Preuve à divulgation nulle de connaissance pour la vérification d'éligibilité dans le vote par internet

Rapport de stage encadré par Alexandre Debant et Lucca Hirschi au Loria.

Voisin Benjamin ENS Rennes
benjamin.voisin@ens-rennes.fr

I. Introduction

Le vote électronique permet à la fois de simplifier et accélérer les processus électoraux, tout en proposant des possibilités de votes plus larges (mode de scrutin). Il est cependant primordial de veiller à la sécurité du vote : une attaque sur un vote par internet peut-être plus facilement d'une ampleur suffisante à changer le cours d'une élection.

A. Motivations

En pratique, le vote électronique est déjà utilisé, parfois pour des élections à grands enjeux. Par exemple en 2022 pour les élections des députés français de l'étranger, et où la solution de vote s'est avérée faillible à de multiples attaques[1]. Par exemple, il existe le problème du ballot stuffing, où le serveur de vote, sachant qui a voté, envoi un vote dans l'urne en usurpant l'identité d'un abstentionniste.

Il existe également un risque lors de la distribution des identifiants de vote, qui peuvent être volés ou achetés.

Pour palier à ces problèmes, une proposition est d'utiliser un fournisseur d'identité externe à l'élection, ce qui apporte plusieurs propriétés intéressantes :

- La distribution des identifiants peut être faite de manière très sécurisée et dissociée de l'élection.
- L'achat d'identifiants est compliqué, car ils peuvent être utilisés pour d'autres services que l'élection, ce qui augmente leur valeur et donc leur prix.
- Le serveur ne peut pas usurper l'identité d'un votant puisqu'il devrait s'authentifier avec des codes qu'il ne connait pas.

L'utilisation d'un fournisseur d'identité externe soulève cependant de nouveaux problèmes à résoudre, notamment dans le respect des propriétés de privacy et de vérifiabilité.

B. Approche et contributions

L'objectif est donc d'évaluer la possibilité d'utiliser le standard OpenID Connect pour l'authentification lors d'un processus de vote par internet. Le protocole complet a déjà été établi, mais la faisabilité et complexité d'une partie reste à évaluer. Il s'agit de la conception d'une preuve à divulgation nulle de connaissance d'éligibilité. La génération de cette preuve peut être très coûteuse d'un point de vue calculatoire. Nos contributions se font donc dans trois catégories :

a) Design de preuve : Il faut d'abord designer une preuve à divulgation nulle de connaissance qui satisfait les propriétés voulues, en essayant de la rendre la moins complexe et longue à générer possible.

b) Implémentation : Il faut ensuite implémenter la génération et vérification de cette preuve pour rendre la chose utilisable en pratique et l'évaluer.

c) Évaluation : Il faut finalement évaluer le coût de la solution proposée.

II. Contexte

A. Vote électronique et vérification d'éligibilité

En vote électronique, deux propriétés désirables peuvent souvent entrer en collision : la vérifiabilité, et la privacy. C'est le cas lorsqu'il s'agit d'authentification et de vérification d'éligibilité.

La vérification d'éligibilité est la possibilité donnée à des auditeurs indépendants (entités de contrôle ou votants eux-mêmes), de vérifier que tous les bulletins de vote présents dans l'urne y ont bien été déposés par un votant éligible.

Une solution simple pour obtenir cette propriété serait de publier, à côté du bulletin de vote chiffré, l'identité du votant. Ainsi, une simple comparaison avec les listes électorales permet de s'assurer que tous les bulletins correspondent bien à un votant éligible, et n'importe qui peut se rendre compte d'une usurpation de son identité (par un attaquant, ou par le serveur de vote) en voyant son nom apparaître.

Malheureusement, cette solution ne respecte pas la condition d'everlasting privacy. En effet, il est possible que dans un futur plus ou moins proche, on soit capable de déchiffrer facilement les bulletins de l'urne. On obtiendrait alors une liste exacte de qui a voté pour qui, ce qu'il faut absolument éviter.

Pour concilier ces deux propriétés, un système comme Belenios publie un subid, un identifiant unique à chaque votant, mais dont l'association identité - subid n'est connue que du Registrar, mais le serveur de vote peut l'apprendre au fur et à mesure que les électeurs votent[2]. L'implémentation faite pour les législatives s'expose néanmoins à une attaque dite par ballot stuffing. En effet, le serveur de vote, s'il est considéré comme malveillant, peut rajouter des votes en fin d'élection, sous les identités des

abstentionnistes[1]. Il est impossible pour un auditeur de s'assurer qu'un vote a bien été émis par un votant, et non par le serveur de vote.

