pseudo_code des fonctions Loria	29
8.1	Fonction_KG_shamir (m,t).....	29
8.2	Fonction create_answer(y, Q, S, m).....	30
8.3	Fonction eg_encrypt(y, r, m).....	31
8.4	Fonction verify_answer(y, S, Q, a).....	32
8.5	Fonction create_ballot(E, N, SU_clair).....	33
8.6	Fonction verify_ballot(E, b).....	34
8.7	Fonction compute_encrypted_tally(E, bV).....	35
8.8	Fonction compute_partial_decryption(Π , s, S).....	36
8.9	Fonction verify_partial_decryption(Π , S, d).....	37
8.10	Fonction compute_result(E, N, Π , I, Δ).....	38
8.11	Fonction verify_result(E, S, B, r).....	39
8.12	Fonction improve(y, S, α , β , r, m, M).....	40
8.13	Fonction verify_iproof(y, S, α , β , M, π).....	41
8.14	Fonction oprove(y, S, e, r, m, a, b).....	42
8.15	Fonction verify_oproof(y, S, e, a, b, π).....	43
8.16	Fonction obprove(y, S, e, r, m, a, b).....	44
8.17	Fonction obprove0(y, S, P, α_0 , β_0 , α_Σ , β_Σ , r_Σ , m_Σ , M).....	45
8.18	Fonction obprove1(y, S, P, α_Σ , β_Σ , r_0 , M).....	46
8.19	Fonction verify_obproof(y, S, e, a, b, π).....	47
8.20	Fonction bprove(y, S, e, r, m).....	48
8.21	Fonction verify_bproof(y, S, e, π).....	49

9	Détail implémentation.....	50
9.1	Zoom sur sur verify_answer() et calcul de h_iproof.....	50
9.1.1	Exemple code JAVA	50
9.1.2	Exemple numérique pour h_iproof.....	50
9.2	Zoom sur verify_answer() et calcul de H_bproof0 et H_bproof1	52
9.2.1	Exemple de code JAVA	53
9.2.2	Exemple numérique pour HBproof0	55
9.2.3	Exemple numérique pour HBproof1	57
9.3	Zoom sur verify_partial_decryption() et le calcul de hdecrypt.....	59
9.3.1	Exemple de code JAVA	59
9.3.2	Exemple numérique	60
9.4	Zoom sur ECPointUtil.ECPTtoHex(G)	61
9.4.1	Méthode ECPTtoHex.....	61
9.4.2	Exemple avec la vérification du Cachet Serveur	62
9.5	Zoom sur Vox_Créationcachetserveur_Preuve_de_Vote(), génération du cachet serveur pour les suffrages 62	
9.5.1	Exemple	63
9.6	Zoom sur Vox_Controlecachetserveur(), vérification du cachet serveur pour les suffrages	65
9.6.1	Exemple	65
9.7	Zoom sur Vox_CleRib97().....	67
9.7.1	Code Java	67
9.8	Zoom sur UUID	67
9.8.1	Exemple de code Java:.....	68
10	Glossaire.....	69

SUIVI DES VERSIONS

Dates	Version	Intervenant	Commentaire
05/12/2021	0.1	B Chenon	<ul style="list-style-type: none"> Création
17/12/2021	0.21	B Chenon	<ul style="list-style-type: none"> Ajout exemple réceptionné de vote
03/01/2022	0.3	B Chenon	<ul style="list-style-type: none"> Ajout précision sur le calcul des déchiffrés partiels, variable hdecrypt (§8.3)
04/01/2022	0.4	B Chenon P Bouquillon	<ul style="list-style-type: none"> Ajout précision sur le calcul des preuves du bulletin, variables h_iproof, h_bproof0 et h_proof1 (§8.1 et §8.2)
10/01/2022	0.5	B Chenon P Bouquillon	<ul style="list-style-type: none"> Précision sur InfoSU dans Vox_CreationCachetServeur (§6.7) Ajout précision sur la génération du cachet serveur (§8.4, §8.5 et §8.6)
11/01/2022	0.6	P Bouquillon	<ul style="list-style-type: none"> Précision sur Vox_CleRib97() (§8.7)
11/01/2022	0.7	P Bouquillon	<ul style="list-style-type: none"> Mise à jour génération et contrôle cachet serveur (§8.5 et §8.6)
11/01/2022	0.8	B Chenon P Bouquillon	<ul style="list-style-type: none"> Précision sur Vox_CleRib97() (§6.9 et §8.7) – gestion caractère inconnu
10/03/2022	0.9	B Chenon	<ul style="list-style-type: none"> Ajout de l'ordre de l'élection au sein de la preuve (§4.4.3 et §8.8)
19/04/2022	1.0	B Chenon	<ul style="list-style-type: none"> Ajout du tour dans InfoSU (§6.7) Mentions sur la propriété intellectuelle
18/08/2022	1.1	B Chenon	<ul style="list-style-type: none"> Correctif réceptionné de vote (§5.1) Correctif sur cachet serveur (§6.7)
24/10/2022	2.01	B Chenon	<ul style="list-style-type: none"> Ajout identifiant de la circonscription consulaire dans la preuve (session_id) Différentes corrections orthographiques Précision sur electionUUID Ajout de la cinématique de vote, flux entre navigateur et serveur
14/12/2022	2.02	B Chenon	<ul style="list-style-type: none"> Précisions sur la clé témoin (§4.1) Précision sur la génération du document « réceptionné de vote » (§4.1 et §4.2)
18/01/2023	2.03	B Chenon	<ul style="list-style-type: none"> Précision sur étape 8 (§4.1) pour la construction SU_Clair
22/02/2023	2.04	B Chenon	<ul style="list-style-type: none"> Correction texte sur réceptionné de vote (§6.2) Correction sur le calcul de InfoSU dans le cachet serveur (§7.7) Précision sur le contrôle H versus H'' : message d'erreur à l'électeur Version diffusée pour les législatives partielles 2023

1 INTRODUCTION

Ce document présente la spécification du protocole de vote Voxaly sur la transparence de l'urne, afin d'effectuer des opérations de vérifiabilité individuelle et de vérifiabilité universelle par un tiers.

Cette spécification concerne la partie chiffrement et la partie déchiffrement.

La modélisation des données permet de gérer différents natures de scrutins, avec des organisations complexes, et parfois plusieurs bureaux de vote. Les conditions particulières pour le scrutin des législatives 2022 et les partielles de 2023 du MEAE sont précisés dans le chapitre « §5.9 - Conditions particulières Législatives 2022. » et « §5.10 - Conditions particulières Législatives partielles 2023 ».

2 PROPRIETE INTELLECTUELLE

Ce document, ainsi que l'ensemble des informations qu'il contient, sont la propriété exclusive de VOXALY DOCAPOSTE. Le présent document n'a pas vocation à opérer le moindre transfert de propriété.

La permission est accordée par la présente, gratuitement, à toute personne obtenant une copie de cette documentation, de l'utiliser afin d'effectuer des opérations de vérifiabilité individuelle et de vérifiabilité universelle dans le cadre du scrutin des législatives 2022 et des partielles 2023 du MEAE. Ce droit d'utilisation implique les droits de copie, de modification, de fusion, de publication, sous réserve du respect des conditions suivantes :

- L'avis de propriété intellectuelle ci-dessus et cet avis d'autorisation doivent être inclus dans toutes documentations reprenant une partie substantielle des présentes, ou établie à partir d'informations présentes dans ce document.
- Toutes les republications doivent conserver une copie intacte de cet avis de propriété intellectuelle.

En tout état de cause, toute personne ayant accès à ce document, ou à toute documentation reprenant les informations contenues dans le présent document, s'interdit toute exploitation commerciale du contenu du présent document sans l'accord écrit préalable de VOXALY DOCAPOSTE.

3 BELENIOS ET PERSONNALISATION VOXALY

La solution retenue résulte d'un rapprochement entre la solution historique Voxcore de Voxaly, un sous-ensemble du protocole Belenios et de certains de ces algorithmes, développés au Loria. Le protocole Belenios est lui-même une évolution du protocole Helios, développé par Ben Adida, Olivier de Marneffe et Olivier Pereira.

Le protocole Belenios a fait l'objet d'une publication dans une conférence académique présentant le protocole ainsi que la preuve (dans un modèle cryptographique) du secret du vote et de la vérifiabilité de l'élection :

Election Verifiability for Helios under Weaker Trust Assumptions.

Véronique Cortier, David Galindo, Stéphane Glondu et Malika Izabachene.

Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14), pages 327–344, LNCS 8713, Springer, 2014.

La solution présentée ici intègre une part importante de Belenios (notamment les preuves zéro-knowledge de bonne formation des bulletins chiffrés et les preuves de bon déchiffrement). Les principales différences sont les suivantes :

- Dans Belenios, les bulletins chiffrés sont également signés par l'électeur.
- L'urne (l'ensemble des bulletins chiffrés) n'est pas publique dans la solution Voxaly. Il s'agit d'une donnée d'audit : avec les preuves de bon déchiffrement présentes dans le résultat, un auditeur peut s'assurer de la conformité du résultat vis-à-vis de l'urne.
- La génération des clés privées des assesseurs a été adaptée à la solution Voxaly : un serveur dédié génère l'ensemble des clés privées mais les données stockées sur le serveur sont uniquement des données publiques, qui ne permettent pas de déchiffrer l'élection.

4 LA CINEMATIQUE DE LA PHASE DE VOTE

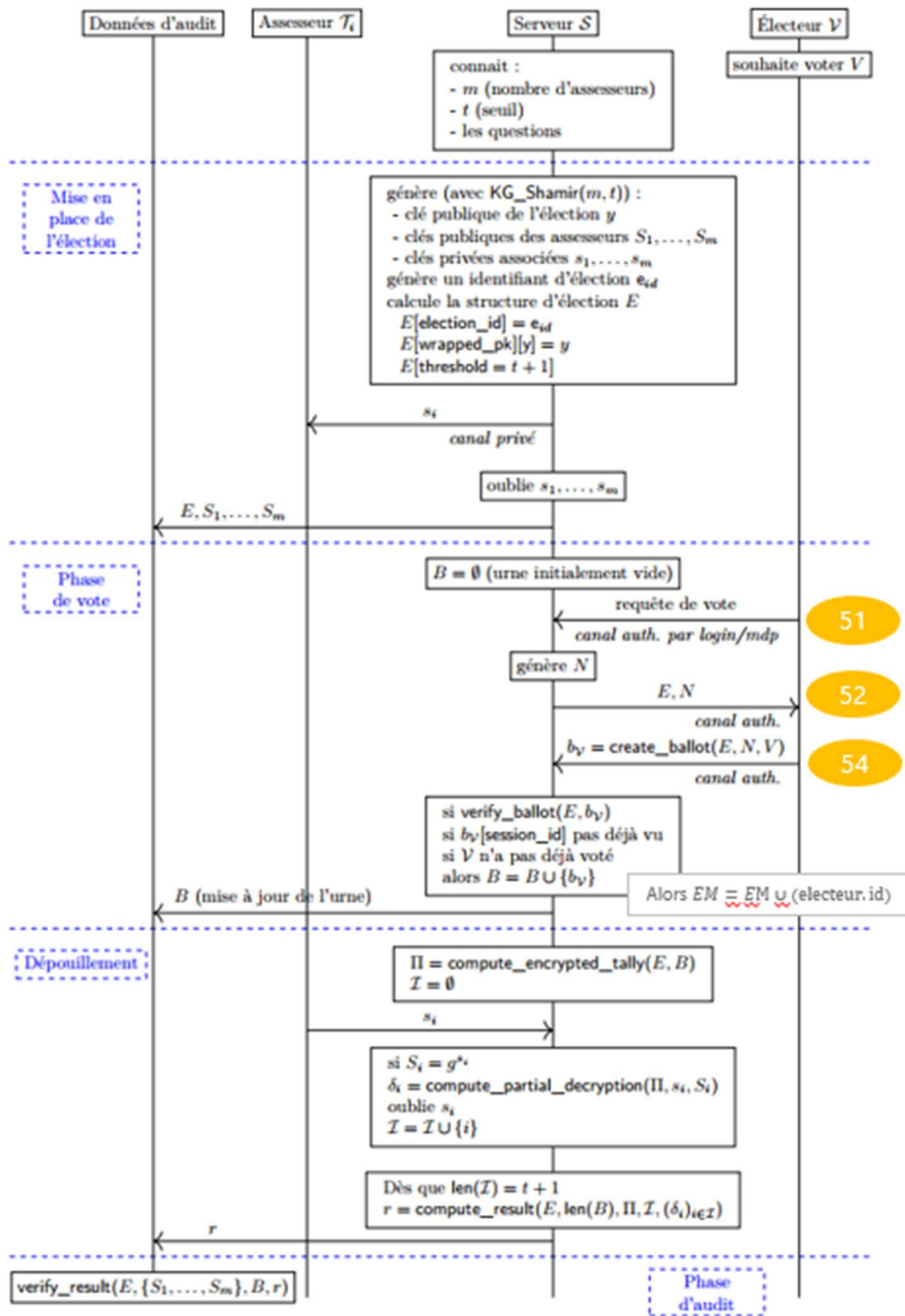
4.1 ETAPES DU VOTE

Nous décrivons ici l'échange entre un électeur V et le serveur SI VOTE :

1. L'électeur tape l'URL du site de vote dans son navigateur (flux #50.1)
2. Le serveur utilise une connexion TLS de niveau suffisant et envoie la page d'accueil et un bulletin témoin #1 à chiffrer,
 - a. Nb : Le bulletin témoin #1 est une chaîne fixe
3. L'électeur V se connecte au SI VOTE avec ses codes de connexion et le navigateur chiffre en parallèle le bulletin témoin #1.
 - a. La clé publique utilisée est une clé dédiée à ce test, ChiffrementBulletinTémoin.keyPub
4. A réception du flux #51, le SI VOTE contrôle les codes de connexion, contrôle le bon déchiffrement du bulletin témoin #1 et autorise l'accès à l'espace de vote.
 - a. Nb : La clé privée utilisée pour ce test est une clé dédiée, ChiffrementBulletinTémoin.keyPriv
5. Si l'électeur est inscrit à plusieurs élections, un échange intermédiaire permet à l'électeur de choisir l'élection (échange non représenté)
6. Le serveur SI VOTE envoie (flux #52) à l'électeur V la structure E de l'élection retenue (contenant notamment la clé publique ChiffrementBulletin.keyPub et les candidats) et ainsi qu'un bulletin témoin #2
 - a. Nb : Le bulletin témoin #2 est un choix, tiré au hasard, parmi les candidats de l'élection retenue
7. L'électeur V choisit sa réponse parmi les choix proposés : choix du candidat ou vote blanc.
8. Le navigateur calcule alors le suffrage en clair (SU_Clair, voir chapitre §5.3) puis le bulletin chiffré $B_v = \text{create_ballot}(E, N, \text{SU_Clair})$ correspondant à V et l'envoie au serveur SI VOTE, N étant composé de l'identifiant de la circonscription électorale et de l'identifiant de la circonscription consulaire
 - a. Le navigateur calcule également le chiffré du bulletin témoin #2, avec la clé publique ChiffrementBulletinTémoin.keyPub.
 - b. Le navigateur calcule la référence $H = \text{Vox_RéférenceBulletin}(E, B_v)$, du bulletin et la stocke localement
9. L'électeur confirme son vote en renseignant le code d'activation.
 - a. Afin de pouvoir afficher un message d'erreur sur la même page, un flux (#54.1) est envoyé vers le serveur pour le contrôle du code d'activation et un premier contrôle de H. Ce flux contient le bulletin chiffré et une preuve de connaissance du code d'activation : $\text{sha256}(\text{code activation et } B_v)$
10. Le serveur envoie la confirmation du code d'activation et également la référence H'.
 - a. Le navigateur contrôle l'égalité entre H et H'
 - b. Le navigateur envoie au serveur SI VOTE la confirmation définitive avec la référence de l'électeur V, `electeur.id`, le bulletin (à nouveau) et le bulletin témoin #2 chiffré (flux #54.3)
11. Le serveur SI VOTE vérifie
 - a. le bon déchiffrement du témoin #2, avec la clé privée ChiffrementBulletinTémoin.keyPriv,
 - b. la validité du bulletin, vis-à-vis des preuves associées,
 - c. l'unicité du bulletin, B_v , vis-à-vis des autres bulletins déjà présents dans la même urne.
12. Puis si tout est ok, le serveur SI VOTE exécute de façon atomique (non dissociable) les traitements suivants (fonction d'enregistrement du vote) :
 - a. Vérification que V n'a pas déjà voté (`electeur.id` non présent dans la liste d'émargement EM)
 - b. Ajout de B_v à l'urne B,
 - c. Ajout de `electeur.id` à la liste d'émargement EM
 - d. Nb : Lors de l'ajout de B_v , le SI VOTE attache à B_v une pastille IDLEC pour permettre un comptage à un niveau plus fin que l'élection. La pastille représente l'identifiant de la circonscription consulaire.

