

Encadrant :

POPINEAU Fabrice

BOUCHER Julien

MAISON Jonas

|  |
| --- |
| CentraleSupélec Campus de Gif, Année 2016-2017  Rapport de projet de développement logiciel :  Cryptographie et arbres de Merkle |

**GitHub :** <https://github.com/Jonas1312/merkle-tree>

Sommaire

[Sommaire 2](#_Toc483748185)

[Introduction 3](#_Toc483748186)

[Schéma de signature de Lamport 4](#_Toc483748187)

[1. Génération de la clé privée 4](#_Toc483748188)

[2. Génération de la clé publique 4](#_Toc483748189)

[3. Génération de la signature d’un message 4](#_Toc483748190)

[4. Vérification de la signature 5](#_Toc483748191)

[5. Avantages et inconvénients 5](#_Toc483748192)

[Arbre de Merkle 6](#_Toc483748193)

[1. Génération de la clé privée 7](#_Toc483748194)

[2. Génération de la clé publique 7](#_Toc483748195)

[3. Génération de la signature d’un message 7](#_Toc483748196)

[4. Vérification de la signature 8](#_Toc483748197)

[5. Avantages et inconvénients 8](#_Toc483748198)

[Conception du logiciel 9](#_Toc483748199)

[1. Langage choisi pour l’implémentation 9](#_Toc483748200)

[2. Classes utilisées 9](#_Toc483748201)

[Conclusion 11](#_Toc483748202)

[Références 12](#_Toc483748203)

Introduction

La signature numérique est une marque permettant, comme une signature papier, de démontrer qu’un fichier ou document signé est bien original et n’a pas été altéré ou modifié par un tiers lors de l’envoi au destinataire. Elle permet entre autre de vérifier l'identité de l'auteur des informations numériques, tels que des documents, des messages électroniques ou des transactions monétaires à l'aide d'algorithmes de chiffrement.

Une signature numérique doit respecter les conditions suivantes :

* Infalsifiable : elle ne peut pas être falsifiée par un tiers.
* Authentique : elle permet de garantir que le signataire est bien celui qu’il prétend être.
* Inaltérable : une fois que le document est signé on ne peut plus le modifier.
* Irrévocable : la personne qui a signé ne peut le nier.
* Non réutilisable : elle ne doit être utilisée qu’une fois pour authentifier un document.

Contrairement aux idées reçues, la signature numérique ne chiffre ou ne crypte pas le document, il reste visible et lisible à n’importe quel utilisateur présent sur le réseau de communication.

Dans le cadre de notre projet de développement logiciel nous avons étudié un schéma de signature basé sur l’arbre de Merkle (ou arbre de hachage), qui est une structure de données utilisée notamment pour différents protocoles ou logiciels tels que Git, Bittorrent ou le Bitcoin, qui reposent tous sur l’utilisation d’un système de signature à usage unique pour authentifier les échanges de données.

Les arbres de Merkle ont été développés dans les années 70 par Ralph Merkle, et permettent principalement d’économiser le nombre de clés publiques nécessaires pour transmettre des messages. Un autre avantage est qu’il semblerait que le système de signature de Merkle soit résistant aux attaques exploitants des algorithmes quantiques.

Dans un premier temps nous expliquerons dans ce rapport le fonctionnement de la signature de Lamport, qui est souvent utilisée avec les arbres de Merkle que nous étudierons ensuite. Puis nous verrons le démonstrateur qui a été conçu, permettant de mettre en œuvre la signature et la transmission de données sur le réseau, et enfin nous conclurons sur notre projet.

Schéma de signature de Lamport

Le schéma de signature de Lamport, inventé en 1979 par Leslie Lamport, est une méthode permettant de créer des signatures numériques. La sécurité des signatures de Lamport repose sur des fonctions de hachages cryptographiques, aussi appelées « one way function ». Le schéma de signature du Lamport produit ce que l’on appelle des « One-Time Signature », car elles ne peuvent être utilisées qu’une seule fois pour signer un message, elles sont à usage unique.

Une fonction de hachage est une fonction qui est facile à calculer mais qui est très extrêmement difficile, voire impossible, à inverser. Ces fonctions sont en général utilisées pour générer des empreintes uniques de fichier, mais sont aussi fortement utilisées dans le domaine de la cryptographie.