Pour se protéger de ce type d'attaque, on peut déléguer l'authentification à un autre service, un identity provider, qui gère le processus d'authentification, ainsi que l'association identité - subid. Ainsi, le serveur de vote malveillant ne peut pas ajouter de vote, car Le vote électronique permet à la fois de simplifier et accélérer les processus électoraux, tout en proposant des possibilités de votes plus larges (mode de scrutin). Il est cependant primordial de veiller à la sécurité du vote : une attaque sur un vote par internet peut-être plus facilement d'une ampleur suffisante à changer le cours d'une élection.il faudrait qu'il s'authentifie auprès de l'identity provider avec des credentials qu'il ne possède (normalement) pas. Cet identity provider serait un service bien établi, de confiance, et relativement dissocié du scrutin.

B. OpenID Connect

OpenID Connect est un protocole d'autorisation standardisé qui permettrait de fournir une authentification externe au serveur de vote. Le votant s'authentifie auprès d'un fournisseur d'identité. Le fournisseur d'identité renvoi alors un token d'authentification, qui prouve cryptographiquement que le votant est authentifié sous une certaine identité. Le votant peut alors présenter ce token comme preuve d'éligibilité.

Ce token d'autorisation est un JSON Web Token (JWT). Il est composé de trois parties :

a) Le header : Il contient les informations utiles pour son utilisation : le type de JSON que c'est (un JWT ici donc), ainsi que l'algorithme de chiffrement utilisé pour la signature.

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

b) La payload : Il contient les informations du votant, ainsi que quelques éléments indispensables tels que le temps d'expiration, l'identité du fournisseur d'identité. Les informations du votant peuvent varier d'un fournisseur d'identité à l'autre, mais elles contiennent au moins un champ sub (pour subject), un identifiant unique par votant.

```
{
  "sub": "1234567890",
  "name": "Benjamin Voisin",
  "admin": true,
  "iat": 1516239022
}
```

c) La signature : La signature est la preuve cryptographique que les informations précédentes ont bien été émises par le fournisseur d'identité. Elle est ce qui empêche n'importe qui de créer un token valide pour se faire passer pour un autre votant. Sa formation dépend de l'algorithme donné dans le header, mais dans le cas d'OpenID Connect, il s'agit systématiquement d'une signature RSA. La signature est donc générée ainsi : $RSA(pk_{\mathsf{idProvider}}, base64(header) +' .' + base64(payload))$. On peut alors vérifier la signature à l'aide de la clé publique du fournisseur d'identité.

On pourrait alors simplement publier ce JWT à côté du bulletin de vote, mais ceci compromettrait la everlasting privacy du vote. On veut donc prouver deux propriétés sur ce token : l'identité du votant est bien dans la liste électorale et elle n'a été utilisée que pour au plus un vote. Pour prouver ces propriétés sans divulguer directement le JWT, on peut faire appel à des preuves à divulgation nulle de connaissance (Zero Knowledge Proof ou ZKP), qui permettent de prouver des propriétés sur des éléments sans en révéler le contenu.

C. Preuve à divulgation nulle de connaissance

Une preuve à divulgation nulle de connaissance (ZKP) est une manière de prouver des propriétés sur un élément, sans le révéler. Elles ont été présentées pour la première fois en 1985[3] pour quelques preuves particulières, sous la forme de dialogues entre un prouveur et un vérifieur. Il a ensuite été montré que, pour n'importe quel problème dans NP, on pouvait prouver la connaissance d'une solution au problème, sans révéler la solution[4]. Le déroulement d'une ZKP peut se décomposer en 3 phases :

1) Un commit, le premier message envoyé par le prouveur
2) Un défi envoyé par le vérifieur, qui doit être résoluble par le prouveur s'il connait réellement une solution au problème
3) Une réponse du prouveur qui satisfait le défi

Étant donné que le prouveur peut obtenir une réponse valide au défi par chance, ce schéma peut être répété autant de fois que nécessaire, jusqu'à ce que le vérifieur soit convaincu, à une certaine probabilité près, que le prouveur dit vrai.