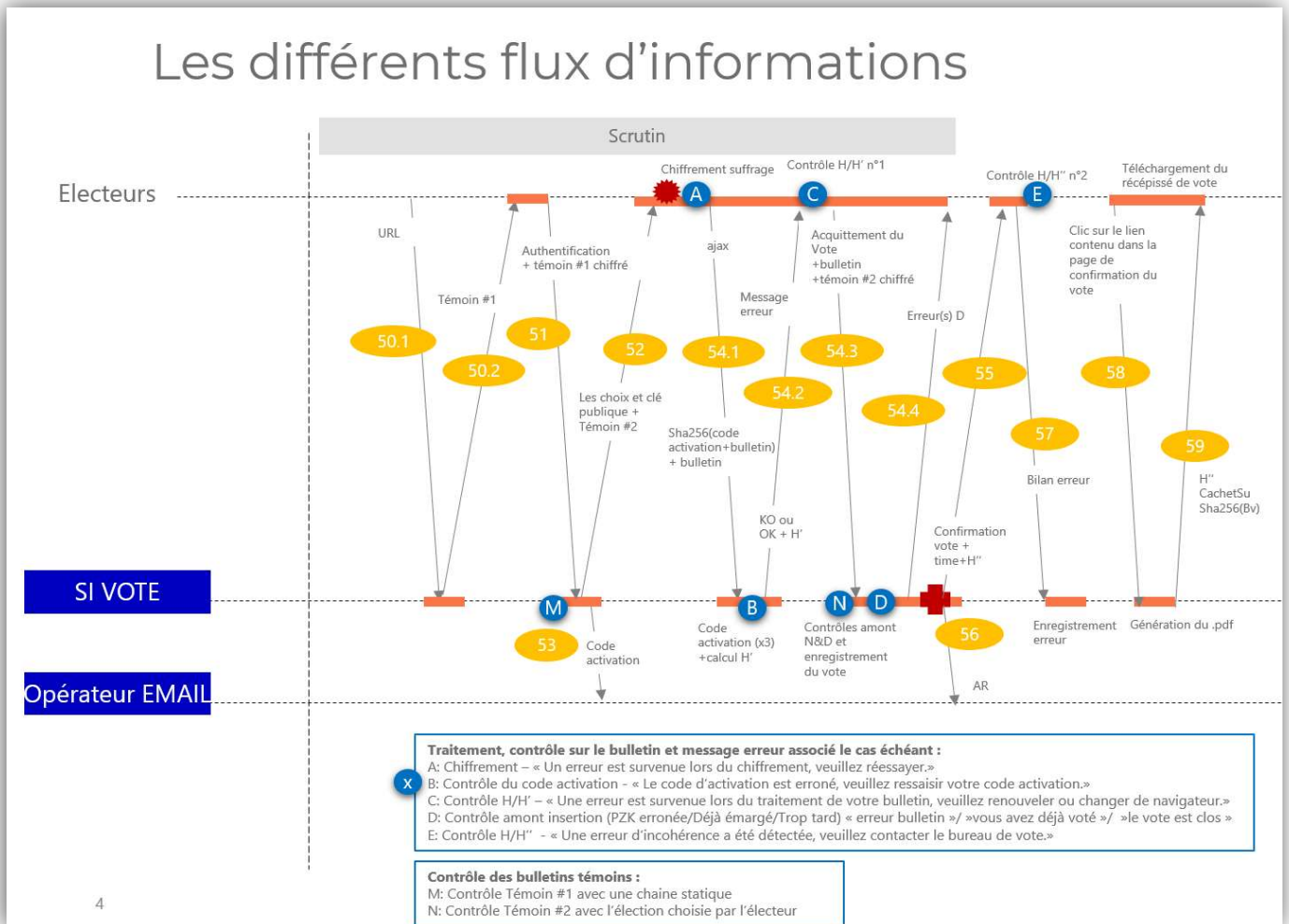
13. Le serveur SI VOTE recalcule la référence H'' et le cachet serveur CachetSU, et stocke ces informations en session
14. Enfin, le serveur SI VOTE renvoie à l'électeur sur son navigateur (flux #55)
 - a. la date de son émargement, Time
 - b. la page accusé de réception avec la référence H''
 - c. le lien pour télécharger le pdf, nommé « Récépissé de vote » (voir étape 17)
15. A réception de la réponse du serveur, le navigateur
 - a. affiche la référence H, stockée localement (cf. étape #8.b),
 - b. affiche la référence H'', fournie par le serveur,
 - c. affiche le lien pour récupérer le récépissé de vote
 - d. et contrôle l'égalité des références H et H'' : Si H est bien récupérée et si une différence est constatée, le navigateur informe l'électeur et envoie au serveur une requête pour l'avertir (flux #57).
 - e. Nb : L'électeur peut également faire le contrôle lui-même entre H et H''.
16. Le SI VOTE envoie également par email à l'électeur, via l'opérateur email :
 - a. la date de son émargement, Time
17. Si l'électeur clique sur le lien « Récépissé de vote », le serveur génère un pdf contenant (flux #59) :
 - a. la référence du bulletin, H'', construite à partir de H'' stockée en session,
 - b. l'empreinte sha256 du bulletin chiffré Bv,
 - c. et le cachet serveur CachetSU.
18. A l'expiration de la durée de vie de la session, le SI VOTE détruit celle-ci automatiquement, entraînant la suppression des informations stockées temporairement : Bv, H'', et CachetSU.

4.2 CORRESPONDANCE AVEC LA DESCRIPTION BELENIOS



4.3 FLUX ENTRE LE NAVIGATEUR ET LE SI VOTE

Lors du vote, voici les différentes étapes et les données échangées, selon un ordre chronologique :



5 STRUCTURE DES DONNEES

5.1 ELEMENTS CRYPTOGRAPHIQUES

L'algorithme de chiffrement utilisé est ElGamal, et donc un chiffré est une paire d'éléments (α, β) d'éléments de \mathbb{G} . Les preuves à divulgation nulle de connaissance (Zero-Knowledge) sont bâties à partir de preuves élémentaires qui prennent la forme de paires (challenge, response) d'éléments de \mathbb{Z}_q .

$$\text{proof} = \left\{ \begin{array}{l} \text{challenge} : \mathbb{Z}_q \\ \text{response} : \mathbb{Z}_q \end{array} \right\} \quad \text{ciphertext} = \left\{ \begin{array}{l} \text{alpha} : \mathbb{G} \\ \text{beta} : \mathbb{G} \end{array} \right\}$$

5.2 BUREAU DE VOTE, ELECTION, CONSULAT ET CANDIDATS

5.2.1 Loria Elections

On suppose ici que le groupe est un sous-groupe d'ordre premier q du groupe multiplicatif d'un corps fini \mathbb{F}_p . On fixe g un générateur.

$$\text{wrapped_pk} = \left\{ \begin{array}{l} g : \mathbb{G} \\ p : \mathbb{N} \\ q : \mathbb{N} \\ y : \mathbb{G} \end{array} \right\}$$

La clé publique de l'élection, notée y dans ce document, est calculée pendant la mise en place de l'élection, et empaquetée avec les paramètres du groupe au sein d'une structure `wrapped_pk`.

$$\text{question} = \left\{ \begin{array}{l} \text{answers} : \text{string}^* \\ \text{blank} : \text{bool} \\ \text{min} : \text{int} \\ \text{max} : \text{int} \\ \text{question} : \text{string} \end{array} \right\} \quad \text{election} = \left\{ \begin{array}{l} ?\text{description} : \text{string} \\ ?\text{name} : \text{string} \\ \text{public_key} : \text{wrapped_pk} \\ \text{questions} : \text{question}^* \\ \text{threshold} : \text{int} \\ \text{election_id} : \text{string} \end{array} \right\}$$

Une élection se compose d'une clé publique empaquetée avec un groupe, de questions et de métadonnées supplémentaires permettant d'identifier l'élection. Le champ `threshold` indique le nombre d'assesseurs nécessaires au dépouillement ($t+1$). Il est important que le champ `election_id` contienne une valeur différente pour chaque élection.

Une question se compose d'un intitulé `question`, d'une liste de réponses possibles `answers`, et de bornes `min` et `max` sur le nombre total de réponses qui peuvent être choisies. Le champ `blank` d'une question indique si l'électeur peut voter blanc (indépendamment de `min` et de `max`) ou non à cette question.

5.2.2 Mise en œuvre Voxcore

Voici la modélisation générique du produit Voxcore, valable pour tous types de scrutin (les conditions particulières pour les législatives 2022 et partielles 2023 sont détaillées au chapitre §5.9 ci-dessous) :

- 1 bureau de vote gère différentes d'élections,
- 1 élection correspond à 1 circonscription électorale.
- Chaque élection est décomposée en plusieurs sous-urnes, correspondant aux différentes circonscriptions consulaires.
- Au sein d'1 élection, l'électeur peut choisir 1 candidat parmi n, ou voter blanc
- (autres modalités de vote non encore décrite : référendum, scrutin de liste avec ou sans rature, ...)
- Chaque élection appartient à un et un seul bureau de vote. Toutes les élections d'un même bureau de vote partagent la même clé publique.

Un bureau de vote est défini par :

- BV.ordre : identifiant unique
- BV.assesseurs : la liste des assesseurs
- BV.min : le nombre minimal d'assesseurs pour dépouiller (nommé également t)
- BV.ChiffrementBulletin.keyPub : la clé publique pour le chiffrement des suffrages (chaque BV dispose de sa propre clé)

Un assesseur (membre du bureau de vote), de T-1 à T_n, est défini par :

- ASS.ordre : identifiant unique
- ASS.CléPublique
- ASS.Cléprivéprotégée
- ASS.Phrasesecretempreinte : PS.E, empreinte de la phrase pour contrôler la saisie au dépouillement
- BV.fk_ordre : son bureau de vote de rattachement

Une élection est définie par :

- Election.ordre : identifiant unique (appelé également « election_id » ou « e_id » dans le document Loria)
- Election.nom : nom affiché sur les écrans
- Election.fk_etordre : la référence de la circonscription associée
- Election.fk_bv.ordre : la référence du BV d'appartenance (et permet de connaître la clé publique)

Une listeCandidats est définie :

- LC.ordre : ordre d'affichage
- LC.nom : nom affichée

Un candidat est défini par :

- Candidat.ordre : ordre d'apparition
- Candidat.info : informations affichées, pouvant également comprendre la description du suppléant
- Candidat.fk_election : élection de rattachement
- Candidat.fk_listeCandidat : liste de candidate de référence dans laquelle le candidat est présent
- Dans un scrutin de nom : une liste fictive regroupe les différents candidats

Une circonscription électorale ou consulaire est définie par

- Etablissement.ordre : son identifiant
- Etablissement.nom : nom de la circonscription
- Etablissement.fk_parent :

- ▶ Pour une circonscription consulaire= id de la circonscription électorale
- ▶ Pour une circonscription électorale = vide

5.3 SUFFRAGE EN CLAIR SU_CLAIR

1 bulletin de vote est composé d'un tableau de compteurs :

- Autant de compteurs que de candidats,
- Et 1 compteur final pour le vote blanc

Chaque compteur est également appelé « micro-bulletin ». Pour chaque compteur, la valeur est définie à 1 si le candidat (ou vote blanc) correspondant a été choisi et défini à 0 sinon.

NB : si d'autres modalités de choix sont proposées, la structure du tableau SU_Clair et donc le nombre de réponses potentielles est adapté en conséquence.

5.4 SUFFRAGE CHIFFRE

5.4.1 Loria Réponse chiffrée

$$\text{answer} = \left\{ \begin{array}{l} \text{choices} : \text{ciphertext}^* \\ \text{individual_proofs} : \text{proof}^{**} \\ \text{overall_proof} : \text{proof}^* \\ \text{?blank_proof} : \text{proof}^* \end{array} \right\}$$

Une réponse chiffrée à une question (dont le nombre total de choix positifs doit être compris entre min et max) est le vecteur **choices** des choix chiffrés donnés à chaque réponse possible. Quand le vote blanc n'est pas autorisé, ce vecteur a pour longueur le nombre de choix ; lorsque le vote blanc est autorisé, ce vecteur possède un choix supplémentaire au début correspondant au fait que le vote est blanc ou non. Chaque choix est accompagné d'une preuve (dans **individual_proofs**, qui

a la même longueur que **choices**) qu'il est bien égal à 0 ou 1. La réponse totale est accompagnée de preuves que les choix respectent bien les contraintes de la question. Le champ **blank_proof** est optionnel et n'est présent que lorsque le vote blanc est autorisé.

La création d'une réponse chiffrée est faite par la fonction **create_answer** (table 3), qui prend la réponse en clair en paramètre *m*. Par exemple, dans le cas d'une question à 6 choix (A_1, \dots, A_6) où on peut choisir entre 2 et 4 choix ou voter blanc, *m* peut être [1,0,0,0,0,0] (vote blanc) ou [0,0,1,0,1,0,1] (vote pour A_2 , A_4 et A_6), mais pas [1,0,0,1,0,0,0] (vote blanc et pour A_3) ni [0,1,1,1,1,1,1] (vote pour tous les candidats). Note : la fonction **create_answer** échoue (en particulier, la création des preuves) lorsque *m* n'est pas valide.

5.4.2 Loria Bulletin de vote

$$\text{ballot} = \left\{ \begin{array}{l} \text{answers} : \text{answer}^* \\ \text{session_id} : \text{string} \\ \text{election_id} : \text{string} \end{array} \right\}$$

Un bulletin est un vecteur de réponses chiffrées telles que décrites dans la section 4.3 (une réponse par question), accompagné du numéro de session et de l'identifiant de l'élection. Une empreinte de cette structure peut être prise et présentée à l'électeur pour assurer une forme de vérifiabilité individuelle.

5.4.3 Mise en œuvre Voxcore

Chaque suffrage est défini par :

- Election.ordre : l'id de l'élection (circonscription électorale)
- Electeur.fk_etordre : l'id de la circonscription consulaire
- BV : le suffrage chiffré
- h : son empreinte (voir Vox_référencebulletin()) (nb : à ne pas confondre avec H majuscule)

L'identifiant de la circonscription électorale et l'identifiant de la circonscription consulaire sont ajoutés à la configuration du bulletin (champ UUID, session_id), afin que ce numéro soit également pris en compte dans la preuve associée au bulletin. Cela permet ainsi de détecter éventuellement un bulletin qui aurait été déplacé d'une urne à une autre, occasionnant alors un changement du numéro d'ordre de l'élection.

5.5 ACCUMULES

5.5.1 Loria Accumulation

$$\text{encrypted_tally} = \text{ciphertext}^{**}$$

Le résultat chiffré (calculé par la fonction compute_encrypted_tally en table 8) est le produit point à point des éléments chiffrés de tous les bulletins acceptés.

5.5.2 Mise en œuvre Voxcore

Résultat des accumulations, selon la maille définie.

Pour chaque maille, le résultat d'accumulation comprend

- Identifiant de l'élection (élection.ordre)
- Identifiant de la maille (ID_LEC)

- 1 compteur en clair global (nul+blanc+votes valablement exprimés)
- 1 compteur en clair pour les votes nuls
- 1 compteur chiffré de vote blanc
- Autant de compteurs chiffrés que de candidats

5.6 DECHIFFRES PARTIELS

5.6.1 Loria déchiffrement partiel

$$\text{partial_decryption} = \left\{ \begin{array}{l} \text{decryption_factors} : \mathbb{G}^{**} \\ \text{decryption_proofs} : \text{proof}^{**} \end{array} \right\}$$

À partir du résultat chiffré, \mathcal{S} calcule au nom de chaque assesseur \mathcal{T}_i un déchiffrement partiel (table 9) en utilisant sa clé secrète s_i et la clé publique associée $S_i = g^{s_i}$.

Cette structure se compose de facteurs de déchiffrement et de preuves qu'ils ont bien été calculés. Son calcul fait intervenir $\mathcal{H}_{\text{decrypt}}$, qui est défini comme suit :

$$\mathcal{H}_{\text{decrypt}}(X, A, B) = \text{SHA256}(\text{decrypt} | X | A, B) \pmod q$$

où `decrypt`, les barres verticales et les virgules sont littérales. Le résultat est interprété comme un nombre de 256 bits commençant par les poids forts.

Ces preuves sont vérifiées (table 10) en utilisant la clé publique S_i qui a été publiée par le serveur lors de la mise en place de l'élection.

5.6.2 Mise en œuvre Voxcore -modélisation

Un déchiffré partiel est défini au niveau de l'élection

- Election.ordre
- Ass.ordre
- Candidat.ordre ou 0 pour le vote blanc
- Déchiffré partiel

Et au niveau de la maille la plus fine

- Election.ordre
- Electeur.fk_etordre = pastille de vote (circonscription consulaire LEC)
- Ass.ordre
- Candidat.ordre ou 0 pour le vote blanc
- Déchiffré partiel

5.7 RESULTATS FINAUX

5.7.1 Loria Résultat de l'élection

$$\text{result} = \left\{ \begin{array}{l} \text{num_tallied} : \text{int} \\ \text{encrypted_tally} : \text{encrypted_tally} \\ \text{partial_decryptions} : \text{partial_decryption}^* \\ \text{result} : \text{int}^{**} \end{array} \right\}$$

Le résultat de l'élection est calculé par la fonction `compute_result` (table 11).

Après l'élection, les données suivantes doivent être publiées pour vérifier le dépouillement :

- E et S_1, \dots, S_m ;
- l'ensemble des bulletins acceptés ;
- la structure `result`.

Cette vérification est effectuée par la fonction `verify_result` (table 12).

5.7.2 Mise en œuvre voxcore

Résultat, pour chaque maille définie

- Identifiant de la maille (ID_LEC)
- Nb votes total (nuls+blancs+votes valablement exprimés). Est égal au nombre d'émargements sur le même périmètre
- Nb votes nuls (à priori zéro)
- Nb votes blancs
- Pour chaque candidat, nb votes reçus
- Variable `r` de type `result`

5.8 CACHET DE SIGNATURE DU SUFFRAGE PAR LE SERVEUR

Cachet généré par `Vox_créationcachetserveur_Preuve_de_vote()`

5.9 CONDITIONS PARTICULIERES LEGISLATIVES 2022

- 1 unique bureau de vote (et donc un unique jeu de clé publique chiffrement/clés privée assesseurs)
- 11 circonscriptions électorales, représentées par 11 élections
- Et au sein de ces 11 circonscriptions électorales, environ 200 circonscriptions consulaires,
- Une dizaine de candidats par élection (=circonscription électorale),

- Environ 1 400 000 électeurs,
- Chaque électeur est inscrit à une et seule circonscription électorale (et donc à 1 et 1 seule élection).