Le schéma de signature de Lamport se déroule en quatre étapes : génération d’une clef privée, génération d’une clef publique, génération d’une signature pour un message donné, et enfin vérification de la signature du message par le destinataire du message.

On suppose ici qu’Alice souhaite envoyer un message ainsi que la signature de celui-ci à Bob. Elle dispose d’un générateur de nombre aléatoires et d’une fonction de hash cryptographique sur 256 bits telle que le SHA256.

### Génération de la clé privée

Pour créer la clef privée, Alice utilise un générateur de nombre aléatoires pour générer 256 paires de nombres aléatoires, chaque nombre étant défini sur 256 bits. C’est la clef privée d’Alice qui doit absolument rester privée. La taille de cette clef est de 2x256x256 bits, c’est-à-dire 16 KiB au total.

|  |  |  |
| --- | --- | --- |
|  | X0 | X1 |
| Paire 0 | 256 bits | 256 bits |
| Paire 1 | 256 bits | 256 bits |
| … | 256 bits | 256 bits |
| Paire 255 | 256 bits | 256 bits |

### Génération de la clé publique

Pour créer la clef publique, il suffit de calculer les hashs de chaque nombre de la clef privée (512 nombres donc). La fonction de hachage générant des nombres de 256 bits, la taille de la clef publique est identique à celle de la clef privée, soit 16 KiB.

### Génération de la signature d’un message

Dans un premier temps Alice utilise la fonction de hachage pour générer le hash du message à signer. Ensuite, pour chaque bit du hash précédemment calculé Alice prend le nombre de la colonne X0 de la clef privée si le bit est 0, sinon elle prend le nombre de la colonne X1 de la clef privée. Cela produit une séquence de 256 nombres de 256 bits, soit 8 KiB, c’est la signature du message.

Maintenant que la clef privée d’Alice a été utilisée, elle peut être détruite car elle ne doit pas être transmise et elle ne doit plus jamais servir sous peine de rendre vulnérable la sécurisation des messages.

Alice transmet à Bob le message, la signature de Lamport et la clef publique utilisée pour générer la signature.

### Vérification de la signature

Bob souhaite vérifier que le message est bien authentique et n’a pas été modifié lors du transfert. Pour cela, il commence par calculer un hash du message et génère les hashs de chacun des 256 nombres de la signature de Lamport. Pour chaque bit du hash du message, si la valeur du bit est 0 on prend le nombre de la colonne X0 (ou X1 si le bit est à 1) de la clef publique et on le compare au hash correspondant dans la signature de Lamport. On répète l’opération 256 fois et si tous les hashs correspondent le message est authentique, sinon il y a eu une modification du message de la signature ou de la clef publique.

### Avantages et inconvénients

La sécurité des signatures de Lamport dépend directement de la fonction de hachage utilisée ainsi que du générateur de nombre aléatoire utilisé pour générer la clef privée. En effet, certaines fonctions de hachage comme le MD5 ne sont pas parfaitement fiables et présentent des collisions (deux fichiers ayant le même hash). La signature de Lamport présente cependant l’avantage d’être facilement adaptable à n’importe quelle fonction de hachage.

L’inconvénient majeur des signatures de Lamport est qu’un couple de clefs publiques et privées ne peut être utilisé qu’une seule fois pour signer un message. De plus, la taille des données à transmettre est assez importante quel que soit la taille du message (16 KiB pour la clef publique et 8KiB pour la signature). Pour pallier à ce problème, on utilise les arbres de Merkle qui permettent d’utiliser une seule clef publique pour signer plusieurs messages.

Arbre de Merkle

Comme expliqué précédemment, le principal problème des « One-Time Signature » et en particulier des signatures de Lamport, est qu’elles possèdent en général des clefs publiques assez volumineuses et qu’il faut obligatoirement en générer une à chaque nouveau message. Une solution à ce problème est d’utiliser un arbre de Merkle, aussi appelé arbre de hachage.