L'interactivité de ces preuves les rend peu utilisables dans notre cas. On aimerait pouvoir simplement générer une preuve, la publier, et qu'elle soit vérifiable par n'importe qui sans interaction avec le prouveur. Heureusement, il existe une méthode pour rendre une ZKP non interactive[5]

De nombreux systèmes de preuve existent pour faciliter la création de preuve. Dans notre cas, nous utiliserons un système basé sur le paradigme des circuits arithmétiques. Un circuit arithmétique est semblable à un circuit binaire, dans lequel chaque fil ne représente pas un nombre binaire, mais un réel (dans l'implémentions, il s'agira uniquement d'entiers naturels codés sur 64 bits).

D. Design initial : Protocole abstrait

Le protocole décrivant les différents échanges, les éléments privés et les éléments publiés dans l'urne de vote sont schématisé dans la Figure 1.

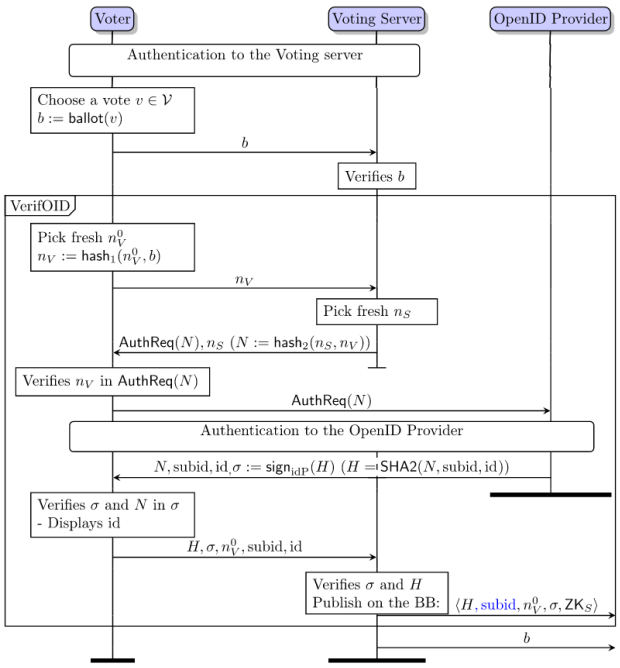

Figure 1. Schéma du protocole d'authentification avec OpenID Connect

Il repose sur trois acteurs : le votant, ou plus formellement l'appareil utilisé pour le vote, le serveur de vote, et le fournisseur d'identité OpenID.

a) Explication du protocole : Après que le votant a choisi puis chiffré un vote, le serveur de vote vérifie que le vote est un vote valide, à l'aide d'une ZKP non discutée ici. On entre ensuite dans la phase d'authentification auprès du fournisseur d'identité, qui s'articule autour de la phase Authentification to the OpenID Provider, lors de laquelle le votant s'authentifie directement auprès du fournisseur d'identité qui lui renvoie alors le JWT d'authentification. Ces informations sont ensuite utilisées pour calculer la preuve ZK_S, qui est publiée dans l'urne de vote.

b) Nonces n_S et n_V : Pour garantir certaines propriétés, des nonces sont ajoutés dans le protocole. Le nonce n_V permet au votant de s'assurer que son authentification ne peut être utilisée que pour son bulletin de vote. Il garde le secret n_V^0 jusqu'à ce qu'il se soit assuré de cette propriété, rendant impossible pour le serveur de vote de générer un token d'authentification correct pour un mauvais bulletin.

Le nonce n_S permet au serveur de vote de faire la distinction entre "j'ai un token d'authentification" et "je sais m'authentifier auprès du fournisseur d'identité", évitant ainsi le rejeu de token d'authentification par un acteur malveillant.

c) Éléments publiés : À la fin du protocole, plusieurs éléments sont rendus publics :

— H, le hashé du JWT
— subid, un identifiant unique pour chaque identité, dont l'association id - subid n'est connue que du fournisseur d'identité.
— n_V^0,
— σ, la signature du JWT, prouvant que le JWT dont le hache est H a bien été émis par le fournisseur d'identité.
— ZK_S, une ZKP prouvant que l'identifiant dans le champ *id* du JWT est bien dans les listes électorales, que l'élément dans le champ subid correspond bien à celui qui est public, que le hashé du JWT est bien égal à H et que le nonce utilisé dans le JWT est bien le résultat du calcul $\mathrm{hash}_2(n_S, n_V)$.