Gestion du second tour (le cas échéant) :

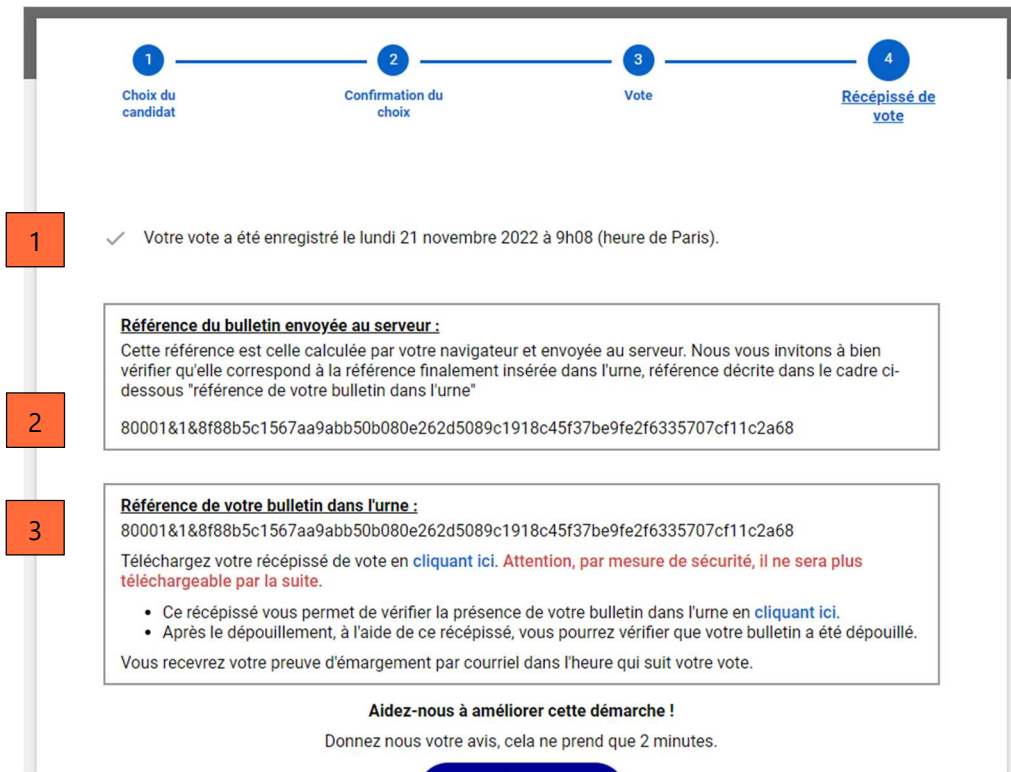
- Les candidats changent pour le second tour, au nombre de 2 dans la majorité des cas
- Le référentiel électeurs change également
- La clé publique de signature de cachet est réinitialisée

5.10 CONDITIONS PARTICULIERES LEGISLATIVES PARTIELLES 2023

- 3 circonscriptions électorales

6 PREUVES UTILISEES ET MISE A DISPOSITION DE TIERS

6.1 ACCUSE DE RECEPTION



1 Choix du candidat 2 Confirmation du choix 3 Vote 4 Récépissé de vote

1 ✓ Votre vote a été enregistré le lundi 21 novembre 2022 à 9h08 (heure de Paris).

2 **Référence du bulletin envoyée au serveur :**
 Cette référence est celle calculée par votre navigateur et envoyée au serveur. Nous vous invitons à bien vérifier qu'elle correspond à la référence finalement insérée dans l'urne, référence décrite dans le cadre ci-dessous "référence de votre bulletin dans l'urne"
 80001&1&8f88b5c1567aa9abb50b080e262d5089c1918c45f37be9fe2f6335707cf11c2a68

3 **Référence de votre bulletin dans l'urne :**
 80001&1&8f88b5c1567aa9abb50b080e262d5089c1918c45f37be9fe2f6335707cf11c2a68
 Téléchargez votre récépissé de vote en [cliquant ici](#). **Attention, par mesure de sécurité, il ne sera plus téléchargeable par la suite.**

- Ce récépissé vous permet de vérifier la présence de votre bulletin dans l'urne en [cliquant ici](#).
- Après le dépouillement, à l'aide de ce récépissé, vous pourrez vérifier que votre bulletin a été dépouillé.

Vous recevrez votre preuve d'émargement par courriel dans l'heure qui suit votre vote.

Aidez-nous à améliorer cette démarche !
 Donnez nous votre avis, cela ne prend que 2 minutes.

ID	Commentaire
1	Date et heure de l'émargement
2	Référence H du bulletin
3	Référence H'' du bulletin

Ces données sont envoyées via le flux #55.

6.3 PREUVE DU CACHET SERVEUR DE LA PREUVE DE DEPOT DU BULLETIN

La preuve fournie est le fichier JSON CachetSU autoporteur comprenant les différents éléments : infoSU, σ , hSkeySU, CléCachetSU.

6.4 PREUVE DE DECHIFFREMENT

Les preuves fournies sont

- La structure des élections
- Les clés publiques des assesseurs
- L'urne B
- Les accumulations
- Les différents déchiffrements partiels
- Les résultats finaux
- r de type result pour chaque élection

7 DETAILS ET PSEUDO_CODE DES FONCTIONS VOXALY

7.1 FONCTION VOX_SCHNORR_CREATIONKEY()

Entrée

- (rien)

Sortie

- Clé privée SignatureCachet.keypriv
- Clé publique SignatureCachet.keyPub

Traitement

- Ecckeypairgenerator() sur la courbe secp256r1, via la librairie Bouncycastle

7.2 FONCTION VOX_SCHNORR_CREATION_SIGNATURE()

Signe le message passé en paramètre avec la clé privée

Entrée :

- @param keyPair pair de clé de signature
- @param msg le message à signer

Sortie

- @return le couple de signature (e,s) sous forme de chaine hexa

Traitement:

```

public static String signMessage(AsymmetricCipherKeyPair keyPair, String msg) {
    ECPublicKeyParameters publicKey = (ECPublicKeyParameters) keyPair.getPublic();
    ECPrivateKeyParameters privateKey = (ECPrivateKeyParameters) keyPair.getPrivate();
    ECPoint G = publicKey.getParameters().getG();
    ECPoint Y = publicKey.getQ();
    BigInteger N = publicKey.getParameters().getN();
    BigInteger x = privateKey.getD();
    BigInteger t = RandomUtil.pickRandom(N);
    ECPoint U = G.multiply(t);
    // Hash(G,Y,U,msg)
    String msg2H =
    ECPointUtil.ECPointToHex(G).concat("%").concat(ECPointUtil.ECPointToHex(Y)).concat("%").concat(ECPointUtil.ECPointToHex
    x(U)).concat("%").concat(msg);
    BigInteger e = new BigInteger(DigestUtils.sha256Hex(msg2H), 16).mod(N);
    BigInteger s = t.subtract(e.multiply(x)).mod(N);
    return e.toString(32) + "%" + s.toString(32);
}

```

7.3 FONCTION VOX_SCHNORR_CONTROLE_SIGNATURE()

Vérifie que la signature générée pour le message msg est valide

Entrée

- @param publicKey clé de vérification
- @param msg msg d'origine
- @param signature signature à vérifier

Sortie

- @return true si la signature est valide, false sinon

Traitement :

```
public static boolean verifyForMessage(ECPublicKeyParameters publicKey, String msg, String signature) {
    String[] str = signature.split("%");
    BigInteger e = new BigInteger(str[0], 32);
    BigInteger s = new BigInteger(str[1], 32);
    ECPoint G = publicKey.getParameters().getG();
    ECPoint Y = publicKey.getQ();
    BigInteger N = publicKey.getParameters().getN();
    ECPoint U = G.multiply(s).add(Y.multiply(e));
    //      Hash(G,Y,U,V,msg)                                     String      msg2H      =
    ECPointUtil.ECPTtoHex(G).concat("%").concat(ECPointUtil.ECPTtoHex(Y)).concat("%").concat(ECPointUtil.ECPTtoHex(U)).concat("%").concat(msg);
    BigInteger ep = new BigInteger(DigestUtils.sha256Hex(msg2H), 16).mod(N);
    return ep.compareTo(e) == 0;
}
```

7.4 FONCTION VOX_ENCODAGE_CLEPRIVEEASSESEUR()

Permet de protéger la clé privée individuelle d'un assesseur

En entrée :

- S_i, la clé privée
- La phrase secrète PS

En sortie

- PS.E (empreinte de PS)
- Un fichier

Traitement:

- Dérivation de la passphrase pour obtenir une empreinte
 - ▶ Calcul du sel Sel_Passphrase, avec securerandom(), taille=16
 - ▶ dérivation= pbkdf2, sha256, 10 000 itérations, à partir de PS et sel_Passphrase
 - ▶ PS.E=s_{el}_passphrase+&+dérivation
 - ▶ « + » signifie concaténation des chaînes de caractères

- Dérivation de la passphrase pour obtenir la clé de chiffrement
 - ▶ Calcul du sel Sel_cléprivée, avec securerandom(), taille=16
 - ▶ key= pbkdf2, sha1, 10 000 itérations, length=128, à partir de PS et Sel_Cléprivée
- Chiffrement de la clé privée :
 - ▶ AES/CBC/PKCS5Padding
 - ▶ avec clé=Key
 - ▶ avec vecteur IV = généré avec secureRandom, taille=16
- puis encodage du chiffrement en base64
- écriture de sel_Cléprivée et résultat de l'encodage dans fichier
- gestion oubli : réécriture dans l'objet ayant contenu la clé puis libération objet

(méthode java=EnregistrerEmpreinteChiffrée())

7.5 FONCTION VOX_DECODAGECLEPRIVEEASSESEUR()

Permet de récupérer une clé privée individuelle (Mécanisme inverse de Vox_encodageCléPrivéeAssesseur())

En entrée :

- Un fichier
- La phrase secrète PS
- PS.E (empreinte de PS)

En sortie

- S_i, la clé privée

Traitement :

- Contrôle de l'empreinte
 - ▶ Extraction de sel_passphrase et dérivation) partir de PS.E
 - ▶ Dérivation'= pbkdf2, sha256, 10 000 itérations, à partir de PS' et sel_passphrase
 - ▶ Contrôle dérivation'=dérivation ?
- Extraction de Sel_Cléprivée du fichier
- Décodage base64 du chiffrement
 - ▶ Dérivation de la clé key'= pbkdf2, sha1, 10 000 itérations, length=128, à partir de PS et Sel_Cléprivée
- Déchiffrement du fichier contenant le chiffrement de la clé privée :
 - ▶ AES/CBC/PKCS5Padding
 - ▶ avec clé=key'
 - ▶ avec vecteur IV = récupéré dans le fichier

7.6 FONCTION VOX_REFERENCEBULLETIN()

Entrée :

- Bv= bulletin chiffré
- Tour.ordre = numéro d'ordre du tour (1 ou 2)

- Election.ordre = l'identifiant de l'élection
- Electeur.fk_etordre = pastille de vote (circonscription électorale LEC)

Sortie

- H

La référence du bulletin est calculée ainsi

1. $h = \text{sha256}(\text{Bv} + \text{tour.ordre} + \text{election.ordre} + \text{Electeur.fk_etordre})$
2. $\text{clé97} = \text{Vox_CléRIB97}(h)$
3. $H = \text{tour.ordre} + \ll \& \gg + \text{election.ordre} + \gg \& \gg + h + \text{clé97}$

NB : « + » = concaténation de chaînes

7.7 FONCTION VOX_CREATIONCACHETSERVEUR_PREUVE_DE_VOTE()

En entrée

- Election.ordre : Numéro ordre de l'élection
- Election.nom : Nom de l'élection
- Election.fk_etordre : Numéro d'ordre de la circonscription
- Bv= bulletin chiffré
- SignatureCachet.keyPub (la clé privée de signature)
- SignatureCachet.keypriv (la clé publique de signature)

Sortie :

- CachetSU

Le traitement de construction du cachet suit globalement le même principe que celui de l'émargement :

1. $\text{Clécontrôle} = \text{vox_CléRib97}(\text{sha256}(\text{bv}))$
2. $\text{infoSU} = \text{Election.tour} + \ll | \gg + \text{Election.ordre} + \ll | \gg + \text{Election.nom} + \ll | \gg + \text{Election.fk_etordre} + \ll | \gg + \text{sha256}(\text{Bv} + \text{tour.ordre} + \text{election.ordre} + \text{Electeur.fk_etordre}) + \ll | \gg + \text{Clécontrôle}$
3. $\text{hSU} = \text{sha256}(\text{infoSU})$
4. $\sigma = \text{vox_schnorr_Création_Signature}(\text{hSU}, \text{SignatureCachet.keypriv})$
5. $\text{CachetbrutSU} = \text{infoSU} + \sigma + \text{SignatureCachet.keypub}$
6. $\text{CléCachetSU} = \text{vox_CléRib97}(\text{CachetbrutSU})$

« + » signifie concaténation des chaînes de caractères

Le cachet serveur Suffrage (CachetSU) est la structure JSON comprenant les différents éléments : infoSU, σ , SignatureCachet.keypub CléCachetSU.

Contrairement à l'émargement, ce cachet N'est PAS stocké (afin de renforcer l'anonymat).

7.8 FONCTION VOX_CONTROLECACHETSERVEUR()

En entrée

- CachetSU

Sortie

- message

Cela consiste à vérifier que la signature (Σ ou σ) correspond bien aux données signées (infoEM ou infoSU).

Le traitement de vérification est commun quel que soit le cachet électronique transmis.

Afin de vérifier le cachet électronique, les opérations suivantes sont réalisées (exemple pris avec SU) :

- Contrôle de cohérence : vérifier que le contenu du cachet (CachetbrutSU) est toujours cohérent vis-à-vis de la clé modulo 97 (CléCachetSU)
- Contrôle de cohérence bis : vérifier que le hash de la clé publique (hSU) est bien identique au hash recalculé à partir de la clé publique (SkeyEmargement) fournie en entrée
- Recalcul du hash des données
 - ▶ $h'SU = sha256(infoSU)$
- Contrôle final : La signature (σ) est contrôlée avec la clé publique fournie (KSignatureSuffragePub) et le hash recalculé (h'SU), via Vox_Schnorr_Controle_signature()
- Si résultat est OK :
 - ▶ Le hash issu du cachet électronique est affiché en vert
 - ▶ il est affiché en bas de page :

 **Le contenu du cachet électronique confirme l'authenticité des informations mentionnées**

- S'ils sont différents :
 - ▶ Le hash issu du cachet électronique est affiché en rouge
 - ▶ il est affiché en bas de page :

 **L'authenticité des informations mentionnées n'est pas attestée par le contenu du cachet électronique**

7.9 FONCTION VOX_CLELIB97()

Cette fonctionnalité permet d'ajouter un contrôle de forme sur les données saisies : cela permet rapidement de détecter des erreurs de saisie.

Entrée :

- S (chaîne de caractères)

Sortie :

- clé97

Programme :

1. Transformation de S : Si S comporte des lettres, remplacer la lettre par son index de 1 à 9 dans l'alphabet (A=1,..., I=9, J=1, etc.), en laissant un saut entre R et S (R=9, S=2).
2. Si caractère non reconnu, index=0
3. Clé97 = $97 - (S * 100 \text{ modulo } 97)$.

8 DETAIL ET PSEUDO_CODE DES FONCTIONS LORIA

8.1 FONCTION_KG_SHAMIR (M,T)

Entrées

- m (de type `int`) : nombre total d'assesseurs
- t (de type `int`) : nombre tel que $t + 1$ assesseurs sont nécessaires pour déchiffrer

Sorties

- y (de type \mathbb{G}) : clé publique de l'élection
- S_1, \dots, S_m (de type \mathbb{G}) : clés publiques des assesseurs
- s_1, \dots, s_m (de type \mathbb{Z}_q) : clés secrètes des assesseurs

Programme

1. Tirer au hasard a_0, \dots, a_t dans \mathbb{Z}_q
2. Calculer $y = g^{a_0}$
3. Soit f le polynôme défini par $f(x) = a_0 + a_1x + \dots + a_tx^t$
4. Pour $i \in \{1, \dots, m\}$, calculer $s_i = f(i)$ et $S_i = g^{s_i}$

TABLE 2 – Fonction KG_Shamir(m, t)

8.2 FONCTION CREATE_ANSWER(Y, Q, S, M)

<p>Entrées</p> <ul style="list-style-type: none"> — y (de type \mathbb{G}) : clé publique de l'élection — Q (de type question) : structure associée à la question — S (de type string) : préfixe à utiliser dans les preuves — m (de type int^*) : réponse en clair <p>Sortie</p> <ul style="list-style-type: none"> — a (de type answer) : réponse chiffrée <p>Programme</p> <ol style="list-style-type: none"> 1. Si $Q[\text{blank}]$ est vrai, vérifier que $\text{len}(m) = \text{len}(Q[\text{answers}]) + 1$, sinon vérifier que $\text{len}(m) = \text{len}(Q[\text{answers}])$ 2. Soit r un tableau de même taille que m 3. Remplir r avec des valeurs aléatoires prises dans \mathbb{Z}_q 4. Définir $a[\text{choices}]$ à un tableau de même taille que m 5. Définir $a[\text{individual_proofs}]$ à un tableau de même taille que m 6. Pour $j \in \{0, \dots, \text{len}(m) - 1\}$: <ol style="list-style-type: none"> (a) Définir $a[\text{choices}][j]$ à $\text{eg_encrypt}(y, r[j], m[j])$ (table 4) (b) Soient $\alpha = a[\text{choices}][j][\text{alpha}]$ et $\beta = a[\text{choices}][j][\text{beta}]$ (c) Définir $a[\text{individual_proofs}][j]$ à $\text{iprove}(y, S, \alpha, \beta, r[j], m[j], \{0, 1\})$ (table 13) 7. Si $Q[\text{blank}]$ est faux, c'est-à-dire que le vote blanc n'est pas autorisé : <ol style="list-style-type: none"> (a) Définir $a[\text{overall_proof}]$ à $\text{oprove}(y, S, a[\text{choices}], r, m, Q[\text{min}], Q[\text{max}])$ (table 15) (b) $a[\text{blank_proof}]$ n'est pas défini 8. Si $Q[\text{blank}]$ est vrai, c'est-à-dire que le vote blanc est autorisé : <ol style="list-style-type: none"> (a) Définir $a[\text{overall_proof}]$ à $\text{obprove}(y, S, a[\text{choices}], r, m, Q[\text{min}], Q[\text{max}])$ (table 17) (b) Définir $a[\text{blank_proof}]$ à $\text{bprove}(y, S, a[\text{choices}], r, m)$ (table 21)
--