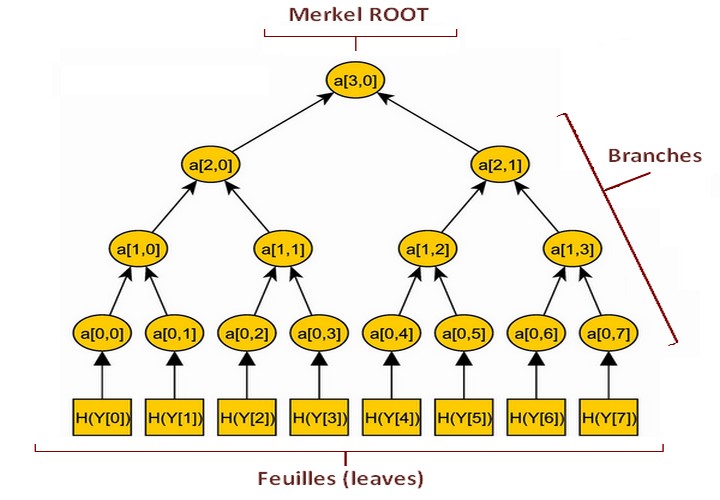


Figure 1 : Arbre de Merkle binaire

Le principe d’un arbre de hachage est de stocker les hashs des données à signer dans les feuilles de l’arbre. On calcule ensuite pour chaque nœud restant le hash résultant de la concaténation des hashs des deux fils du nœud en question. Le hash au sommet de l’arbre est appelé Merkle root.

Ce type d’arbre est très utilisé dans les réseaux peer to peer, où les fichiers sont découpés en blocs de taille fixe, et transmis dans un ordre aléatoire. Pour vérifier l’intégrité d’un fichier on utilise ce protocole :

1. Alice envoie le hash du fichier à Bob.
2. Bob vérifie si le hash correspond bien au Merkle root de l’arbre qu’il a construit à partir du fichier reçu. Si les deux hashs sont identiques le fichier est valide, sinon on passe à l’étape 3.
3. Si le hash du nœud n’est pas valide, Bob demande à Alice de lui transmettre les hashs des deux nœuds fils.
4. Bob répète l’opération 3 jusqu’à qu’il puisse isoler la ou les feuilles qui présentent un défaut, et donc demander à Alice de retransmettre les blocs qui ont été altérés.

Grâce à ce protocole on contrôle bien que le fichier est valide et on économise la bande passante du réseau car on n’envoie que les hashs et les blocs nécessaires.

Dans notre cas, nous utilisons les arbres de hachages de la même manière sauf que désormais les feuilles de l’arbre contiennent les hashs des clefs publiques générées par le schéma de signature de Lamport. On suppose ici qu’on utilise un arbre binaire et qu’Alice souhaite envoyer messages à Bob.

### Génération de la clé privée

Alice commence par générer couples de clefs privées/publiques de Lamport. C’est la clef privée du schéma de signature de Merkle. La taille de la clef est de KiB.

### Génération de la clé publique

Alice calcule ensuite pour chacune des N clefs publiques générées les hashs grâce à une fonction de hachage choisie préalablement. Ces hashs sont insérés dans les feuilles de l’arbre de Merkle qui est donc composé de feuilles, nœuds et niveaux (en comptant le Merkle root). On désigne un nœud par son niveau et sa position dans le niveau .

On calcule ensuite chaque nœud des niveaux supérieurs en calculant le hash de la concaténation des hashs des fils du nœud. Par exemple : . Le nœud du niveau le plus haut est le Merkle root et il représente la clef publique du schéma de signature de Merkle. La taille de la clef est donc de 256 bits si on utilise le SHA256 comme fonction de hachage.

Alice transmet la clef publique à Bob.

### Génération de la signature d’un message

Pour générer la signature d’un message Alice choisi une paire de clef publique/privée de Lamport qui n’a pas encore été utilisée et génère la signature de Lamport associée au message. Dans la figure ci-dessous la paire choisie est la 3.

Alice cherche ensuite le chemin d’identification (authentication path), qui est composé des nœuds nécessaires pour calculer le Merkle root sachant le nœud de départ.