L'objectif de ce stage est donc de construire la preuve ZK_S, de manière sécurisée et suffisamment efficace pour être utilisable lors d'élections.

III. Design

Dû au choix du système de preuve expliqué dans la partie suivante, la structure de la preuve sera celle d'un circuit arithmétique. On peut alors décomposer le circuit de preuves en plusieurs sous-circuit, chacun servant à prouver une propriété.

A. Les « modules » de preuves

1) SHA256 : Bien qu'il existe des fonctions de hash ZK-friendly, l'utilisation d'OpenID Connect nous force à utiliser SHA256 comme fonction de hashage.

Cette contrainte ajoute une très grande complexité au calcul de la preuve, entre autres à cause du besoin de faire les calculs directement à partir de la représentation binaire des entrées, là les circuits qu'on utilise son fait pour gérer des entiers allant jusqu'à 64 bits. Le nombre de contraintes dans le circuit en est donc significativement augmenté.

2) Fonction de hash au choix : Notre preuve ZK_S doit fournir une autre preuve de hash, mais cette fois-ci, rien ne contraint le choix de la fonction en question. On peut donc utiliser une fonction de hash ZK-friendly. Ici, le choix d'implémentation conduit très naturellement à la fonction de hash Poseidon, une fonction conçue pour être utilisé dans des ZKP[6].

Ce choix réduit donc le sous circuit de cette preuve de hash à une simple porte "fonction de hash Poseidon" déjà présente et optimisée dans le système de preuve choisi. Hmm

3) Preuve d'appartenance : Prouver l'appartenance d'un élément à un ensemble peut sembler compliquer de premier abord, surtout si on souhaite éviter d'encoder tout l'ensemble dans la ZKP (ce qui ajouterais autant de contrainte que l'ensemble est grand, et rend donc les preuves très rapidement très lente).

La solution qu'on utilisera sera celle des Merkle Tree, qu'on va détailler.

Un Merkle Tree est une manière de compacter un ensemble de donnée à l'aide de hash successifs. Concrètement, il s'agit d'un arbre binaire, dans lequel la valeur des nœuds est égal au hash des deux nœuds inférieurs, et où les feuilles prennent pour valeurs les éléments de notre ensemble. Cette structure permet de réduire grandement la taille d'une preuve d'appartenance, puisqu'il s'agit alors