TABLE 3 – Fonction $\text{create_answer}(y, Q, S, m)$

8.3 FONCTION EG_ENCRYPT(Y, R, M)

<p>Entrées</p> <ul style="list-style-type: none">— y (de type \mathbb{G}) : clé publique de l'élection— r (de type \mathbb{Z}_q) : nombre aléatoire— m (de type <code>int</code>) : message à chiffrer <p>Sortie</p> <ul style="list-style-type: none">— e (de type <code>ciphertext</code>) : message chiffré <p>Programme</p> <ol style="list-style-type: none">1. Définir $e[\text{alpha}]$ à g^r2. Définir $e[\text{beta}]$ à $y^r g^m$

TABLE 4 – Fonction `eg_encrypt(y, r, m)`

8.4 FONCTION VERIFY_ANSWER(Y, S, Q, A)

Entrées

- y (de type `G`) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser dans les preuves
- Q (de type `question`) : structure associée à la question
- a (de type `answer`) : réponse chiffrée

Sortie

- un booléen représentant le succès ou l'échec de la vérification

Programme

1. Soit $n = \text{len}(Q[\text{answers}])$
2. Si $\text{len}(a[\text{choices}]) \neq n$, retourner faux
3. Si $\text{len}(a[\text{individual_proofs}]) \neq n$, retourner faux
4. Pour $j \in \{0, \dots, n - 1\}$:
 - (a) Soient $\alpha = a[\text{choices}][j][\text{alpha}]$ et $\beta = a[\text{choices}][j][\text{beta}]$
 - (b) Si $\text{check}(\alpha)$ ou $\text{check}(\beta)$ (table 11) retourne faux, retourner faux
 - (c) Si $\text{verify_iproof}(y, S, \alpha, \beta, \{0, 1\}, a[\text{individual_proofs}][j])$ (table 14) retourne faux, retourner faux
5. Si $Q[\text{blank}]$ est faux, c'est-à-dire que le vote blanc n'est pas autorisé :
 - (a) Si $\text{verify_oproof}(y, S, a[\text{choices}], Q[\text{min}], Q[\text{max}], a[\text{overall_proof}])$ (table 16) retourne faux, retourner faux
6. Si $Q[\text{blank}]$ est vrai, c'est-à-dire que le vote blanc est autorisé :
 - (a) Si $\text{verify_obproof}(y, S, a[\text{choices}], Q[\text{min}], Q[\text{max}], a[\text{overall_proof}])$ (table 20) retourne faux, retourner faux
 - (b) Si $\text{verify_bproof}(y, S, a[\text{choices}], a[\text{blank_proof}])$ (table 22) retourne faux, retourner faux
7. Retourner vrai

TABLE 5 – Fonction $\text{verify_answer}(y, S, Q, a)$

8.5 FONCTION CREATE_BALLOT(E, N, SU_CLAIR)

Nb : pour éviter des confusions dans la notation, $V = \text{SU_clair}$ dans la description ci-dessous

<p>Entrées</p> <ul style="list-style-type: none">— E (de type <code>election</code>) : structure associée à l'élection— N (de type <code>string</code>) : numéro de session communiqué par le serveur (frais pour chaque bulletin)— V (de type <code>int**</code>) : réponses aux questions (par exemple, lorsque le vote blanc n'est pas autorisé, $V[i][j]$ vaut 1 si la réponse j de la question i a été choisie et 0 sinon) <p>Sortie</p> <ul style="list-style-type: none">— b (de type <code>ballot</code>) : bulletin chiffré <p>Programme</p> <ol style="list-style-type: none">1. Soit $y = E[\text{public_key}][y]$2. Définir $b[\text{election_id}]$ à $E[\text{election_id}]$3. Définir $b[\text{session_id}]$ à N4. Soit S la chaîne composée de $b[\text{election_id}]$ concaténée avec $b[\text{session_id}]$5. Définir $b[\text{answers}]$ à un nouveau tableau de même taille que $E[\text{questions}]$6. Pour $i \in \{0, \dots, \text{len}(E[\text{questions}]) - 1\}$:<ul style="list-style-type: none">— définir $b[\text{answers}][i]$ à <code>create_answer(y, E[questions][i], S, V[i])</code> (table 3)
--

TABLE 6 – Fonction `create_ballot(E, N, V)`

8.6 FONCTION VERIFY_BALLOT(E, B)

Entrées

- E (de type `election`) : structure associée à l'élection
- b (de type `ballot`) : bulletin chiffré

Sortie

- un booléen représentant le succès ou l'échec de la vérification

Programme

1. Soit $y = E[\text{public_key}][y]$
2. Si $b[\text{election_id}] \neq E[\text{election_id}]$, retourner faux
3. Si $b[\text{answers}]$ et $E[\text{questions}]$ n'ont pas la même taille, retourner faux
4. Soit S la chaîne composée de $b[\text{election_id}]$ concaténée avec $b[\text{session_id}]$
5. Pour $i \in \{0, \dots, \text{len}(E[\text{questions}]) - 1\}$:
 - si $\text{verify_answer}(y, S, E[\text{questions}][i], b[\text{answers}][i])$ (table 5) retourne faux, retourner faux
6. Retourner vrai

TABLE 7 – Fonction `verify_ballot(E, b)`

8.7 FONCTION COMPUTE_ENCRYPTED_TALLY(E, BV)

Entrées

- E (de type `election`) : structure de l'élection
- $(b_v)_{v \in \mathcal{B}}$ (de type `ballot*`) : bulletins acceptés (à dépouiller)

Sortie

- e (de type `encrypted_tally`) : résultat chiffré

Programme

1. Définir e à un tableau de taille $\text{len}(E[\text{questions}])$
2. Pour $i \in \{0, \dots, \text{len}(E[\text{questions}]) - 1\}$:
 - (a) Définir $e[i]$ à un tableau de taille $\text{len}(E[\text{questions}][i])$
 - (b) Pour $j \in \{0, \dots, \text{len}(E[\text{questions}][i]) - 1\}$:

- i. Calculer

$$\alpha = \prod_{v \in \mathcal{B}} b_v[\text{answers}][i][\text{choices}][j][\text{alpha}]$$

et

$$\beta = \prod_{v \in \mathcal{B}} b_v[\text{answers}][i][\text{choices}][j][\text{beta}]$$

- ii. Définir $e[i][j][\text{alpha}]$ à α et $e[i][j][\text{beta}]$ à β .

TABLE 8 – Fonction `compute_encrypted_tally(E, (b_v)_{v \in \mathcal{B}})`

8.8 FONCTION COMPUTE_PARTIAL_DECRYPTION(Π, s, S)

Entrées

- Π (de type `encrypted_tally`) : résultat chiffré
- s (de type \mathbb{Z}_q) : clé secrète d'assesseur
- S (de type \mathbb{G}) : clé publique d'assesseur

Sortie

- d (de type `partial_decryption`) : déchiffrement partiel

Programme

1. Définir $d[\text{decryption_factors}]$ à un tableau de taille $\text{len}(\Pi)$
2. Définir $d[\text{decryption_proofs}]$ à un tableau de taille $\text{len}(\Pi)$
3. Pour $i \in \{0, \dots, \text{len}(\Pi) - 1\}$:
 - (a) Définir $d[\text{decryption_factors}][i]$ à un tableau de taille $\text{len}(\Pi[i])$
 - (b) Définir $d[\text{decryption_proofs}][i]$ à un tableau de taille $\text{len}(\Pi[i])$
 - (c) Pour $j \in \{0, \dots, \text{len}(\Pi[i]) - 1\}$:
 - i. Soit $\alpha = \Pi[i][j][\text{alpha}]$
 - ii. Définir $d[\text{decryption_factors}][i][j]$ à α^s
 - iii. Tirer au hasard $w \in \mathbb{Z}_q$
 - iv. Calculer $A = g^w$, $B = \alpha^w$ et $c = \mathcal{H}_{\text{decrypt}}(S, A, B)$
 - v. Définir $d[\text{decryption_proofs}][i][j][\text{challenge}]$ à c
 - vi. Définir $d[\text{decryption_proofs}][i][j][\text{response}]$ à $w + s \times c \pmod q$

TABLE 9 – Fonction `compute_partial_decryption(Π, s, S)`

8.9 FONCTION VERIFY_PARTIAL_DECRYPTION(Π , S , D)

Entrées

- Π (de type `encrypted_tally`) : résultat chiffré
- S (de type \mathbb{G}) : clé publique d'assesseur
- d (de type `partial_decryption`) : déchiffrement partiel

Sortie

- un booléen représentant le succès ou l'échec de la vérification

Programme

1. Pour $i \in \{0, \dots, \text{len}(\Pi) - 1\}$:
 - (a) Pour $j \in \{0, \dots, \text{len}(\Pi[i]) - 1\}$:
 - i. Soient $F = d[\text{decryption_factors}][i][j]$ et $\pi = d[\text{decryption_proofs}][i][j]$
 - ii. Soient $c = \pi[\text{challenge}]$ et $r = \pi[\text{response}]$
 - iii. Calculer

$$A = \frac{g^r}{S^c} \quad \text{et} \quad B = \frac{\Pi[i][j][\text{alpha}]^r}{F^c}$$
 - iv. Si $\mathcal{H}_{\text{decrypt}}(S, A, B) \neq c$, retourner faux
2. Retourner vrai

TABLE 10 – Fonction `verify_partial_decryption(Π, S, d)`

8.10 FONCTION COMPUTE_RESULT($E, N, \Pi, \mathcal{I}, \Delta$)

Entrées

- E (de type `election`) : structure de l'élection
- N (de type `int`) : nombre de bulletins acceptés (à dépouiller)
- Π (de type `encrypted_tally`) : résultat chiffré
- \mathcal{I} (de type `int*`) : indices des assesseurs qui participent au dépouillement
- Δ (de type `partial_decryption*`, de même taille que \mathcal{I}) : déchiffrements partiels

Sortie

- r (de type `result`) : résultat de l'élection

Programme

1. Si $\text{len}(\mathcal{I}) \neq E[\text{threshold}]$ ou $\text{len}(\Delta) \neq E[\text{threshold}]$, retourner une erreur
2. Définir $r[\text{num_tallied}]$ à N
3. Définir $r[\text{encrypted_tally}]$ à Π
4. Définir $r[\text{partial_decryptions}]$ à Δ
5. Définir $r[\text{result}]$ à un tableau de taille $\text{len}(E[\text{questions}])$
6. Pour $i \in \{0, \dots, \text{len}(E[\text{questions}]) - 1\}$:

(a) Définir $r[\text{result}][i]$ à un tableau de taille $\text{len}(E[\text{questions}][i])$

(b) Pour $j \in \{0, \dots, \text{len}(E[\text{questions}][i]) - 1\}$:

i. Calculer

$$F = \prod_{k=0}^{\text{len}(\mathcal{I})-1} (\Delta[k][\text{decryption_factors}][i][j])^{\lambda_k^{\mathcal{I}}}$$

où les $\lambda_k^{\mathcal{I}}$ sont les coefficients de Lagrange :

$$\lambda_k^{\mathcal{I}} = \prod_{l \in \{0, \dots, \text{len}(\mathcal{I})-1\} \setminus \{k\}} \frac{\mathcal{I}[l]}{\mathcal{I}[l] - \mathcal{I}[k]} \pmod q$$

ii. Calculer

$$R = \log_g \left(\frac{\Pi[i][j][\text{beta}]}{F} \right)$$

Ici, le logarithme discret peut être calculé facilement car il est borné par N .

iii. Définir $r[\text{result}][i][j]$ à R

TABLE 11 – Fonction `compute_result($E, N, \Pi, \mathcal{I}, \Delta$)`

8.11 FONCTION VERIFY_RESULT(E, S, B, R)

Entrées

- E (de type **election**) : structure de l'élection
- S (de type \mathbb{G}^*) : clés publiques des assesseurs
- B (de type **ballot***) : bulletins acceptés
- r (de type **result**) : résultat de l'élection (à vérifier)

Sortie

- un booléen, qui est vrai si le résultat de l'élection est correct

Programme

1. Soit $N = r[\text{num_tallied}]$; si $\text{len}(B) \neq N$, retourner faux
2. Définir D à l'ensemble vide
3. Pour $b \in B$:
 - (a) Si $b[\text{session_id}] \in D$, retourner faux
 - (b) Ajouter $b[\text{session_id}]$ à D
 - (c) Si $\text{verify_ballot}(E, b)$ retourne faux, retourner faux
4. Soit $\Pi = r[\text{encrypted_tally}]$; si $\text{compute_encrypted_tally}(E, B) \neq \Pi$, retourner faux
5. Soit $\Delta = r[\text{partial_decryptions}]$; si $\text{len}(\Delta) \neq E[\text{threshold}]$, retourner faux
6. Définir \mathcal{I} à un tableau de même taille que Δ , rempli de -1
7. Pour $l \in \{0, \dots, \text{len}(\Delta) - 1\}$:
 - (a) Pour $i \in \{0, \dots, \text{len}(S) - 1\}$:
 - Si $\text{verify_partial_decryption}(\Pi, S[i], \Delta[l])$, définir $\mathcal{I}[l]$ à i
 - (b) Si $\mathcal{I}[l] = -1$, retourner faux
8. Si \mathcal{I} contient plusieurs fois le même indice, retourner faux
9. Si $\text{compute_result}(E, N, \Pi, \mathcal{I}, \Delta) \neq r$, retourner faux
10. Retourner vrai

TABLE 12 – Fonction $\text{verify_result}(E, S, B, r)$

8.12 FONCTION IPROVE(Y, S, A, B, R, M, M)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- α, β (chacun de type \mathbb{G}) : message chiffré sur lequel porte la preuve
- r (de type \mathbb{Z}_q) : nombre aléatoire utilisé pour créer le chiffré (α, β)
- m (de type `int`) : message en clair
- M (de type `int*`) : valeurs possibles du message en clair

Sortie

- π (de type `proof*`) : preuve d'appartenance du message en clair à M

Programme

1. Soit i tel que $m = M[i]$; si un tel i n'existe pas, renvoyer une erreur
2. Soit $k = \text{len}(M) - 1$; on a donc $0 \leq i \leq k$
3. Définir π à un tableau de taille $k + 1$
4. Pour $j \in \{0, \dots, k\}$:
 - Si $j \neq i$:
 - (a) Tirer C et R au hasard dans \mathbb{Z}_q
 - (b) Définir $\pi[j][\text{challenge}]$ à C et $\pi[j][\text{response}]$ à R
 - (c) Calculer :

$$A_j = \frac{g^R}{\alpha^C} \quad \text{et} \quad B_j = \frac{y^R}{(\beta/g^{M[j]})^C}$$

5. Tirer au hasard $w \in \mathbb{Z}_q$
6. Calculer $A_i = g^w$ et $B_i = y^w$
7. Calculer :

$$C = \mathcal{H}_{\text{iprove}}(S, \alpha, \beta, A_0, B_0, \dots, A_k, B_k) - \sum_{j \neq i} \pi[j][\text{challenge}] \pmod q$$

8. Calculer $R = w + r \times C \pmod q$
9. Définir $\pi[i][\text{challenge}]$ à C et $\pi[i][\text{response}]$ à R

TABLE 13 – Fonction $\text{iprove}(y, S, \alpha, \beta, r, m, M)$

8.13 FONCTION VERIFY_IPROOF(y, S, A, B, M, π)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- α, β (chacun de type \mathbb{G}) : message chiffré sur lequel porte la preuve
- M (de type `int*`) : valeurs possibles du message en clair
- π (de type `proof*`) : preuve à vérifier

Sortie

- un booléen, qui est vrai si la preuve est correcte

Programme

1. Si M et π n'ont pas la même taille, retourner faux
2. Soit $k = \text{len}(M) - 1$
3. Pour $j \in \{0, \dots, k\}$, calculer :

$$A_j = \frac{g^{\pi[j][\text{response}]}}{\alpha^{\pi[j][\text{challenge}]}} \quad \text{et} \quad B_j = \frac{y^{\pi[j][\text{response}]}}{(\beta/g^{M[j]})^{\pi[j][\text{challenge}]}}$$

4. Calculer :

$$C = \sum_{j=0}^k \pi[j][\text{challenge}] \quad \text{mod } q$$

5. Si $\mathcal{H}_{\text{iprove}}(S, \alpha, \beta, A_0, B_0, \dots, A_k, B_k) \neq C$, retourner faux
6. Retourner vrai

TABLE 14 – Fonction `verify_iproof($y, S, \alpha, \beta, M, \pi$)`

8.14 FONCTION OPROVE(Y, S, E, R, M, A, B)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- e (de type `ciphertext*`) : tableau de messages chiffrés sur lequel porte la preuve
- r (de type \mathbb{Z}_q^* , de même taille que e) : nombres aléatoires utilisés pour créer e
- m (de type `int*`, de même taille que e) : messages en clair
- a (de type `int`) : nombre minimal de 1 dans m
- b (de type `int`) : nombre maximal de 1 dans m

Sortie

- π (de type `proof*`) : preuve que m respecte bien les contraintes imposées par a et b

Programme

1. Calculer :

$$\alpha = \prod_i e[i][\text{alpha}] \quad \text{et} \quad \beta = \prod_i e[i][\text{beta}]$$

2. Calculer :

$$R = \sum_i r[i] \pmod q \quad \text{et} \quad M = \sum_i m[i] \pmod q$$

3. Retourner `iprove(y, S, alpha, beta, R, M, {a, ..., b})`

TABLE 15 – Fonction `oprove(y, S, e, r, m, a, b)`

8.15 FONCTION VERIFY_OPROOF(y, S, E, A, B, Π)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- e (de type `ciphertext*`) : tableau de messages chiffrés sur lequel porte la preuve
- a (de type `int`) : nombre minimal de 1 dans le message en clair
- b (de type `int`) : nombre maximal de 1 dans le message en clair
- π (de type `proof*`) : preuve à vérifier