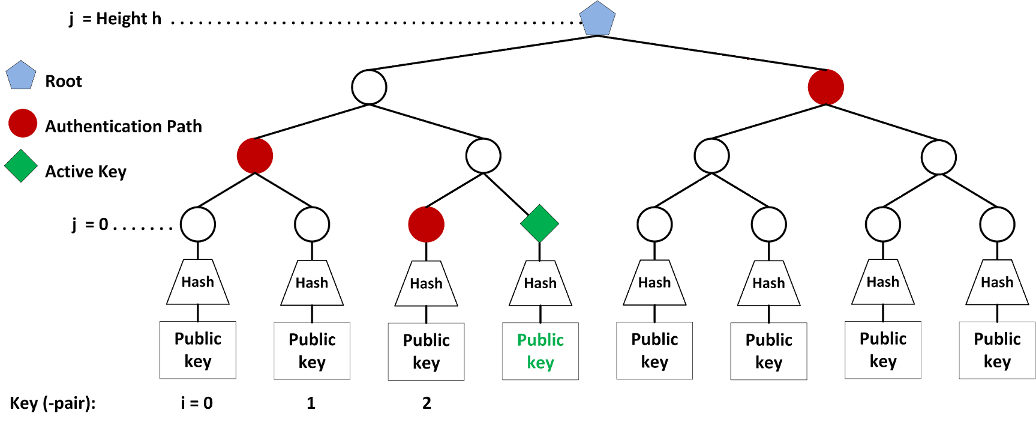


Figure 2 : Authentication path

Alice transmet à Bob le numéro de la paire de clef de Lamport, le message et la signature de Merkle composée de :

* La signature de Lamport (8 KiB)
* La clef publique de Lamport utilisée pour générer la signature (16 KiB)
* L’authentication path (256 bits \* 3 pour l’arbre de la figure 2)

La taille de la signature de Merkle est donc d’environ 24 KiB.

### Vérification de la signature

Bob souhaite savoir si le message est bien authentique et n’a pas été modifié lors du transfert. Il commence par vérifier que la signature de Lamport est correcte. Si celle-ci est correcte Bob reconstruit l’arbre grâce à l’authentication path et compare le Merkle root obtenu à la clef publique qu’Alice lui avait envoyée auparavant. Si les deux sont égaux la signature est bien valide.

### Avantages et inconvénients

L’arbre de Merkle permet donc de combiner plusieurs clefs publiques de Lamport en une seule clef publique à usage multiple et de taille beaucoup plus faible qu’une clef publique de Lamport. Il est relativement simple à implémenter car il ne nécessite que des structures de données basiques et des fonctions de hachage déjà implémentée dans la plupart des langages. Il faut cependant utiliser un générateur de nombres aléatoires suffisamment fiable pour assurer la fiabilité du système.

L’utilisation des arbres de hachage implique cependant beaucoup de calculs numériques pour générer les hashs ainsi que qu’une taille de signature plus importante.

|  |  |  |  |
| --- | --- | --- | --- |
| **Schéma de signature** | **Taille clef publique** | **Taille clef privée** | **Taille signature** |
| Merkle (N feuilles) | 256 bits | 2\*N\*16 KiB | 8 KiB + 16 KiB + log2(N)\*256 bits |
| Lamport | 16 KiB | 16 KiB | 8 KiB |

Conception du logiciel

### Langage choisi pour l’implémentation

Nous avons décidé d’implémenter l’algorithme dans le langage Python pour deux raisons : notre connaissance préalable du langage, ainsi que la simplicité d’implémentation. De plus, cette algorithme étant relativement peu gourmand au niveau des ressources nécessaires, la puissance de calcul de Python est largement suffisante.

### Classes utilisées

Pour implémenter l’algorithme de signature basé sur les arbres de Merkle, nous avons dû créer les classes MerkleTree, LamportSignature, ainsi que les classes Client et Server pour la partie simulation de réseau.

* MerkleTree :

Cette classe possède les attributs *tree*, *n\_levels* et *n\_leaves*. Le premier représente l’arbre sous forme de dictionnaire de tuples (*position*, *data*), position étant lui-même un tuple (*level*, *index*). On compte dans l’arbre de bas en haut et de gauche à droite, la racine est donc à la position (*level\_max*, *0*). *n\_levels* représente la hauteur de l’arbre ( on compte à partir de 0, une racine simple est donc de hauteur 0). Enfin, *n\_leaves* représente le nombre de feuilles de l’arbre.

Quant aux méthodes de MerkleTree :

* **add\_node**(*data, position, hashed*) permet d’ajouter un nœud avec la valeur *data* en position *position* de l’arbre.