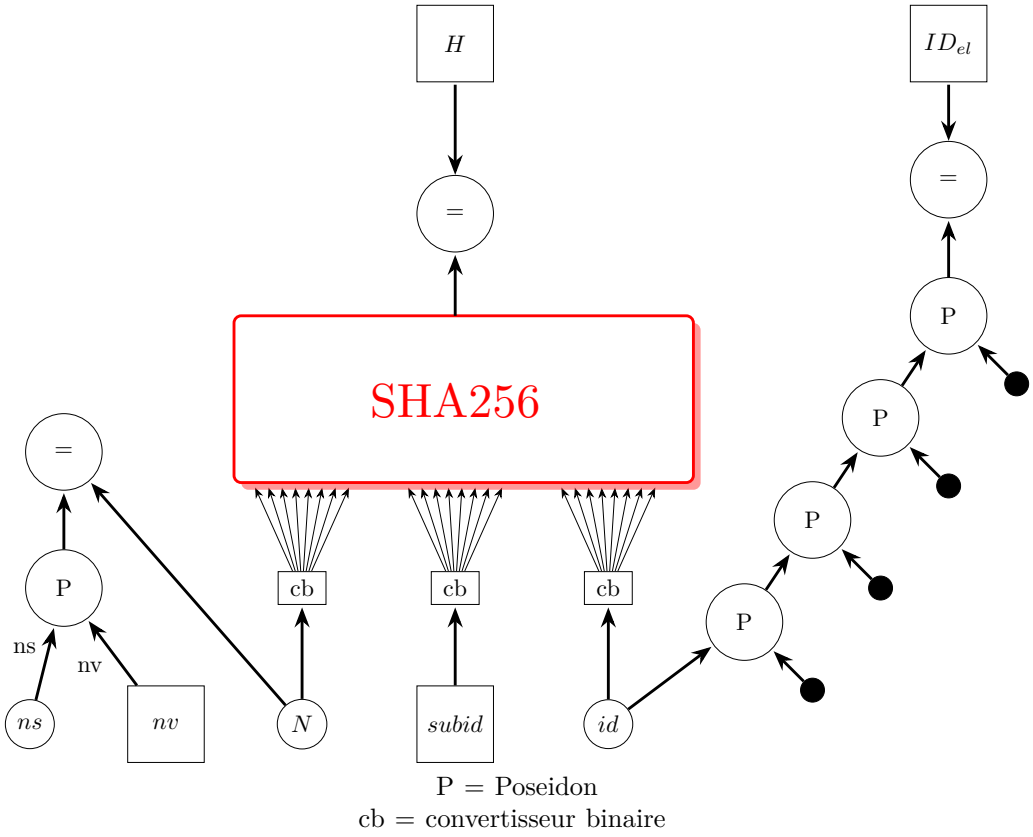

Figure 2. Schéma du circuit arithmétique de la preuve ZK_S

d'une succession de hash. Pour prouver qu'un élément *id* est dans un ensemble IDel public, le prouveur va effectuer un nombre de preuve de hash égal à la hauteur du Merkle Tree ($\log_2$ du nombre d'éléments dans l'ensemble), et montrer que le résultat est bien égal à la racine de l'arbre, connue de tous.

Il ne reste plus qu'à lier l'entrée de cette succession de hash avec le reste du circuit, pour garantir que l'on prouve bien que ce qui est dans l'ensemble est le champ *id* du JWT.

B. Combinaison des modules

Pour rendre la preuve valable, il est indispensable de connecter les modules entre eux de manière à garantir au verifieur qu'ils se font sur les mêmes éléments. Le schéma complet de la preuve peut se présenter comme dans la figure ??.

Cette figure met en évidence les connexion entre les différents modules, avec à gauche le module prouvant que le nonce N est bien formé, à droite le module prouvant que *id* est bien dans l'ensemble ID_{el}, et au centre le bloc de calcul du module SHA256.

1) Conversions binaires : Pour connecter le bloc SHA256 aux autres modules, il est nécessaire de procéder une conversion. En effet, les circuits arithmétiques que nous utilisons portent directement des entiers allant jusqu'à 2^{64}. Ainsi, il y a 4 outputs en sorti d'une porte de hashage Poseidon, le résultat étant 256 bits. Pour pouvoir lier la sortie d'un hash Poseidon avec l'entrée du bloc SHA256, il faut donc incorporer, dans le circuit arithmétique de la preuve, la conversion d'une représentation à l'autre.

La même chose se fait avec l'entrée du module d'appartenance. Bien que la conversion ne soit pas en soit nécessaire, elle permet de réduire le nombre d'input dans la première porte de hash Poseidon, ce qui compense le coût du circuit de conversion.

2) Conversion base64 : Malheureusement, ce n'est pas la seule conversion à effectuer. En effet, OpenID Connect effectue le hashage SHA256 non pas sur le token directement, mais sur la représentation base64 de ce token. Cela pose plusieurs problèmes, dont un dans la liaison entre le module de preuve de correction du nonce N avec le bloc SHA256. En effet, il ne faut pas hasher le résultat de cette fonction de hash, mais la représentation en base64 de ce résultat.

C. Gestion de la conversion base64

Effectuer la conversion base64 dans le circuit arithmétique serait extrêmement coûteux, cela nécessiterait de multiplier des tables de conversions pour chaque groupe. Au lieu de ça, on peut effectuer une modification du nonce avant de l'envoyer à OpenID Connect, de sorte que la conversion base64 qui sera effectué soit plus simple à calculer.

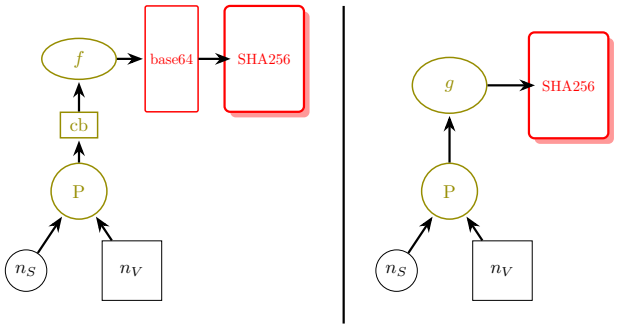

Figure 3. Schéma de la simplification de la conversion base64

Formellement, on introduit une fonction f, et au lieu d'envoyer $\mathsf{Poseidon}(n_S, n_V)$, on envoie $f(\mathsf{Poseidon}(n_S, n_V))$, de sorte que le calcul effectué sera $\mathsf{SHA256}(\mathsf{base64}(f(\mathsf{Poseidon}(n_S, n_V))))$. Le but est d'avoir une fonction $g = \mathsf{base64} \circ f$ simple à implémenter en circuit arithmétique.