Sortie

- un booléen, qui est vrai si la preuve est correcte

Programme

1. Calculer :

$$\alpha = \prod_i e[i][\text{alpha}] \quad \text{et} \quad \beta = \prod_i e[i][\text{beta}]$$

2. Retourner `verify_iproof($y, S, \alpha, \beta, \{a, \dots, b\}, \pi$)`

TABLE 16 – Fonction `verify_oproof(y, S, e, a, b, π)`

8.16 FONCTION OBPROVE(y, S, e, r, m, a, b)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- e (de type `ciphertext*`) : tableau de messages chiffrés sur lequel porte la preuve
- r (de type \mathbb{Z}_q^* , de même taille que e) : nombres aléatoires utilisés pour créer e
- m (de type `int*`, de même taille que e) : messages en clair
- a, b (de type `int`) : contraintes min et max pour les votes non blancs

Sortie

- π (de type `proof*`) : preuve à mettre dans le champ `overall_proof` de la réponse chiffrée

Programme

1. Soit $k = \text{len}(e) - 1$
2. $e[0]$ indique donc s'il s'agit d'un vote blanc, et $e[1], \dots, e[k]$ représentent les choix
3. Soient $\alpha_0 = e[0][\text{alpha}]$ et $\beta_0 = e[0][\text{beta}]$
4. Calculer :

$$\alpha_\Sigma = \prod_{i=1}^k e[i][\text{alpha}] \quad \text{et} \quad \beta_\Sigma = \prod_{i=1}^k e[i][\text{beta}]$$

5. Soit P la chaîne " $g, y, \alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma$ "
6. Soient $r_0 = r[0]$ et $m_0 = m[0]$
7. Calculer :

$$r_\Sigma = \sum_{i=1}^k r[i] \pmod q \quad \text{et} \quad m_\Sigma = \sum_{i=1}^k m[i] \pmod q$$

8. Soit $M = \{a, \dots, b\}$; la preuve retournée sera une preuve de $m_0 = 1 \vee m_\Sigma \in M$
9. Si $m_0 = 0$ (i.e. dans $m_0 = 1 \vee m_\Sigma \in M$, le deuxième cas est vrai) :
 - Retourner `obprove0(y, S, P, $\alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma, r_\Sigma, m_\Sigma, M$)` (table [18](#))
10. Si $m_0 = 1$ (i.e. dans $m_0 = 1 \vee m_\Sigma \in M$, le premier cas est vrai) :
 - Retourner `obprove1(y, S, P, $\alpha_\Sigma, \beta_\Sigma, r_0, M$)` (table [19](#))

TABLE 17 – Fonction `obprove(y, S, e, r, m, a, b)`

8.17 FONCTION $\text{OBPROVE}_0(Y, S, P, A_0, B_0, A_\Sigma, B_\Sigma, R_\Sigma, M_\Sigma, M)$

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S, P (de type `string`) : préfixes à utiliser
- α_0, β_0 (chacun de type \mathbb{G}) : message chiffré correspondant au bit blanc
- $\alpha_\Sigma, \beta_\Sigma$ (chacun de type \mathbb{G}) : message chiffré correspondant aux bits non blancs
- r_Σ (de type \mathbb{Z}_q) : nombre aléatoire utilisé pour calculer $(\alpha_\Sigma, \beta_\Sigma)$
- m_Σ (de type `int`) : message en clair associé à $(\alpha_\Sigma, \beta_\Sigma)$
- M (de type `int*`) : valeurs possibles de m_Σ

Sortie

- π (de type `proof*`) : preuve utilisée dans `obprove` dans le cas d'un vote non blanc

Programme

1. Soit i tel que $m_\Sigma = M[i - 1]$; si un tel i n'existe pas, renvoyer une erreur
2. Soit $k = \text{len}(M)$; on a donc $1 \leq i \leq k$
3. Définir π à un tableau de taille $k + 1$
4. Tirer C et R au hasard dans \mathbb{Z}_q
5. Définir $\pi[0][\text{challenge}]$ à C et $\pi[0][\text{response}]$ à R
6. Calculer $A_0 = g^R \times \alpha_0^C$ et $B_0 = y^R \times (\beta_0/g)^C$
7. Pour $j \in \{1, \dots, k\}$:
 - Si $j \neq i$:
 - (a) Tirer C et R au hasard dans \mathbb{Z}_q
 - (b) Définir $\pi[j][\text{challenge}]$ à C et $\pi[j][\text{response}]$ à R
 - (c) Calculer $A_j = g^R \times \alpha_\Sigma^C$ et $B_j = y^R \times (\beta_\Sigma/g^{M[j-1]})^C$
8. Tirer au hasard w dans \mathbb{Z}_q
9. Calculer $A_i = g^w$ et $B_i = y^w$
10. Calculer :

$$C = \mathcal{H}_{\text{bproof}}(S, P, A_0, B_0, \dots, A_k, B_k) - \sum_{j \in \{0, \dots, k\} \setminus \{i\}} \pi[j][\text{challenge}] \pmod q$$

11. Calculer $R = w - r_\Sigma \times C \pmod q$
12. Définir $\pi[i][\text{challenge}]$ à C et $\pi[i][\text{response}]$ à R

TABLE 18 – Fonction $\text{obprove}_0(y, S, P, \alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma, r_\Sigma, m_\Sigma, M)$

8.18 FONCTION OBPVE1(Y, S, P, AΣ, BΣ, R0, M)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S, P (de type **string**) : préfixes à utiliser
- $\alpha_\Sigma, \beta_\Sigma$ (chacun de type \mathbb{G}) : message chiffré
- r_0 (de type \mathbb{Z}_q) : nombre aléatoire utilisé pour chiffrer le bit blanc
- M (de type **int***) : valeurs possibles du total des choix non blancs

Sortie

- π (de type **proof***) : preuve utilisée dans **obprove** dans le cas d'un vote blanc

Programme

1. Soit $k = \text{len}(M)$
2. Définir π à un tableau de taille $k + 1$
3. Pour $j \in \{1, \dots, k\}$:
 - (a) Tirer C et R au hasard dans \mathbb{Z}_q
 - (b) Définir $\pi[j][\text{challenge}]$ à C et $\pi[j][\text{response}]$ à R
 - (c) Calculer $A_j = g^R \times \alpha_\Sigma^C$ et $B_j = y^R \times (\beta_\Sigma / g^{M[j-1]})^C$
4. Tirer au hasard w dans \mathbb{Z}_q
5. Calculer $A_0 = g^w$ et $B_0 = y^w$
6. Calculer :

$$C = \mathcal{H}_{\text{bproof1}}(S, P, A_0, B_0, \dots, A_k, B_k) - \sum_{j=1}^k \pi[j][\text{challenge}] \pmod q$$

7. Calculer $R = w - r_0 \times C \pmod q$
8. Définir $\pi[0][\text{challenge}]$ à C et $\pi[0][\text{response}]$ à R

TABLE 19 – Fonction $\text{obprove}_1(y, S, P, \alpha_\Sigma, \beta_\Sigma, r_0, M)$

8.19 FONCTION VERIFY_OBPROOF(Y, S, E, A, B, Π)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- e (de type `ciphertext*`) : tableau de messages chiffrés sur lequel porte la preuve
- a, b (de type `int`) : contraintes min et max pour les votes non blancs
- π (de type `proof*`) : preuve à vérifier

Sortie

- un booléen, qui est vrai si la preuve est correcte

Programme

1. Soit $k = \text{len}(e) - 1$
2. $e[0]$ indique donc s'il s'agit d'un vote blanc et $e[1], \dots, e[k]$ représentent les choix
3. Soient $\alpha_0 = e[0][\text{alpha}]$ et $\beta_0 = e[0][\text{beta}]$
4. Calculer :

$$\alpha_\Sigma = \prod_{i=1}^k e[i][\text{alpha}] \quad \text{et} \quad \beta_\Sigma = \prod_{i=1}^k e[i][\text{beta}]$$

5. À des fins d'explications, notons m_0 (resp. m_Σ) le déchiffré de (α_0, β_0) (resp. $(\alpha_\Sigma, \beta_\Sigma)$) ; bien sûr, m_0 et m_Σ ne peuvent pas être calculés sans la clé de déchiffrement
6. Soit P la chaîne " $g, y, \alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma$ "
7. Soit $M = \{a, \dots, b\}$; la preuve à vérifier est une preuve de $m_0 = 1 \vee m_\Sigma \in M$
8. Soit $n = \text{len}(M)$
9. Si $\text{len}(\pi) \neq n + 1$, retourner faux
10. Calculer :

$$A_0 = g^{\pi[0][\text{response}]} \times \alpha_0^{\pi[0][\text{challenge}]} \quad \text{et} \quad B_0 = y^{\pi[0][\text{response}]} \times (\beta_0/g)^{\pi[0][\text{challenge}]}$$

11. Pour $j \in \{1, \dots, n\}$, calculer :

$$A_j = g^{\pi[j][\text{response}]} \times \alpha_\Sigma^{\pi[j][\text{challenge}]} \quad \text{et} \quad B_j = y^{\pi[j][\text{response}]} \times (\beta_\Sigma/g^{M[j-1]})^{\pi[j][\text{challenge}]}$$

12. Calculer :

$$C = \sum_{j=0}^n \pi[j][\text{challenge}] \quad \text{mod } q$$

13. Si $\mathcal{H}_{\text{bproof1}}(S, P, A_0, B_0, \dots, A_n, B_n) \neq C$, retourner faux
14. Retourner vrai

TABLE 20 – Fonction `verify_obproof(y, S, e, a, b, pi)`

8.20 FONCTION BPROVE(Y, S, E, R, M)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- e (de type `ciphertext*`) : tableau de messages chiffrés sur lequel porte la preuve
- r (de type \mathbb{Z}_q^* , de même taille que e) : nombres aléatoires utilisés pour créer e
- m (de type `int*`, de même taille que e) : messages en clair

Sortie

- π (de type `proof*`) : preuve à mettre dans le champ `blank_proof` de la réponse chiffrée

Programme

1. Soit $k = \text{len}(e) - 1$
2. $e[0]$ indique donc s'il s'agit d'un vote blanc, et $e[1], \dots, e[k]$ représentent les choix
3. Soient $\alpha_0 = e[0][\text{alpha}]$ et $\beta_0 = e[0][\text{beta}]$
4. Calculer :

$$\alpha_\Sigma = \prod_{i=1}^k e[i][\text{alpha}] \quad \text{et} \quad \beta_\Sigma = \prod_{i=1}^k e[i][\text{beta}]$$

5. Soit P la chaîne " $g, y, \alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma$ "

6. Soient $r_0 = r[0]$ et $m_0 = m[0]$

7. Calculer :

$$r_\Sigma = \sum_{i=1}^k r[i] \pmod q \quad \text{et} \quad m_\Sigma = \sum_{i=1}^k m[i] \pmod q$$

8. La preuve retournée sera une preuve de $m_0 = 0 \vee m_\Sigma = 0$

9. Définir π à un tableau de taille 2

10. Si $m_0 = 0$ (i.e. dans $m_0 = 0 \vee m_\Sigma = 0$, le premier cas est vrai) :

- Tirer C et R au hasard dans \mathbb{Z}_q
- Définir $\pi[1][\text{challenge}]$ à C et $\pi[1][\text{response}]$ à R
- Calculer $A_\Sigma = g^R \times \alpha_\Sigma^C$ et $B_\Sigma = y^R \times \beta_\Sigma^C$
- Tirer w au hasard dans \mathbb{Z}_q
- Calculer $A_0 = g^w$ et $B_0 = y^w$
- Calculer $C = \mathcal{H}_{\text{bproof}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) - \pi[1][\text{challenge}] \pmod q$
- Calculer $R = w - r_0 \times C \pmod q$
- Définir $\pi[0][\text{challenge}]$ à C et $\pi[0][\text{response}]$ à R

11. Si $m_0 = 1$ (i.e. dans $m_0 = 0 \vee m_\Sigma = 0$, le deuxième cas est vrai) :

- Tirer C et R au hasard dans \mathbb{Z}_q
- Définir $\pi[0][\text{challenge}]$ à C et $\pi[0][\text{response}]$ à R
- Calculer $A_0 = g^R \times \alpha_0^C$ et $B_0 = y^R \times \beta_0^C$
- Tirer w au hasard dans \mathbb{Z}_q
- Calculer $A_\Sigma = g^w$ et $B_\Sigma = y^w$
- Calculer $C = \mathcal{H}_{\text{bproof}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) - \pi[0][\text{challenge}] \pmod q$
- Calculer $R = w - r_\Sigma \times C \pmod q$
- Définir $\pi[1][\text{challenge}]$ à C et $\pi[1][\text{response}]$ à R

TABLE 21 – Fonction `bprove(y, S, e, r, m)`

8.21 FONCTION VERIFY_BPROOF(y, S, e, π)

Entrées

- y (de type \mathbb{G}) : clé publique de l'élection
- S (de type `string`) : préfixe à utiliser
- e (de type `ciphertext*`) : tableau de messages chiffrés sur lequel porte la preuve
- π (de type `proof*`) : preuve à vérifier

Sortie

- un booléen, qui est vrai si la preuve est correcte

Programme

1. Soit $k = \text{len}(e) - 1$
2. $e[0]$ indique donc s'il s'agit d'un vote blanc, et $e[1], \dots, e[k]$ représentent les choix
3. Soient $\alpha_0 = e[0][\text{alpha}]$ et $\beta_0 = e[0][\text{beta}]$
4. Calculer :

$$\alpha_\Sigma = \prod_{i=1}^k e[i][\text{alpha}] \quad \text{et} \quad \beta_\Sigma = \prod_{i=1}^k e[i][\text{beta}]$$

5. À des fins d'explications, notons m_0 (resp. m_Σ) le déchiffré de (α_0, β_0) (resp. $(\alpha_\Sigma, \beta_\Sigma)$) ; bien sûr, m_0 et m_Σ ne peuvent être calculés sans la clé de déchiffrement
6. Soit P la chaîne " $g, y, \alpha_0, \beta_0, \alpha_\Sigma, \beta_\Sigma$ "
7. La preuve à vérifier est une preuve de $m_0 = 0 \vee m_\Sigma = 0$
8. Si $\text{len}(\pi) \neq 2$, retourner faux
9. Calculer $A_0 = g^{\pi[0][\text{response}]} \times \alpha_0^{\pi[0][\text{challenge}]}$ et $B_0 = y^{\pi[0][\text{response}]} \times \beta_0^{\pi[0][\text{challenge}]}$
10. Calculer $A_\Sigma = g^{\pi[1][\text{response}]} \times \alpha_\Sigma^{\pi[1][\text{challenge}]}$ et $B_\Sigma = y^{\pi[1][\text{response}]} \times \beta_\Sigma^{\pi[1][\text{challenge}]}$
11. Calculer $C = \pi[0][\text{challenge}] + \pi[1][\text{challenge}] \pmod q$
12. Si $\mathcal{H}_{\text{bproof}}(S, P, A_0, B_0, A_\Sigma, B_\Sigma) \neq C$, retourner faux
13. Retourner vrai

TABLE 22 – Fonction `verify_bproof(y, S, e, π)`

9 DETAIL IMPLEMENTATION

9.1 ZOOM SUR SUR VERIFY_ANSWER() ET CALCUL DE H_IPROOF

9.1.1 Exemple code JAVA

```

public class HIprouve<GE> {
    private static final String PROVE = "prove";
    private static final String SEPARATOR = "|";
    private static final String COMMA = ",";

    private final GroupParameters<GE> groupParameters;

    public HIprouve(final GroupParameters<GE> groupParameters) {
        this.groupParameters = groupParameters;
    }

    /**
     * Permet de générer un HIprouve. Chapitre 4.8 Proofs of interval membership
     * @param S
     * @param alpha
     * @param beta
     * @param elements <code>CipherText</code> contenant le couple Ai, Bi
     * @return
     */
    public BigInteger apply(final String S, final GroupElement<GE> alpha, final GroupElement<GE> beta, final Iterator<CipherText<GE>> elements) {
        final HProofBuilder<GE> builder = new HProofBuilder<>(this.groupParameters).add(PROVE).add(SEPARATOR).add(S).add(SEPARATOR).add(alpha).add(COMMA).add(beta);
        boolean isFirst = true;
        while (elements.hasNext()) {
            final CipherText<GE> ct = elements.next();
            final String separator = isFirst ? SEPARATOR : COMMA;
            builder.add(separator).add(ct.getAlpha()).add(COMMA).add(ct.getBeta());
            isFirst = false;
        }
        return builder.toProof();
    }
}

```

```

public BigInteger toProof() {
    String proofable = ""; proofable: "prove|966cce38-35b6-4daf-b015-1b8b7151d3cbf22fbc38a4e8b493a8c252a5e2b07b8|198243171545
    for (ProofElement pe : this.proofElements) { proofElements: size = 15
        proofable += pe.toProofElement();
    }
    LOG.trace("Building proof of : {}" + proofable);
    final byte[] hash = DigestUtils.getSha256Digest().digest(proofable.getBytes(StandardCharsets.US_ASCII)); hash: {68, -62, -
    return new BigInteger(signum: 1, hash).mod(Q); hash: {68, -62, -6, 37, -85, 105, 92, -37, 45, -3, +22 more} Q: "723700557
}