Test unitaire : Vérifie que la fonction convertit bien *data* si besoin, et qu’elle l’insère à la bonne place dans l’arbre

* **generate\_tree**() permet de générer l’arbre entier à partir des feuilles

Test unitaire : Vérifie la bonne génération d’un arbre donné

* **get\_root**() nous donne la racine de l’arbre

Test unitaire : vérifie la racine d’un arbre donné

* **get\_brother\_node\_hash**(*position*) renvoie le hash du nœud voisin (le deuxième fils du nœud père)

Test unitaire : Vérifie le bon fonctionnement sur un exemple donné

* **get\_brother\_node\_position**(*position*) renvoie la position du nœud voisin

Test unitaire : Vérifie le fonctionnement peu importe *position*

* **get\_authentification\_path**(*index*) renvoie le chemin d’authentification en partant de la feuille *index*

Test unitaire : Vérifie le bon fonctionnement sur un exemple particulier

* **get\_authentification\_path\_hashes**(*index*) renvoie les hashs du chemin d’authentification en partant de la feuille *index*
* **hash**(*data*) renvoie le hash de *data*
* LamportSignature :

Cette classe possède les attributs *private\_key, public\_key* et *used*. Les deux premiers contiennent les clefs privées et publiques, used est un booléen indiquant si la signature a déjà été utilisée. Les deux clefs contiennent 256 couples de 32 octets, c’est-à-dire 256 bits : chaque clef contient au final 2x256x256 bits ou encore 16 KiB. À la création d’un objet LamportSignature, on génère une clef privée aléatoire, et à partir de celle-ci la clef publique correspondante.

Pour les méthodes :

* **generate\_private\_key**() génère une clef aléatoire

Test unitaire : Vérifie la longueur et le type de l’élément renvoyé

* **generate\_public\_key**() génère la clef publique correspondante

Test unitaire : Idem

* **concatenate\_key**(*key*) concatène tous les couples de 32 octets, puis tous les groupes de 64 octets obtenus, pour finalement ne garder qu’une liste contenant tous les bits

Test unitaire : Vérifie le bon fonctionnement sur un cas particulier

* **decatenate\_key**(*key*) effectue l’opération inverse de la méthode précédente

Test unitaire : Vérifie que concaténer puis décaténer redonne bien la clef d’origine

* **get\_key**(*key\_type, concatenate*) renvoie la clef *key\_type* (publique ou privée), concaténée ou non selon la valeur de *concatenate*.

Test unitaire : Vérifie le bon fonctionnement des différents cas de figure

* **sign**(*msg*) renvoie la signature d’un message donné en clair.
* **verify**(*msg, signature, public\_key*) vérifie la *signature* grâce à *msg* et *public\_key*.

Test unitaire : Vérifie le bon fonctionnement que la signature soit vraie ou fausse

* Client :

Cette classe possède l’attribut *socket*. L’instanciation d’un objet client se fait avec un *host* et un *port* donné (par défaut : *localhost* et *12800* respectivement). Les différentes méthodes de cette classe sont les suivantes :

* **send**(*data*) : envoie *data*
* **close**() : ferme le socket
* Server :

Cette classe possède un attribut *socket*. L’instanciation se fait avec un *port* donné (défaut : *12800*). La classe possède deux méthodes :

* **receive**() : écoute le socket et reçoit les données
* **close**() : ferme le socket

Conclusion

La structure des arbres de hachage est actuellement utilisée dans beaucoup de système tels que le peer to peer, git et la blockchain. Cependant, à cause de temps de calculs longs et de tailles de signatures importantes le schéma de Merkle est finalement assez peu utilisé pour signer des messages basiques, d’autant plus que d’autres algorithmes comme le RSA existent déjà.

L’intérêt croissant porté pour le schéma de signature de Merkle vient du fait qu’il est supposé être résistant aux attaques par des algorithmes quantiques. En effet, la plupart des systèmes utilisés actuellement pour signer ou chiffrer des messages sont supposés être vulnérables aux attaques par l’algorithme quantique de Grover, qui serait apparemment capable de les casser sans effort. Il faut cependant relativiser car tous ces algorithmes ne seront obsolètes que lorsque les ordinateurs quantiques seront démocratisés ce qui est encore loin d’être le cas.

Références

[1] RALPH C. MERKLE. A Digital Signature Based On A Conventional Encryption Function

[2] LESLIE LAMPORT. Constructing Digital Signatures from a One Way Function

[3] Georg Becker. Merkle Signature Schemes, Merkle Trees and Their Cryptanalysis