La solution trouvée consiste à découper le résultat en demi-mots de 4 bits, puis de rajouter les bits "00" devant pour obtenir un mot de 6 bit, donc l'image en base64 sera dans le sous-ensemble A-P. Ainsi, la suite de bits 0101 1101 0000 1011 deviens 001010 001101 000000 001111. Si on regarde la représentation binaire de la conversion base64 de cette suite de bits, on trouve 01001011 01001110 01000001 0101000.

On obtient donc une fonction g simple : ajouter 0100 devant tous les demi-mots de 4bits, puis incrémenter de 1 les nouveaux mots de 8 bits. Implémenter cette fonction permet alors de simplifier le schéma arithmétique.

IV. Implémentation

L'implémentation est un élément important de ce stage. Les contraintes d'implémentations ont joué un rôle important dans certains choix de design, et l'objectif premier de ce stage était de fournir une preuve de concept montrant ou pas la faisabilité et le potentiel coût d'une telle solution.

A. Choix du système de preuve

Plutôt que de réinventer la roue inefficacement en créant pleins de problèmes de sécurité, on peut utiliser un système de preuve, une librairie que l'on peut utiliser pour construire nos ZKP. Beaucoup d'options différentes existent, et il convient de choisir intelligemment le système à utiliser. Pour ça, il faut s'intéresser à nos différentes contraintes.

a) Contraintes : La plupart des systèmes de preuves sont dévloppés pour des utilisations dans des blockchains, avec pour principal objectif des temps de vérification très court, et parfois des tailles de preuve les plus petites possible. Le temps de génération de preuve est moins important que le temps de vérification. Or, dans notre cas, c'est l'inverse. On souhaite réduire au maximum le temps de génération de preuve, car celles-ci doivent être générées au cours de l'élection. Les vérifications peuvent se faire plus lentement après.

On sait déjà que le principal coût de notre preuve se fera au niveau du calcul de SHA256. La vitesse de preuve de préimage SHA256 sera donc déterminante dans le choix du système de preuve.

On peut lister quelques options envisageables :

— Circom
— gnark
— Arkworks
— Halo2
— Plonky2
— Starky

Ces différents framework ont été comparés dans [7], sur l'épreuve de la préimage SHA256.

Au final, le choix a été fait d'utiliser Plonky2, une librairie Rust avec de très bonnes performances, avec en tête le possible passage à Starky, Plonky2 étant mieux documenté, avec des exemples et quelques guides d'utilisation qui rendent la librairie utilisable. Il est aussi intéressant de savoir qu'un circuit de preuve Plonky2 peut intégrer un morceau de preuve généré avec Starky, ce qui facilite le passage de l'un à l'autre et permettra éventuellement d'accélerer les passages les plus lents de la preuve.

B. Architecture de la libraire

Cette librairie doit être utilisable par une personne sans connaissance particulière sur les ZKP. En particulier, on souhaite abstraire l'utilisation de Plonky2, afin de pouvoir changer de fRAMework de preuve sans changer l'interface avec la librairie.

Pour clarifier les différents éléments dans le code, un soin a été apporté à la séparation entre les éléments purement ZKP, qui manipulent des circuits arithmétiques de Plonky2, et les calculs plus haut niveau fait en code Rust. On a donc par exemple, une fonction prove qui va générer tous ZK_S. Cette fonction fait plusieurs calculs en Rust (par exemple, calculer le chemin de id à la racine dans le Merkle Tree), avant de faire appel à une seconde fonction, zkp::prove, qui manipule génère la ZKP à proprement parler.

C. Tests

Tout au long du développement, des tests unitaires ont été ajoutés au code. En particulier, les "modules" sont testés individuellement, certains éléments de connexions (les convertisseurs binaires, ainsi que le circuit de "conversion base64". Tous ces tests ont permis d'aborder sereinement les problèmes de connexion entre les modules, en sachant que si la preuve échouait, le problème venait d'où et comment les modules étaient connectés. En effet, Lorsque la preuve échoue (par exemple deux contraintes incompatibles, une égalité qui n'est pas vérifiée), Plonky2 panique, et le message d'erreur renvoyé est rarement très instructif. Ces tests sont donc indispensables pour pouvoir identifier quelles contraintes causent des problèmes, et ainsi corriger le code.