```

9.1.2 Exemple numérique pour h_iproof

9.1.2.1 zkp

zkp : 966cce38-35b6-4daf-b015-1b8b7151d3cbf22fbc38a4e8b493a8c252a5e2b07b8

9.1.2.2 ECCGroupParameters

```

groupParameters = {ECCGroupParameters@22029}
  type = {CryptoGroupType@22033} "ECC"
  p = {BigInteger@22022} "57896044618658097711785492504343953926634992332820282019728792003956564819949"
  q = {BigInteger@22034} "37095705934669439343138083508754565189542113879843219016388785533085940283555"
  d = {BigInteger@16927} "7237005577332262213973186563042994240857116359379907606001950938285454250989"
  g = {ECCGroupElement@21593} "ECCGroup [value=Point [x= 15112221349535400772501151409588531511454012693041857206046113283949847762202, y=46316835694926478169428394003475163141307993866256225615783033603165251855960]]"
  one = {ECCGroupElement@22023} "ECCGroup [value=Point [x=0, y=1]]"
    
```

p : 57896044618658097711785492504343953926634992332820282019728792003956564819949

d : 37095705934669439343138083508754565189542113879843219016388785533085940283555

q : 7237005577332262213973186563042994240857116359379907606001950938285454250989

g :
 Point x : 15112221349535400772501151409588531511454012693041857206046113283949847762202
 Point y : 46316835694926478169428394003475163141307993866256225615783033603165251855960

one :
 Point x : 0
 Point y : 1

9.1.2.3 electionWrappedPk

```

electionWrappedPk = {ElectionWrappedPk@22219}
  y = {ECCGroupElement@16746} "ECCGroup [value=Point [x= 22351796739323114692988081049508178660559120399198023083571547644031189722301, y= 52901664300892456029245896610078702813052207202341497069387476977313549473386]]"
  p = {BigInteger@22022} "57896044618658097711785492504343953926634992332820282019728792003956564819949"
  q = {BigInteger@16927} "7237005577332262213973186563042994240857116359379907606001950938285454250989"
  g = {ECCGroupElement@21593} "ECCGroup [value=Point [x= 15112221349535400772501151409588531511454012693041857206046113283949847762202, y= 46316835694926478169428394003475163141307993866256225615783033603165251855960]]"
  one = {ECCGroupElement@22023} "ECCGroup [value=Point [x=0, y=1]]"
    
```

y :
 Point x : 22351796739323114692988081049508178660559120399198023083571547644031189722301
 Point y : 52901664300892456029245896610078702813052207202341497069387476977313549473386

p : 57896044618658097711785492504343953926634992332820282019728792003956564819949

q : 7237005577332262213973186563042994240857116359379907606001950938285454250989

g :
 Point x : 15112221349535400772501151409588531511454012693041857206046113283949847762202
 Point y : 46316835694926478169428394003475163141307993866256225615783033603165251855960

one :
 Point x : 0
 Point y : 1

9.1.2.4 Détail Hlprove

```

alpha = {ECCGroupElement@22228} "ECCGroup [value=Point [x= 19824317154564977412154415151877599828990784769087263279553600529004167968, y= 3655567889991447057986901280053476485581014477104020364410969464280883191653]]"
  groupParameters = {ECCGroupParameters@22029}
  A = {ECCGroupElement@lambda@22323}
  value = {Point@22333} "Point [x= 19824317154564977412154415151877599828990784769087263279553600529004167968, y= 3655567889991447057986901280053476485581014477104020364410969464280883191653]"
  beta = {ECCGroupElement@22229} "ECCGroup [value=Point [x= 2275459760050001246309668352590852635837408670628647340099652043915596389055, y= 55318545159327487817383683197199468744864812281159615691790205779687084603364]]"
  groupParameters = {ECCGroupParameters@22029}
  A = {ECCGroupElement@lambda@23733}
  value = {Point@23734} "Point [x= 2275459760050001246309668352590852635837408670628647340099652043915596389055, y= 55318545159327487817383683197199468744864812281159615691790205779687084603364]"
  elements = {CommitmentsIterator@23700}
  iterator = {ArrayList@23736}
  cursor = 2
  lastRet = 1
  expectedModCount = 2
  this$0 = {ArrayList@23737} size = 2
  0 = {CipherText@CipherTextImpl@23739} "CipherText [alpha=ECCGroup [value=Point [x= 23820904264970460888463209482924709491291013997289376200842034433319431208428, y= 94227988387504559060748580433021633366284650175601873600819806601864782888911], beta=ECCGroup [value=Point [x= 2275459760050001246309668352590852635837408670628647340099652043915596389055, y= 55318545159327487817383683197199468744864812281159615691790205779687084603364]], one=ECCGroup [value=Point [x= 0, y= 1]]]"
  1 = {CipherText@CipherTextImpl@23740} "CipherText [alpha=ECCGroup [value=Point [x= 10893630340918126010532205477997370989110464763473741867606818310282580978608, y= 3815365564655554137652151843545421744630129590061486235601215840685153297603]], beta=ECCGroup [value=Point [x= 2275459760050001246309668352590852635837408670628647340099652043915596389055, y= 55318545159327487817383683197199468744864812281159615691790205779687084603364]], one=ECCGroup [value=Point [x= 0, y= 1]]]"
    
```

S : 966cce38-35b6-4daf-b015-1b8b7151d3cbf22fbc38a4e8b493a8c252a5e2b07b8

alpha :

point x : 1982431715456497741215441515187759982899078476908726332795536085289004167968

point y : 36558678895914470579869012880534764685581014477104020364410969464280883191653

beta :

point x : 22754597600500012463096683525908526358374086706286473400996520439155596389055

point y : 55318545159527487817383683197199468744864812281159615691790205779687084603364

elements :

cipherText 1 / alpha / point x : 23820904264970460988436209482924709491291013997289376200842024433319431208428

cipherText 1 / alpha / point y : 9422799838750455906074858043302163336284650175601873600819806601864782880911

cipherText 1 / beta / point x : 23855459863563298335288547455257917005902243890932435777430923144142253632186

cipherText 1 / beta / point y : 43085577834330016744903436937994262540206487554031721362571209909966530423882

cipherText 2 / alpha / point x : 10893630340918126010532205477997370969110464763473741867606818310282580978608

cipherText 2 / alpha / point y : 38153655646565954137652151843545421744630129590061486235601215840685153297603

cipherText 2 / beta / point x : 1313783604613484423250795032016209311795955055879534322502940250227666746763

cipherText 2 / beta / point y : 3307417169239539200581679770929942290087670044954115493151257452418303403379

```

public BigInteger toProof() {
    String proofable = "";
    for (ProofElement pe : this.proofElements) {
        proofable += pe.toProofElement();
    }
    LOG.trace("Building proof of : {}" + proofable);
    final byte[] hash = DigestUtils.getSha256Digest().digest(proofable.getBytes(StandardCharsets.US_ASCII));
    return new BigInteger(signum: 1, hash).mod(Q);
}

```

Proofable : prove|966cce38-35b6-4daf-b015-

1b8b7151d3cbf22fbc38a4e8b493a8c252a5e2b07b8|1982431715456497741215441515187759982899078476908726332795536085289004167968-

36558678895914470579869012880534764685581014477104020364410969464280883191653,22754597600500012463096683525908526358374086706286473400996520439155596389055-

55318545159527487817383683197199468744864812281159615691790205779687084603364|23820904264970460988436209482924709491291013997289376200842024433319431208428-

9422799838750455906074858043302163336284650175601873600819806601864782880911,23855459863563298335288547455257917005902243890932435777430923144142253632186-

43085577834330016744903436937994262540206487554031721362571209909966530423882,10893630340918126010532205477997370969110464763473741867606818310282580978608-

38153655646565954137652151843545421744630129590061486235601215840685153297603,1313783604613484423250795032016209311795955055879534322502940250227666746763-3307417169239539200581679770929942290087670044954115493151257452418303403379

```

// HiProve
final BigInteger iprove = new HiProve<>(electionWrappedPK).apply(zkp, alphaBeta.getAlpha(), iprove: "2153746177056542023018996616324666950199192991853126252453931267471772207670",
alphaBeta.getBeta(), new CommitmentsIterator<GE>(commitments));
return iprove.equals(totalChallenge.get());

```

Iprove = 2153746177056542023018996616324666950199192991853126252453931267471772207670

9.2 ZOOM SUR VERIFY_ANSWER() ET CALCUL DE H_BPROOF0 ET H_BPROOF1

La variable h_bproof0 est décrite dans verify_bproof().

La variable `h_bproof1` est décrite dans `verify_obproof()`.

9.2.1 Exemple de code JAVA

`HBproof0` et `HBproof1` étendent tous les deux la même classe **HBproofAbstract**

```
package com.docapost.evot.e.crypto.api.loria.proof;

import com.docapost.evot.e.crypto.commons.group.GroupParameters;

public class HBproof0<GE> extends HBproofAbstract<GE> {
    private static final String PROVE = "bproof0";

    public HBproof0(final GroupParameters<GE> groupParameters) {
        super(groupParameters, PROVE);
    }
}
```

```
package com.docapost.evot.e.crypto.api.loria.proof;

import com.docapost.evot.e.crypto.commons.group.GroupParameters;

public class HBproof1<GE> extends HBproofAbstract<GE> {
    private static final String PROVE = "bproof1";

    public HBproof1(final GroupParameters<GE> groupParameters) {
        super(groupParameters, PROVE);
    }
}
```

```
public abstract class HbproofAbstract<GE> {
    private static final String SEPARATOR = "|";
    private static final String COMMA = ",";

    private final GroupParameters<GE> groupParameters;
    private final String prefix;

    public HbproofAbstract(final GroupParameters<GE> groupParameters, final String proofPrefix) {
        this.groupParameters = groupParameters;
        this.prefix = proofPrefix;
    }

    /**
     * Permet de générer un Hbproof0 Chapitre 4.9.1 Computing blank proof ou Hbproof1 Chapitre 4.9.1 Computing overall
     * proof
     * @param S
     * @param Pelements BigInteger constitutifs de P
     * @param elements <code>CipherText</code> contenant le couple Ai, Bi
     * @return
     */
    public BigInteger apply(final String S, final Iterator<GroupElement<GE>> Pelements, final Iterator<CipherText<GE>> elements) {
        final HProofBuilder<GE> builder = new HProofBuilder<>(this.groupParameters).add(this.prefix).add(SEPARATOR).add(S).add(SEPARATOR);
        // Pelements
        boolean isFirst = true;
        while (Pelements.hasNext()) {
            final GroupElement<GE> ct = Pelements.next();
            if (!isFirst) {
                builder.add(COMMA);
            }
            builder.add(ct);
            isFirst = false;
        }
        builder.add(SEPARATOR);

        isFirst = true;
        while (elements.hasNext()) {
            final CipherText<GE> ct = elements.next();
            if (!isFirst) {
                builder.add(COMMA);
            }
            builder.add(ct.getAlpha()).add(COMMA).add(ct.getBeta());
            isFirst = false;
        }
        return builder.toProof();
    }
}
```



```

public BigInteger toProof() {
    String proofable = "";
    for (ProofElement pe : this.proofElements) {
        proofable += pe.toProofElement();
    }
    LOG.trace("Building proof of : {}" + proofable);
    final byte[] hash = DigestUtils.getSha256Digest().digest(proofable.getBytes(StandardCharsets.US_ASCII));
    return new BigInteger( signature, hash).mod(Q);
}

```

9.2.2 Exemple numérique pour HBproof0

```

// 3. check that Hbproof0(S, P, A0, B0, Asigma, Bsigma) = challenge(pi0) + challenge(pisigma) mod q
final BigInteger h = new HBproof0<GE>(electionWrappedPk).apply(zpk, p.iterator(), new CommitmentsIterator<>(commitments));
total_challenges = total_challenges.mod(Q);
return h.equals(total_challenges);

```

9.2.2.1 En entrée

zpk : c3eb5f4b-c422-40c7-9b05-f3be32718ffac7f1060a0b99848a477c55174353e8dc

electionWrappedPk :

```

electionWrappedPk = [ElectionWrappedPk@24188]
  y = [ECCGroupElement@16746] "ECCGroup [value=Point [x=22351796739323114692988081049508178660559120399198023083571547644031189722301, y=52901664300892456029245896610078702813052207202341497069387476977313549473386]]"
  p = [BigInteger@22022] "57896044610658097711785492504343953926634992332820282019728792003956564819949"
  q = [BigInteger@16927] "7237005577332262213973186563042994240857116359379907606001950938285454250989"
  g = [ECCGroupElement@21593] "ECCGroup [value=Point [x=15112221349535400772501151409588531511454012693041857206046113283949847762202, y=46316835694926478169428394003475163141307993866256225615783033603165251855960]]"
  one = [ECCGroupElement@22023] "ECCGroup [value=Point [x=0, y=1]]"

```

y :
Point x : 22351796739323114692988081049508178660559120399198023083571547644031189722301
Point y : 52901664300892456029245896610078702813052207202341497069387476977313549473386

p : 57896044618658097711785492504343953926634992332820282019728792003956564819949

q : 7237005577332262213973186563042994240857116359379907606001950938285454250989

g :
Point x : 15112221349535400772501151409588531511454012693041857206046113283949847762202
Point y : 46316835694926478169428394003475163141307993866256225615783033603165251855960

one :
Point x : 0
Point y : 1

P :

```

P = (ArrayList<ArrayList<24190>) size = 6
  0 = [ECCGroupElement@21593] "ECCGroup [value=Point [x=15112221349535400772501151409588531511454012693041857206046113283949847762202, y=46316835694926478169428394003475163141307993866256225615783033603165251855960]]"
  1 = [ECCGroupElement@16746] "ECCGroup [value=Point [x=22351796739323114692988081049508178660559120399198023083571547644031189722301, y=52901664300892456029245896610078702813052207202341497069387476977313549473386]]"
  2 = [ECCGroupElement@24201] "ECCGroup [value=Point [x=4708511671278324273392989801448518578027287517765343312853619169123108577344, y=391967599748831068265233269282636550658882456415996180343077495723094074879848]]"
  3 = [ECCGroupElement@24202] "ECCGroup [value=Point [x=20360173152792688415114128682344169225472722714949157537717849466776421675719, y=39970368253188221663552711626372123125099456645420009268107129146892903670129]]"
  4 = [ECCGroupElement@24203] "ECCGroup [value=Point [x=882159843943476117873585618536098458216072609654656692429421115909520226929, y=29250518761112242806545695848515081656944211984366610748846135020799918755737]]"
  5 = [ECCGroupElement@24204] "ECCGroup [value=Point [x=43365927693485442856217988481378086634828425696908064194503966839043076434340, y=1687849130513090561505244902045346352700713962737188579829352923363731093281]]"

```

Point 0 x : 15112221349535400772501151409588531511454012693041857206046113283949847762202
Point 0 y : 46316835694926478169428394003475163141307993866256225615783033603165251855960

Point 1 x : 22351796739323114692988081049508178660559120399198023083571547644031189722301
 Point 1 y : 52901664300892456029245896610078702813052207202341497069387476977313549473386

Point 2 x : 47085116712783242733929898014485185780277287517765343312853619169123108577344
 Point 2 y : 39196759974883106826523326928263655065882456415996180343077495723094074879848

Point 3 x : 20360173152792688415114128682344169225472722714949157537717849466776421675719
 Point 3 y : 39970368253188221663552711626372123125099456645420009268107129146892903670129

Point 4 x : 8821598439434761178735856185360984582160726090654656692429421115909520226929
 Point 4 y : 29250518761112242806545695848515081656944211984366610748846135020799918755737

Point 5 x : 43365927693485442856217988481378088634828425696908064194503966839043076434340
 Point 5 y : 16878491305130905615052449020453463527007139627371885798293522923363731093281

Commitments :

```

commitments = (GroupElement[4]@24174)
0 = [ECCGroupElement@24182] "ECCGroup [value=Point [x=7364942946434817233813307527393340732547279520477991026008142750857925076534, y=33257597448649425653977375693474330793981791293304823682792394054892522205818]]"
1 = [ECCGroupElement@24184] "ECCGroup [value=Point [x=8641707145997617348694408803377131713943258910303881408361230160628404868985, y=33546897791516317250618560336616441877532271174732467843138602282922438400039]]"
2 = [ECCGroupElement@24185] "ECCGroup [value=Point [x=13146845636201970938174825455856110350470884624179519217471204128467365365205, y=42478332828408138194114709707705243697582060150998314896168120748462458845007]]"
3 = [ECCGroupElement@24187] "ECCGroup [value=Point [x=6119200759845557446295891801079966023403567020408969346246858885990648212237, y=42820400145557235828101017213064999824871244927254382924207601628556460108245]]"
    
```

Point 0 x : 7364942946434817233813307527393340732547279520477991026008142750857925076534
 Point 0 y : 33257597448649425653977375693474330793981791293304823682792394054892522205818

Point 1 x : 8641707145997617348694408803377131713943258910303881408361230160628404868985
 Point 1 y : 33546897791516317250618560336616441877532271174732467843138602282922438400039

Point 2 x : 13146845636201970938174825455856110350470884624179519217471204128467365365205
 Point 2 y : 42478332828408138194114709707705243697582060150998314896168120748462458845007

Point 3 x : 6119200759845557446295891801079966023403567020408969346246858885990648212237
 Point 3 y : 42820400145557235828101017213064999824871244927254382924207601628556460108245

9.2.2.2 Dans la méthode toProof()

Proofable : bproof0|c3eb5f4b-c422-40c7-9b05-f3be32718fac7f1060a0b99848a477c55174353e8dc|15112221349535400772501151409588531511454012693041857206046113283949847762202-46316835694926478169428394003475163141307993866256225615783033603165251855960,22351796739323114692988081049508178660559120399198023083571547644031189722301-52901664300892456029245896610078702813052207202341497069387476977313549473386,47085116712783242733929898014485185780277287517765343312853619169123108577344-39196759974883106826523326928263655065882456415996180343077495723094074879848,20360173152792688415114128682344169225472722714949157537717849466776421675719-39970368253188221663552711626372123125099456645420009268107129146892903670129,8821598439434761178735856185360984582160726090654656692429421115909520226929-29250518761112242806545695848515081656944211984366610748846135020799918755737,43365927693485442856217988481378088634828425696908064194503966839043076434340-16878491305130905615052449020453463527007139627371885798293522923363731093281|7364942946434817233813307527393340732547279520477991026008142750857925076534-33257597448649425653977375693474330793981791293304823682792394054892522205818,8641707145997617348694408803377131713943258910303881408361230160628404868985-33546897791516317250618560336616441877532271174732467843138602282922438400039,13146845636201970938174825455856110350470884624179519217471204128467365365205-42478332828408138194114709707705243697582060150998314896168120748462458845007,6119200759845557446295891801079966023403567020408969346246858885990648212237-42820400145557235828101017213064999824871244927254382924207601628556460108245