D. Benchmarks et évaluation de faisabilité

Pour évaluer la génération de la preuve et voir quelles sont les parties coûteuses, nous avons mesuré le temps de calcul pour chacune des sous parties, ainsi que le temps total nécessaire à la preuve complète.

a) Comparer avec la litérature : [7]

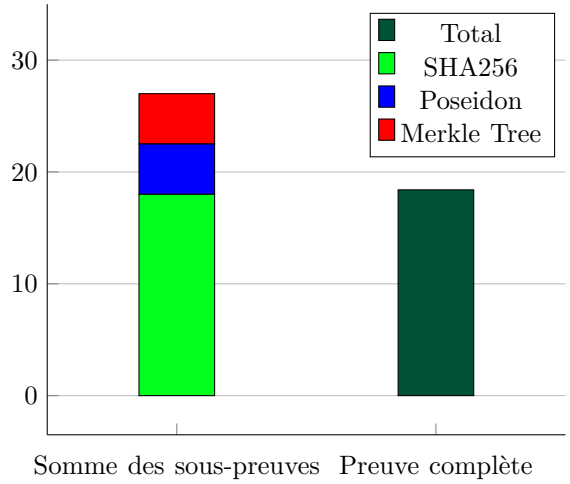

Figure 4. Résultats des benchmarks

On voit donc clairement, comme attendu, que c'est la partie SHA256 qui est la plus coûteuse. De plus, la parallélisation effectuée par Plonky permet d'avoir une preuve totale pas beaucoup plus longue que la partie SHA256 individuellement.

Une autre mesure importante concerne l'utilisation de la RAM lors de la génération de la preuve. Celle-ci peut monter à plus de 4GB utilisé, ce qui exclut la possibilité de faire générer les preuves sur l'appareil du votant.

On peut donc estimer un temps de génération de preuve de l'ordre de 20 secondes sur un serveur un peu costaud. Concrètement, cela se traduit en 5h30 de calcul pour une élection à 1 000 votants, et 55 heures pour 10 000 (2 jours et 7 heures). C'est une option qui peut donc s'envisager, les calculs de preuves pouvant être différé et lissés sur toute la durée de l'élection, mais cela constitue tout de même un surtout très important, surtout si l'on souhaite augmenter l'échelle de l'élection (il faut alors multiplier les machines).

En revanche, si on regarde uniquement le temps de génération de preuve, en omettant la création du circuit, on peut réduire le temps de calcul à 6.5 secondes par preuves, ce qui fait 1h48 pour 1 000 votants et 18h pour 10 000. Pouvoir réutiliser un même circuit pour différentes preuves représente donc un gain considérable.

Ces résultats sont certes encore lents, mais le monde des ZKP avance très vite grâce à l'engouement envers les crypto-monnaies (Plonky2 est à la base développée pour une crypto monnaie), et il est raisonnable de voir arriver dans les années proches de nouveaux systèmes de preuves significativement plus rapides. En plus de ça, nous avons pu identifier plusieurs axes d'amélioration pour accélérer considérablement le temps de preuve.

V. Améliorations possibles

La durée limitée du stage a contraint à se concentrer sur la production d'une version fonctionnelle, avant d'étudier de possibles optimisations. Même si elles n'ont pas pu être implémenté, plusieurs améliorations ont été envisagées et approfondies.

A. Circuit réutilisable

La première d'entre elles est de construire un circuit arithmétique unique pour toutes les preuves, et ainsi économiser le temps de création du circuit. Cependant, cette optimisation soulève beaucoup de problèmes, que nous allons lister, et auxquels nous allons tenter de proposer une solution.

1) Choisir les entrées publiques / privées : Le premier problème est de choisir quels éléments sont publiques et quels éléments sont privés dans le circuit arithmétique. En effet, on souhaite que le vérifieur puisse observer certaines valeurs dans le circuit arithmétique, notamment dans l'entrée du bloc SHA256, c'est-à-dire dans le JWT. Par exemple, il faut pouvoir vérifier que les bits avant la partie privé qui est connecté au Merkle Tree correspondent bien aux caractères "id:". Or, la taille du JWT, et l'index des différents éléments, varie d'un utilisateur à l'autre, et on ne peut pas savoir à l'avance quels bit d'entrée doit être publique et quel bit d'entrée doit être privé.

Une solution serait de rendre paramétrique le statut publique / privé de chaque input. Pour ce faire, on considère tous les inputs du bloc SHA256 privés, et on les relie à une seconde couche identique, mais où tous les éléments sont publiques. Pour choisir entre privé et public, on ajoute entre ces deux couches des portes "et", dont on peut choisir la valeur du second input. Ainsi, lors de la génération de la preuve, le prouveur peut mettre à 0 tous les éléments devant être privée, et laisse inchangée les éléments publique.

Figure 5. Schéma de la layer avec les porte and pour les inputs publique

2) Connecter les modules aux bons endroits : Pour les mêmes raisons que précédemment, on ne peut pas savoir à l'avance où il faut connecter les différents modules. Les indices et tailles des différents claims pouvant changer pour chaque utilisateur. Une solution similaire peut être envisagée, en connectant plusieurs inputs du bloc SHA256 aux entrées des autres blocs, et en conditionnant quelle connexion est utilisée à l'aide de porte logique. Cette solution peut représenter un gros surcout si on connecte tous les inputs de SHA256 avec tous les inputs des autres modules, mais en réduisant au maximum la variance dans les positions et tailles des différents éléments du JWT, on peut arriver à un surcout marginal.

Figure 6. Schéma de la layer de connexion entre les différents indices

3) Taille du bloc SHA256 : Le dernier problème à résoudre se situe dans la taille du bloc SHA256 à générer, étant donné que la taille du JWT est variable. Ici, nous n'avons pas encore trouvé de solution, et ignorons si le problème est soluble de manière à peu près efficace.

B. Rendre possible l'envoi d'un nonce autrement que bit à bit

Pour rendre la librairie plus utilisable, il faut modifier le circuit de conversion bass64, puisqu'il repose aujourd'hui sur l'hypothèse forte de pouvoir envoyer un nonce bit à bit, afin d'éviter une double conversion base64. Il faudrait alors repenser le circuit de conversion base64, mais les différentes solutions imaginées pour l'instant semblent très coûteuses.

C. Starky

Finalement, il peut être intéressant de faire appel à Starky. En particulier, on peut imaginer prouver la préimage SHA256 à l'aide de Starky (il existe déjà des exemples de code de préimage SHA256 à l'aide de Starky), puis insérer cette preuve dans le circuit arithmétique déjà utilisé, optimisant alors significativement la partie la plus coûteuse de la preuve.

Références

[1] A. Debant and L. Hirschi, "Reversing, breaking, and fixing the french legislative election e-voting protocol," Cryptology ePrint Archive, Paper 2022/1653, 2022. [Online]. Available : https://eprint.iacr.org/2023/1653.pdf

[2] V. Cortier, P. Gaudry, and S. Glondu, Belenios : A Simple Private and Verifiable Electronic Voting System. Cham : Springer International Publishing, 2019, pp. 214–238. [Online]. Available : https://inria.hal.science/hal-02066930/document

[3] S. Goldwasser, S. Micali, and C. Rackoff, The Knowledge Complexity of Interactive Proof-Systems, ser. STOC '85. New York, NY, USA : Association for Computing Machinery, 1985, p. 291–304. [Online]. Available : https://doi.org/10.1145/22145.22178

[4] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems," J. ACM, vol. 38, no. 3, p. 690–728, jul 1991. [Online]. Available : https://doi.org/10.1145/116825.116852

[5] A. Fiat and A. Shamir, How to Prove Yourself : Practical Solutions to Identification and Signature Problems. Berlin, Heidelberg : Springer-Verlag, 1987, p. 186–194. [Online]. Available : https://link.springer.com/content/pdf/10.1007/3-540-47721-7_12.pdf

[6] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon : A new hash function for Zero-Knowledge proof systems," in 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, Aug. 2021, pp. 519–535. [Online]. Available : https://www.usenix.org/conference/usenixsecurity21/presentation/grassi

[7] C. Network, "The pantheon of zero knowledge proof development frameworks," 2023. [Online]. Available : https://blog.celer.network/2023/03/01/the-pantheon-of-zero-knowledge-proof-development-frameworks/