9.2.2.3 En sortie de HBproof0

h : 71378572008068508055096639234845013665306784989958276514497489143296628301

total_challenges : 71378572008068508055096639234845013665306784989958276514497489143296628301

9.2.3 Exemple numérique pour HBproof1

```
// 3. check that Hbproof1(S, P, A0, B0, . . . , Ak, Bk) = SIGMA(j=0 to k) challenge(pij) mod q
final BigInteger h = new HBproof1<>().apply(zkp, p.iterator(), new CommitmentsIterator<>().commitments());
total_challenges = total_challenges.mod(Q);

return h.equals(total_challenges);
```

9.2.3.1 En entrée

zkp : c3eb5f4b-c422-40c7-9b05-f3be32718ffac7f1060a0b99848a477c55174353e8dc

electionWrappedPk :

```
electionWrappedPk = {ElectionWrappedPk@24188}
  y = [ECCGroupElement@16746] "ECCGroup [value=Point [x=22351796739323114692988081049508178660559120399198023083571547644031189722301, y=52901664300892456029245896610078702813052207202341497069387476977313549473386]]"
  p = [BigInteger@22022] "57896044618658097711785492504343953926634992332820282019728792003956564819949"
  q = [BigInteger@16927] "7237005577332262213973186563042994240857116359379907606001950938285454250989"
  g = [ECCGroupElement@21593] "ECCGroup [value=Point [x=15112221349535400772501151409588531511454012693041857206046113283949847762202, y=46316835694926478169428394003475163141307993866256225615783033603165251855960]]"
  one = [ECCGroupElement@22023] "ECCGroup [value=Point [x=0, y=1]]"
```

y :
Point x : 22351796739323114692988081049508178660559120399198023083571547644031189722301
Point y : 52901664300892456029245896610078702813052207202341497069387476977313549473386

p : 57896044618658097711785492504343953926634992332820282019728792003956564819949

q : 7237005577332262213973186563042994240857116359379907606001950938285454250989

g :
Point x : 15112221349535400772501151409588531511454012693041857206046113283949847762202
Point y : 46316835694926478169428394003475163141307993866256225615783033603165251855960

one :
Point x : 0
Point y : 1

P :

```
P = (ArrayList@24190) size = 6
  0 = [ECCGroupElement@21593] "ECCGroup [value=Point [x=15112221349535400772501151409588531511454012693041857206046113283949847762202, y=46316835694926478169428394003475163141307993866256225615783033603165251855960]]"
  1 = [ECCGroupElement@16746] "ECCGroup [value=Point [x=22351796739323114692988081049508178660559120399198023083571547644031189722301, y=52901664300892456029245896610078702813052207202341497069387476977313549473386]]"
  2 = [ECCGroupElement@24201] "ECCGroup [value=Point [x=4708511671278324273392989801448518578027287517765343312853619169123108577344, y=39196759974883106826523326928263655065882456415996180343077495723094074879848]]"
  3 = [ECCGroupElement@24202] "ECCGroup [value=Point [x=2036017315279268841511412868234416922547272714949157537717849466776421675719, y=399703682531882216635271162637212312509945664542009268107129146892903670129]]"
  4 = [ECCGroupElement@24203] "ECCGroup [value=Point [x=8821598439434761178735856185360984582160726090654656692429421115909520226929, y=2925051876112242806545695848515081656944211984366610748846135020799918755737]]"
  5 = [ECCGroupElement@24204] "ECCGroup [value=Point [x=4336592769348544285621798848137808863482842569608064194503966839043076434340, y=16878491305130905615052449020453463527007139627371885798293522923363731093281]]"
```

Point 0 x : 15112221349535400772501151409588531511454012693041857206046113283949847762202
Point 0 y : 46316835694926478169428394003475163141307993866256225615783033603165251855960

Point 1 x : 22351796739323114692988081049508178660559120399198023083571547644031189722301
Point 1 y : 52901664300892456029245896610078702813052207202341497069387476977313549473386

Point 2 x : 4708511671278324273392989801448518578027287517765343312853619169123108577344
Point 2 y : 39196759974883106826523326928263655065882456415996180343077495723094074879848

Point 3 x : 20360173152792688415114128682344169225472722714949157537717849466776421675719
 Point 3 y : 39970368253188221663552711626372123125099456645420009268107129146892903670129

Point 4 x : 8821598439434761178735856185360984582160726090654656692429421115909520226929
 Point 4 y : 29250518761112242806545695848515081656944211984366610748846135020799918755737

Point 5 x : 43365927693485442856217988481378088634828425696908064194503966839043076434340
 Point 5 y : 16878491305130905615052449020453463527007139627371885798293522923363731093281

Commitments :

```

commitments = (GroupElement[6]@24339)
0 = (ECCGroupElement@24344) "ECCGroup [value=Point [x=37478355093319634067402596296064960200243064346395183082845999099438408912657, y=10602221433878933811234735055570741192105596989521604415861330515810741709367]]"
1 = (ECCGroupElement@24345) "ECCGroup [value=Point [x=37944633020287344532844274562564952043945696774659793712485421663460929022517, y=33159769629811546955893756711658239263489116611887378340317674129080047671109]]"
2 = (ECCGroupElement@24393) "ECCGroup [value=Point [x=37408311813549046489643477723132191679448774248656531403745039646121749050406, y=28697239020364439126329244839223132433571073810346459425390941636944016136096]]"
3 = (ECCGroupElement@24394) "ECCGroup [value=Point [x=1619886921949957037176610788050869341277704126923974356243128367083751066907, y=45032754184671893883021467165321276053813414194829766998270904426529387979881]]"
4 = (ECCGroupElement@24395) "ECCGroup [value=Point [x=1643607971831831422157540349472033691592738224664204090268350372571399483787, y=35246486448514342801627801435105171829059305714933742066335367440665562288188]]"
5 = (ECCGroupElement@24396) "ECCGroup [value=Point [x=48074145675201065751306623088565070325208852094658654370520329459938261150963, y=43037914938238311032210625702311792981876275555131688302901010308110808456243]]"
    
```

Point 0 x : 37478355093319634067402596296064960200243064346395183082845999099438408912657
 Point 0 y : 10602221433878933811234735055570741192105596989521604415861330515810741709367

Point 1 x : 37944633020287344532844274562564952043945696774659793712485421663460929022517
 Point 1 y : 33159769629811546955893756711658239263489116611887378340317674129080047671109

Point 2 x : 37408311813549046489643477723132191679448774248656531403745039646121749050406
 Point 2 y : 28697239020364439126329244839223132433571073810346459425390941636944016136096

Point 3 x : 1619886921949957037176610788050869341277704126923974356243128367083751066907
 Point 3 y : 45032754184671893883021467165321276053813414194829766998270904426529387979881

Point 4 x : 1643607971831831422157540349472033691592738224664204090268350372571399483787
 Point 4 y : 35246486448514342801627801435105171829059305714933742066335367440665562288188

Point 5 x : 48074145675201065751306623088565070325208852094658654370520329459938261150963
 Point 5 y : 43037914938238311032210625702311792981876275555131688302901010308110808456243

9.2.3.2 Dans la méthode toProof()

Proofable : bproof1|c3eb5f4b-c422-40c7-9b05-f3be32718ffac7f1060a0b99848a477c55174353e8dc|15112221349535400772501151409588531511454012693041857206046113283949847762202-46316835694926478169428394003475163141307993866256225615783033603165251855960,22351796739323114692988081049508178660559120399198023083571547644031189722301-52901664300892456029245896610078702813052207202341497069387476977313549473386,47085116712783242733929898014485185780277287517765343312853619169123108577344-39196759974883106826523326928263655065882456415996180343077495723094074879848,20360173152792688415114128682344169225472722714949157537717849466776421675719-39970368253188221663552711626372123125099456645420009268107129146892903670129,8821598439434761178735856185360984582160726090654656692429421115909520226929-29250518761112242806545695848515081656944211984366610748846135020799918755737,43365927693485442856217988481378088634828425696908064194503966839043076434340-16878491305130905615052449020453463527007139627371885798293522923363731093281|37478355093319634067402596296064960200243064346395183082845999099438408912657-10602221433878933811234735055570741192105596989521604415861330515810741709367,37944633020287344532844274562564952043945696774659793712485421663460929022517-33159769629811546955893756711658239263489116611887378340317674129080047671109,37408311813549046489643477723132191679448774248656531403745039646121749050406-28697239020364439126329244839223132433571073810346459425390941636944016136096,16198869219499570371766107880508693412777

04126923974356243128367083751066907-
45032754184671893883021467165321276053813414194829766998270904426529387979881,16436079718318314221575403494720336915927
382246642040902683503725751399483787-
35246486448514342801627801435105171829059305714933742066335367440665562288188,48074145675201065751306623088565070325208
852094658654370520329459938261150963-43037914938238311032210625702311792981876275555131688302901010308110808456243

9.2.3.3 En sortie de HBproof1

h : 4046842079809703264546892496670727560944237786896355203961160748676598328543

total_challenges : 4046842079809703264546892496670727560944237786896355203961160748676598328543

9.3 ZOOM SUR VERIFY_PARTIAL_DECRYPTION() ET LE CALCUL DE HDECRYPT

La variable hdecrypt est décrite dans verify_partial_decryption().

9.3.1 Exemple de code JAVA

Pour le calcul du hdecrypt :

- Concaténation
- Découpage dans un tableau d'octets (getBytes(StandardCharsets.US_ASCII))
- Hash 256 du tableau
- Application de modulo Q

```
public class HDecrypt<GE> {
    private static final String DECRYPT = "decrypt";
    private static final String SEPARATOR = "|";
    private static final String COMMA = ",";

    private final GroupParameters<GE> groupParameters;
    public HDecrypt(final GroupParameters<GE> groupParameters) {
        this.groupParameters = groupParameters;
    }

    public BigInteger apply(final GroupElement<GE> public_key, final GroupElement<GE> A, final GroupElement<GE> B) {
        final HProofBuilder<GE> builder = new HProofBuilder<>(this.groupParameters);
        builder.add(DECRYPT).add(SEPARATOR).add(public_key).add(SEPARATOR).add(A).add(COMMA).add(B);
        return builder.toProof();
    }
}
```

```

public BigInteger toProof() {
    String proofable = "";
    for (ProofElement pe : this.proofElements) {
        proofable += pe.toProofElement();
    }
    LOG.trace("Building proof of : {}" + proofable);
    final byte[] hash = DigestUtils.getSha256Digest().digest(proofable.getBytes(StandardCharsets.US_ASCII));
    return new BigInteger( signum: 1, hash).mod(Q);
}
    
```

9.3.2 Exemple numérique

Données en entrée

```

public_key = [ECCGroupElement@16132] "ECCGroup [value=Point [x=25975657525715006578031461139752016186868869900895106751373699167442474892, y=1590549649524245897822129474320739046592324898867311944271485410154561653897]]"
A = [ECCGroupElement@16987] "ECCGroup [value=Point [x=220765253883998355791414781128519540302319265538570694537312330431855841331, y=23205601944196511853789362950810014004362954144786747652009156633281461212032]]"
B = [ECCGroupElement@16988] "ECCGroup [value=Point [x=1018550985673351361558684341428944984399079737933882785548124155881826603958, y=57029227074986748347503514978198937257371989528218330031854911040357882024367]]"
builder = [HProofBuilder@16986]
    
```

Calcul de toProof

```

public BigInteger toProof() {
    String proofable = ""; proofable: "decrypt|25975657525715006578031461139752016186868869990895186751373699167442474892~15905496495242458978
    for (ProofElement pe : this.proofElements) { proofElements: size = 7
        proofable += pe.toProofElement();
    }
    LOG.trace("Building proof of : {}" + proofable); proofable: "decrypt|259756575257150065780314611397520161868688699908951867513736991674424
    final byte[] hash = DigestUtils.getSha256Digest().digest(proofable.getBytes(StandardCharsets.US_ASCII));
    return new BigInteger( signum: 1, hash).mod(Q);
}
    
```

Détail ProofElement

```

this = [HProofBuilder@16986]
proofElements = [ArrayList@2008] size = 7
0 = [HProofBuilder$Elem@17012]
  arg$1 = "decrypt"
1 = [HProofBuilder$Elem@17013]
  arg$1 = "T"
2 = [HProofBuilder$Elem@17014]
  arg$1 = [ECCGroupElement@16132] "ECCGroup [value=Point [x=25975657525715006578031461139752016186868869990895106751373699167442474892, y=1590549649524245897822129474320739046592324898867311944271485410154561653897]]"
3 = [HProofBuilder$Elem@17015]
  arg$1 = "T"
4 = [HProofBuilder$Elem@17016]
  arg$1 = [ECCGroupElement@16987] "ECCGroup [value=Point [x=220765253883998355791414781128519540302319265538570694537312330431855841331, y=23205601944196511853789362950810014004362954144786747652009156633281461212032]]"
5 = [HProofBuilder$Elem@17017]
  arg$1 = "T"
6 = [HProofBuilder$Elem@17018]
  arg$1 = [ECCGroupElement@16988] "ECCGroup [value=Point [x=1018550985673351361558684341428944984399079737933882785548124155881826603958, y=57029227074986748347503514978198937257371989528218330031854911040357882024367]]"
Q = [BigInteger@16256] "723700557733226213973186563042994240857116359379907606001950938285454250989"
proofable = "decrypt|25975657525715006578031461139752016186868869990895106751373699167442474892~1590549649524245897822129474320739046592324898867311944271485410154561653897|220765253883998355791414781128519540302319265538570694537312330431855841331~23205601944196511853789362950810014004362954144786747652009156633281461212032"
    
```

Détail de la chaîne Proofable

```
decrypt|259756575257150065780314611397520161868688699908895106751373699167442474892,
-1590549649524245897822129474320739046592324898867311944271485410154561653897,
|22076525388399898355791414781128519540302319265538570694537312330431855841331,
-23205601944196511853789362950810014004362954144786747652009156633281461212032,,
10185509856733513615586843414289449843990797337933882785548124155881826603958,
-57029227074986748347503514978198937257371989528218330031854911040357882024367
```

```
public BigInteger toProof() {
    String proofable = ""; proofable: "decrypt|259756575257150065780314611397520161868688699908895106751373699167442474892-159054964952424589782
    for (ProofElement pe : this.proofElements) { proofElements: size = 7
        proofable += pe.toProofElement();
    }
    LOG.trace("Building proof of : {}" + proofable);
    final byte[] hash = DigestUtils.getSha256Digest().digest(proofable.getBytes(StandardCharsets.US_ASCII)); hash: {-99, -4, -105, -45, -60, 36,
    return new BigInteger( signum: 1, hash).mod(Q); hash: {-99, -4, -105, -45, -60, 36, 116, 59, -24, 76, -22, 101, 6}; Q: "723708557733226221397314
```

Résultat final

- Au final Hdecrypt='6326...4009'

```
part = (PartialDecryption$Answer$Partial@21208) "Partial [alpha=ECCGroup [value=Point [x=30187018021727343703034363244545910459691176400754089962035468008861300040479, y=35711598349761619854402773800099502259687508113075997050033993718489652209685], de
alpha = [ECCGroupElement@21207] "ECCGroup [value=Point [x=30187018021727343703034363244545910459691176400754089962035468008861300040479, y=35711598349761619854402773800099502259687508113075997050033993718489652209685]]"
decryptionFactor = [ECCGroupElement@21209] "ECCGroup [value=Point [x=22081335968000797512975366441517075990826102563268078363448417271536126335912, y=31726021384966114654171826579043024086602557896430652666923957716109830945864]]"
proof = (Proof$ProofImpl@21210) "Proof [challenge:63263603648099335273625678419486288644022574379283016225344283353946124009, response:6012172072011839662160739698302627573363123646997507163586262764673650480232]"
challenge = (BigInteger@21239) "63263603648099335273625678419486288644022574379283016225344283353946124009"
response = (BigInteger@21240) "6012172072011839662160739698302627573363123646997507163586262764673650480232"
A = [ECCGroupElement@21004] "ECCGroup [value=Point [x=43514074292645164925873900260728315798307379429613291685896277346078830588089, y=28032644951657633173116895933577854319436833307073167820552763351755086428]]"
B = [ECCGroupElement@21005] "ECCGroup [value=Point [x=25276463472488153143813581059882379151093526207108249218543580547913285237123, y=367943100108328993420191346006902335034507716965617648929942456685752768408]]"
Hdecrypt = (BigInteger@21212) "63263603648099335273625678419486288644022574379283016225344283353946124009"
this.groupParameters = [ECCGroupParameters@16184]
```

Conclusion : Le résultat hdecrypt correspond bien au challenge porté dans la preuve.

9.4 ZOOM SUR ECPOINTUTIL.ECPTOHEX(G)

9.4.1 Méthode ECPTtoHex

```
// com.voxaly.verifiabilite.Convert an ECPoint to Hex-String
public static String ECPTtoHex(ECPoint p) {
    if (p.isInfinity()) return "Infinity%Infinity";

    StringBuilder sb = new StringBuilder();
    sb.append(p.normalize().getXCoord().toBigInteger().toString(16)).append("%").append(p.normalize().getYCoord().toBigInteger().toString(16));
    return sb.toString();
}
```


9.4.2 Exemple avec la vérification du Cachet Serveur

```

p = {ECPoint$Fp@17875} "(6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296,4fe342e2fe1a7f9b8ee7eb4
  ▶ f curve = {ECCurve$Fp@19728}
  ▶ f x = {ECFieldElement$Fp@19729} "6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296"
  ▶ f y = {ECFieldElement$Fp@19730} "4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5"
  ▶ f zs = {ECFieldElement[2]@19731}
  ▶ f preCompTable = {Hashtable@19733} size = 2
  
```

x : 6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296

y : 4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

sb.toString() :

6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296%4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

9.5 ZOOM SUR VOX_CREATIONCACHETSERVEUR_PREUVE_DE_VOTE(), GENERATION DU CACHET SERVEUR POUR LES SUFFRAGES

Les grandes étapes lors de la construction du cachet SU

- Récupération de la paire de clé pour la signature des suffrages
- Calcul de la clé RIB de l’empreinte du suffrage
- Création des informations pour le cachet SU
- Signature Schnorr du SHA256 des informations
- Mise en forme du cachet ‘brut’
(concaténation de ‘infoSU’ de la signature schnorr et de la clé publique des suffrages)
- Création d’un JSON

Les grandes étapes lors de la construction du cachet SU

- Récupération de la paire de clé pour la signature des suffrages
- Calcul de la clé RIB de l’empreinte du suffrage
- Création des informations pour le cachet SU
- Signature Schnorr du SHA256 des informations
- Mise en forme du cachet ‘brut’
(concaténation de ‘infoSU’ de la signature schnorr et de la clé publique des suffrages)
- Création d’un JSON

9.5.1 Exemple

9.5.1.1 Données en entrées

```
private static final String SEP = "|";
private static final String SEP_SKIPPED = "||";

private static final String DATE_PATTERN = "yyyyMMdd HH:mm:ss";
private static final CharSequence INFO_TYPE_SU = "infoSU";
```

Ordre de l'établissement électeur : 294

Empreinte du suffrage : 24074d65a1be1f6bdc9901ce7525b279ae50d6eabab884cad8a420a2d931c12a

Nom de l'élection : 11ème Circonscription des Français de l'étranger

L'ordre de l'élection : 11

```
int cleControle = getRibKey(empreinteSuffrageSHA256); cleControle: 12

// Création des informations pour le cachet SU
StringBuilder sb = new StringBuilder(); sb: "11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d
String normalizedName = StringUtil.sansAccent(election.getNom()).replaceAll(regex: "|", replacement: "_"); norm
sb.append(election.getOrdre()).append(SEP); election: Method threw 'org.hibernate.LazyInitializationExcepti
sb.append(normalizedName).append(SEP); normalizedName: "11eme_Circonscription_des_Francais_de_l'etranger"
sb.append(electeurEtablissementOrdre).append(SEP); electeurEtablissementOrdre: 294
sb.append(empreinteSuffrageSHA256).append(SEP); empreinteSuffrageSHA256: "24074d65a1be1f6bdc9901ce7525b279a
sb.append(String.format("%02d", cleControle)); cleControle: 12

String infoSU = sb.toString(); infoSU: "11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1b
```

9.5.1.2 En sortie

cleControle : 12

normalizedName : 11eme_Circonscription_des_Francais_de_l'etranger

```
String infoSU = sb.toString(); infoSU: "11|11eme_Circonscription_des_Fran
String hSu = DigestUtils.sha256Hex(infoSU); hSu: "d8d02a1435b5a6bb1e00fb0
String schnorr = SchnorrSignature.signMessage(suffrageKeyPair, hSu); schr
```

infoSU :

11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1be1f6bdc9901ce7525b279ae50d6eabab884ca
d8a420a2d931c12a|12

hSu : d8d02a1435b5a6bb1e00fb0a69977fff781a48794af84194cee28e015ce562cc

schnorr :

1kldpd3g122nljc0cp0ikmkt76vm1dca9p9g31mahc0u2glf14vt%19ceo57afu93h716dd6htlcjuagrck3asig39pl6g350iup
qtcvd

```
// Cachet 'brut'
sb.setLength(0);
sb.append(infoSU).append(schnorr).append(publicKeySu); infoSU: "11|11eme_
String cachetBrutSU = sb.toString(); cachetBrutSU: "11|11eme_Circonscription
String cleCachetBrutSU = String.format("%02d", getRibKey(cachetBrutSU));
```

cachetBrutSu :

11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1be1f6bdc9901ce7525b279ae50d6eabab884ca
d8a420a2d931c12a|121kldpd3g122nljc0cp0ikmkt76vm1dca9p9g31mahc0u2glf14vt%19ceo57afu93h716dd6htlcjuag
rck3asig39pl6g350iupqtcvd-----BEGIN VERIFICATION KEY-----

81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c43444f4%feb0342eb166fd09aca85c004637a9d66e
80a51fea0d5f40ce26a2f5e46b1c8f

-----END VERIFICATION KEY-----

cleCacheBRutSU : 08

```
// Création du JSON
CachetSuJSON cachetSuJSON = new CachetSuJSON(); cachetSuJSON: CachetSuJSON@21606
cachetSuJSON.setCleCachetBrut(cleCachetBrutSU); cleCachetBrutSU: "08"
cachetSuJSON.setInfoSU(infoSU); infoSU: "11|11eme_Circonscription_des_Francais_de_l
// Traitement dans le rapport jasper: remplacement des espaces dans la clé publique
cachetSuJSON.setPublicKeySu(publicKeySu.replaceAll( regex: " ", remplacement: "_")); publ
cachetSuJSON.setSchnorr(schnorr); schnorr: "1kldpd3g122nljc0cp0ikmkt76vm1dca9p9g31m
```

Nb :Pour la clé publique on remplace les espaces par des '_'

Cachet en 'clair' qui sera envoyé au pdf :

```
{"infoSU":"11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1be1f6bdc9901ce7525b279ae50d6e  
abab884cad8a420a2d931c12a|12","schnorr":"1kldpd3g122nljc0cp0ikmkt76vm1dca9p9g31mahc0u2glf14vt%19ceo5  
7afu93h716dd6htlcjuagrck3asig39pl6g350iupqtcvd","publicKeySu":"-----BEGIN_VERIFICATION_KEY-----  
\\n81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c43444f4%feb0342eb166fd09aca85c004637a9d  
66e80a51fea0d5f40ce26a2f5e46b1c8f\\n-----END_VERIFICATION_KEY-----","cleCachetBrut":"08"}
```

Pour l’envoyer au PDF, celui-ci sera encodé en Base64.

```
bean.setSignature(Base64.encodeBase64String(signature.getBytes(StandardCharsets.UTF_8))); | s1
```

Cachet :

eyJpbmZvU1UiOilxMXwxMwVtZV9DaXJjb25zY3JpcHRpb25fZGVzX0ZyYW5jYWlZx2RlX2wnZXRYyW5nZXJ8Mjk0fDI0
MDc0ZDY1YTFiZTFmNmJkYzk5MDFJZTc1MjViMjc5YWU1MGQ2ZWFiYWI4ODRjYWQ4YTQyMGEyZDkzMWwMmF8
MTiILCJzY2hub3JyJoiMwtsZHBkM2cxMjJubGpMGNwMGIrbWt0NzZ2bTFkY2E5cDlnMzFtYWVhMjY2xmMTR2dCU
xOWNlBzU3YVZ1OTNoNzE2ZGQ2aHRsY2p1YWdyY2szYXNpZmM5cGw2ZmM1MGI1cHF0Y3ZkliwicHVibGljS2V5U3Ui
OiltLS0tLUJFR0lOX1ZFUKlGSUNBVEIPTl9LRVktLS0tLVxyXG44MWE5Z3Zk2MwU2Mjc3NjhjNGY2MGJIN2Y0YmY3ZDJhZj
ZkZmY3YzI1M2I1YWU0MDRmZTljNDNmOWM0MzQ0NGY0JWZlYjAzNDJlYjE2NmZkMDIhY2E4NWMwMDQ2Mzdho
WQ2NmU4MGE1MWZlYTBknWY0MGNIMjZmMmY1ZTQ2YjFjOGZcclxuLS0tLS1FTkRfVkvSSUZjQ0FUSU9OX0tFWS0tL
S0tliwiY2xlQ2FjaGV0QnJ1dCI6IjA4In0

9.6 ZOOM SUR VOX_CONTROLECACHETSERVEUR(), VERIFICATION DU CACHET SERVEUR POUR LES SUFFRAGES

Les grandes étapes de la vérification sont :

- Contrôle de cohérence: vérifier que le contenu du cachet (CachetbrutEM) est toujours cohérent vis-à-vis de la clé modulo 97 (CléCachetEM)
- Contrôle final : Le cachet électronique brut (Σ) est contrôlé avec la clé publique fournie (SkeyEmargement) et le hash recalculé (h'EM)

9.6.1 Exemple

9.6.1.1 Données en entrées

Le cachet en 'clair' déjà décodé (base64Decode) :

```
{"infoSU":"11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1be1f6bdc9901ce7525b279ae50d6eabab884cad8a420a2d931c12a|12","schnorr":"1klpd3g122nljc0cp0ikmkt76vm1dca9p9g31mahc0u2glf14vt%19ceo57afu93h716dd6htlcjuagrck3asig39pl6g350iupqtcvd","publicKeySu":"-----BEGIN_VERIFICATION_KEY-----\r\n81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c43444f4%feb0342eb166fd09aca85c004637a9d66e80a51fea0d5f40ce26a2f5e46b1c8f\r\n-----END_VERIFICATION_KEY-----","cleCachetBrut":"08"}
```

9.6.1.2 Pré-traitement

- On transfère le JSON en format text dans un bean
- On remet la clé publique en forme (cf. capture d'écran ci-dessous)

pubKeySu : -----BEGIN VERIFICATION KEY-----

```
81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c43444f4%feb0342eb166fd09aca85c004637a9d66e80a51fea0d5f40ce26a2f5e46b1c8f
```

-----END VERIFICATION KEY-----

9.6.1.3 Traitement

On régénère la variable 'cachet brut SU' ainsi que sa clé RIB et on compare les clés RIB ensuite.

```
// generer cachetBrutSu
String pubKeySu = cachetSuJSON.getPublicKeySu().replaceAll( regex: "_", replacement: " "); pub
StringBuilder sb = new StringBuilder(); sb: "11|11eme_Circonscription_des_Francais_de_l'e
sb.append(cachetSuJSON.getInfoSU()).append(cachetSuJSON.getSchnorr()).append(pubKeySu); p
String cachetBrutSU = sb.toString(); cachetBrutSU: "11|11eme_Circonscription_des_Francais
String cleCachetBrutSU = String.format("%02d", getRibKey(cachetBrutSU)); cleCachetBrutSU:
```

cachetBrutSu :

```
11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1be1f6bdc9901ce7525b279ae50d6eabab884ca
```

d8a420a2d931c12a|121kldpd3g122nljc0cp0ikmkt76vm1dca9p9g31mahc0u2glf14vt%19ceo57afu93h716dd6htlcjuagrck3asig39pl6g350iupqtcvd-----BEGIN VERIFICATION KEY-----

81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c43444f4%feb0342eb166fd09aca85c004637a9d66e80a51fea0d5f40ce26a2f5e46b1c8f

-----END VERIFICATION KEY-----

cleCachetBrutSU : 08

Si les clés RIB sont identiques, on continue la vérification.

A partir de notre Bean, on relis les variables, **infoSu** et **schnorr**.

infoSu :

11|11eme_Circonscription_des_Francais_de_l'etranger|294|24074d65a1be1f6bdc9901ce7525b279ae50d6eabab884ca
d8a420a2d931c12a|12

schnorr :

1kldpd3g122nljc0cp0ikmkt76vm1dca9p9g31mahc0u2glf14vt%19ceo57afu93h716dd6htlcjuagrck3asig39pl6g350iupqtcvd

On recalcul l'empreinte de l'infoSU

hEm : d8d02a1435b5a6bb1e00fb0a69977fff781a48794af84194cee28e015ce562cc

Ensuite on vérifie la signature schnorr avec comme paramètre, la clé publique, l'empreinte de infoSu et la valeur de la signature.

```
ECPublicKeyParameters publicKeyParameters = eccKeyGen.loadSignaturePublicKey(new StringReader(pubKeySu));
boolean isValid = SchnorrSignature.verifyForMessage(publicKeyParameters, hEm, schnorr);
validationResult.put("isValid", isValid);
```

```
public static boolean verifyForMessage(ECPublicKeyParameters publicKey, String msg, String signature) {
    String[] str = signature.split(regex: "%");
    BigInteger e = new BigInteger(str[0], radix: 32);
    BigInteger s = new BigInteger(str[1], radix: 32);
    ECPoint G = publicKey.getParameters().getG();
    ECPoint Y = publicKey.getQ();
    BigInteger N = publicKey.getParameters().getN();
    ECPoint U = G.multiply(s).add(Y.multiply(e));
    // Hash(G, Y, U, V, msg)
    String msg2H = ECPointUtil.ECPointToHex(G).concat("%").concat(ECPointUtil.ECPointToHex(Y)).concat("%").concat(ECPointUtil.ECPointToHex(U)).concat("%").concat(msg);
    BigInteger ep = new BigInteger(DigestUtils.sha256Hex(msg2H), radix: 16).mod(N);
    return ep.compareTo(e) == 0;
}
```

e : 95292765713413301943077445525217322694116001722662332031434516964997847880701

s : 74883846457047897747863679106431385409599208694033059093651784061894863533037

G.x : 6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296

G.y : 4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

Y.x : 81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c43444f4

Y.y : feb0342eb166fd09aca85c004637a9d66e80a51fea0d5f40ce26a2f5e46b1c8f

N : 115792089210356248762697446949407573529996955224135760342422259061068512044369

U.x : 71c0b034884b40125b62f3c00d1e2c7bf2be16c5e20b8af3b4dadf56dfa14827

U.y : 2a201d370b846e577c9082c67a7488e3620a61a0ea5dc53b57f80a967a29c9b9

msg2h :

6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296%4fe342e2fe1a7f9b8ee7eb4a7c0f9e162
 bce33576b315ececbb6406837bf51f5%81a7e961e627768c4f60be7f4bf7d2af6dff7c253b5ae404fe9c43f9c434444f4%fe
 b0342eb166fd09aca85c004637a9d66e80a51fea0d5f40ce26a2f5e46b1c8f%a06844befd7b56c3b871a24735f431824e
 ea61da2001e06fee09bf957a7ba924%98d0a311293d275ba6c44a88ae6b9fd10c816ea15304d302395d30131efc3c68
 %d8d02a1435b5a6bb1e00fb0a69977fff781a48794af84194cee28e015ce562cc

ep : 95292765713413301943077445525217322694116001722662332031434516964997847880701

9.7 ZOOM SUR VOX_CLE_RIB97()

9.7.1 Code Java

```

public static int getRibKey(String rib) {
    StringBuilder extendedRib = new StringBuilder(rib.length());
    for (char currentChar : rib.toCharArray()) {
        //Works on base 36
        int currentCharValue = Character.digit(currentChar, Character.MAX_RADIX);

        // Valeur purement arbitraire, on vient remplacer les caractères non reconnu
        if (currentCharValue == -1) {
            currentCharValue = 0;
        }

        //Convert character to simple digit
        extendedRib.append(currentCharValue < 10 ? currentCharValue : (currentCharValue + (int) StrictMath.pow(2, (currentCharValue - 10) / 9)) % 10);
    }

    BigDecimal extendedRibInt = new BigDecimal(extendedRib.toString());
    return 97 - extendedRibInt.multiply(new BigDecimal(100)).toBigInteger().mod(new BigDecimal(97).toBigInteger()).intValue();
}

```

9.8 ZOOM SUR UUID

L'ordre de l'élection (election.ordre) et l'ordre de l'établissement de l'électeur (electeur.etablissementelecteur.ordre) sont stockés au format UUID (_java.util.UUID_) et cela est généré lors de la création de la structure _VoteConfiguration_ (du Loria).

Il s'agit d'une codification en base 16

Deux cas distincts existent pour la génération de l'UUID :

1. Lors de la création du bulletin de vote, lors de sa vérification et lors de l'accumulation (*permet la détection des permutations des bulletins, utilisation de la méthode `getUUIDfromElectionOrdreEtablissementElecteurOrdre()`*)
2. Pour le déchiffrement partiel et le dépouillement (*méthode `getOrdreElectionFromUUID()`*)

9.8.1 Exemple de code Java:

```

/**
 * Récupère un UUID basique depuis un élection Ordre
 * @param electionOrdre
 * @return
 */
public static UUID getUUIDfromElectionOrdre(int electionOrdre) {
    return new UUID( mostSigBits: 0, electionOrdre);
}

/**
 * Récupère un UUID basique depuis un élection Ordre et établissement Ordre
 * @param electionOrdre
 * @param etablisementElecteurOrdre
 * @return
 */
public static UUID getUUIDfromElectionOrdreEtablissementElecteurOrdre(int electionOrdre, int etablisementElecteurOrdre) {
    return new UUID(etablisementElecteurOrdre, electionOrdre);
}

```

Exemple :

Si election=11, alors la valeur est b ("electionUUID":"00000000-0000-0000-0000-00000000000b")

Si election = 10, alors la valeur est a, si etablisementElecteurOrdre = 1258, alors sa valeur est 4ea ("electionUUID":"00000000-0000-04ea-0000-00000000000a")

10 GLOSSAIRE

- B : urne contenant les bulletins exprimés
- EM : liste d'émargement
- m : le nombre total d'assesseurs,
- T₁, ..., T_m : les m assesseurs
- t : le seuil
- n : le nombre total d'électeurs
- V₁,...,V_n : les n électeurs
- V : un électeur
- PS : la phrase de secrète d'un assesseur
- s_i : la clé privée de l'assesseur i
- S_i : la clé publique de l'assesseur i
- E : la structure de l'élection. Contient la clé publique ChiffrementBulletin.keyPub
- N : le numéro de session
- ID_LEC : pastille rajoutée au bulletin au moment de son insertion dans l'urne pour permettre un comptage plus fin (=election.fk_etordre)
- H : empreinte du bulletin, au moment du chiffrement sur le poste de travail et stockée localement
- H' : empreinte du bulletin, calculée par le serveur lors du contrôle du code activation, vu par le poste de travail
- H'' : empreinte du bulletin, calculée par le serveur lors du vote proprement-dit, vu par le poste de travail
- σ : signature du bulletin (contenue dans CachetSU ou CachetEM)
- Time : timestamp émargement
- electeur.id : identifiant unique de l'électeur
- SignatureCachet.keyPub : la clé publique de signature de la preuve de vote
- SignatureCachet.keypriv : la clé privée de signature de la preuve de vote
- ChiffrementBulletin.keyPub : clé publique de chiffrement des suffrages (appelé également y dans la spécification Loria)
- ChiffrementBulletinTémoin.keyPub : clé publique de chiffrement des bulletins témoin
- ChiffrementBulletinTémoin.keyPriv : clé privée de déchiffrement des bulletins témoin