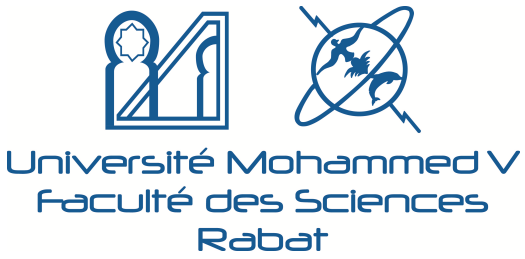
UNIVERSITÉ MOHAMMED V DE RABAT
Faculté des Sciences



Département d'Informatique

Master en Informatique Appliquée Offshoring

Les fonctions de hachage

Présenté par :
El fid Nouhaila
LEFORT Nomenjanahary Nuno

M. Fouad Yakoubi : Professeur à la Faculté des Sciences - Rabat

Année Universitaire 2023-2024

Remerciements

Ce travail résulte d'une conjugaison d'efforts. C'est ainsi que nous ne saurons le clore sans adresser nos remerciements les plus vifs à tous ceux qui ont contribué d'une manière ou d'une autre à sa réalisation.
Nos remerciements iront tout d'abord à l'endroit de Mr. Fouad Yakoubi, notre professseur, pour l'orientation, la documentation, la confiance, et la patience qui a constitué un apport considérable sans lequel ce travail n'aurait pu être au bon port. Qu'il trouve dans ce travail un hommage vivant à sa haute personnalité.

Ensuite, nous profitons de l'occasion pour exprimer notre profonde recon- naissance et gratitude envers l'ensemble du corps professoral et administratif

de la faculté des sciences de Rabat qui, sans leurs efforts notre formation de qualité ne saurait avoir lieu.

Que toutes ces personnes trouvent ici l'expression de nos sincères remerciements.

Résumé

Les fonctions de hachage cryptographiques sont au cœur de la sécurité des systèmes modernes, garantissant l'intégrité des données et l'authentification. Toutefois, malgré leur importance, elles ne suffisent pas à elles seules pour assurer une sécurité cryptographique complète. Ce rapport examine en profondeur les vulnérabilités et les limites des fonctions de hachage, telles que les attaques par collision et de préimage, ainsi que les attaques par canaux auxiliaires. En analysant les faiblesses structurelles et pratiques de ces fonctions, nous mettons en lumière la nécessité de les intégrer avec d'autres mécanismes cryptographiques pour renforcer la sécurité. Une attention particulière est accordée à l'utilisation des fonctions de hachage dans la technologie blockchain, en explorant comment Bitcoin utilise SHA-256 et les arbres de Merkle pour assurer des transactions sécurisées et immuables. Ce rapport vise à fournir une compréhension approfondie des défis et des opportunités associés aux fonctions de hachage dans le cadre de la cryptographie moderne.

Mots clés :Services Cryptographie, fonctions de hachage, sécurité des données, blockchain, algorithmes cryptographiques, monnaie numérique, Bitcoin, SHA-256, arbres de Merkle, cybersécurité, attaques cryptographiques, technologie décentralisée, intégrité des données.

Abstract

This report explores the sufficiency of cryptographic hash functions in ensuring complete cryptographic security. While hash functions are essential for data integrity and authentication, their limitations make them insufficient as standalone security measures. This study delves into various vulnerabilities, including collision, preimage, and side-channel attacks, and examines their implications. Furthermore, the report investigates how hash functions can be effectively utilized in combination with other cryptographic mechanisms, particularly within blockchain technology, highlighting the interplay between Bitcoin, SHA-256, and Merkle Trees. By understanding these interactions, we can develop more robust and secure cryptographic systems.

Keywords : Cryptographic hash functions, security vulnerabilities, collision attacks, preimage attacks, side-channel attacks, blockchain technology, Bitcoin, SHA-256, Merkle Trees, data integrity, cryptographic mechanisms.

Table des matières

Chapitre 1

Introduction Générale

1.1 Introduction à la cryptologie

1.1.1 Un peu d'histoire

La cryptologie est un domaine de recherche scientifique qui tire ses origines des techniques mises en œuvre pour protéger la confidentialité des communications militaires. Pour rendre un message inintelligible pour l'ennemi, son émetteur lui applique une transformation appelée chiffrement. Lors de la réception du message, le destinataire applique l'opération inverse, appelée déchiffrement. On parle de décryptement lorsqu'un attaquant parvient à retrouver le message clair (c'est-à-dire le message avant l'opération de chiffrement) à partir du chiffré. De telles techniques ont vu le jour dès l'antiquité, comme par exemple la scytale utilisée par les Spartiates dès le cinquième siècle avant Jésus-Christ, ou le chiffrement de César.
D'abord très rudimentaires, les mécanismes cryptologiques se sont progressivement complexifiés et répandus dans le domaine militaire. De ce fait, le caractère secret des mécanismes employés est devenu de plus en plus difficile à garantir.
En 1883, Auguste Kerckhoffs énonce un principe fondateur de la cryptologie moderne : les mécanismes de chiffrement et de déchiffrement doivent pouvoir être rendus publics, la confidentialité des messages doit être garantie uniquement par le secret d'une clé. Lors des deux guerres mondiales, la cryptographie a été beaucoup utilisée et la capacité à décrypter les communications ennemies a joué un grand rôle dans l'issue de ces conflits. Dans la deuxième moitié du vingtième siècle, avec l'essor de l'informatique et des télécommunications, la cryptologie a trouvé des applications dans le domaine civil et est devenue un domaine de recherche académique à part entière. Elle permet de protéger des transactions bancaires, des communications téléphoniques, de stocker des données de manière sécurisée, ou encore d'authentifier les utilisateurs d'un système informatique. Si la signification étymologique de cryptologie est science du secret, les problématiques traitées par ce domaine vont aujourd'hui bien au-delà de la confidentialité des communications.
Dans le domaine de la cryptologie, on distingue la cryptographie, qui correspond à la conception de mécanismes cryptologiques, de la cryptanalyse, qui correspond à l'analyse de leur sécurité. Cette branche contient en particulier les attaques contre les mécanismes cryptographiques.

Une autre distinction s'opère entre cryptographie symétrique et cryptographie asymétrique, que nous allons maintenant expliciter.

1.1.2 La cryptographie symétrique

La cryptographie symétrique, aussi appelée cryptographie à clé secrète, regroupe des mécanismes reposant sur la connaissance d'une clé secrète par deux personnes ou entités. Il s'agit de la branche la plus ancienne de la cryptographie.
En effet, les systèmes cryptographiques historiques mettent en œuvre une même clé pour les opérations de chiffrement et de déchiffrement. La cryptographie à clé secrète remplit différentes fonctionnalités, dont les suivantes :

- **Chiffrement symétrique :** Le chiffrement symétrique permet de protéger la confidentialité d'une donnée en la chiffrant avec une clé K. Un mécanisme de déchiffrement utilisant la même clé K permet à quiconque la possède de retrouver la donnée à partir du chiffré.
- **Intégrité d'une donnée :** Pour certaines applications, l'intégrité d'une donnée, c'est-à-dire le fait qu'elle n'ait pas été modifiée après sa création, est au moins aussi importante que sa confidentialité. Par exemple, lors d'une transaction bancaire, il est indispensable que la somme d'argent mise en jeu ne soit pas modifiée. Certains mécanismes de la cryptographie à clé secrète permettent de garantir cette propriété. Ces mécanismes consistent à adjoindre à la donnée M à protéger un code d'authentification de message ou MAC (pour Message Authentication Code), qui dépend de M et d'une clé secrète K. La connaissance de M et K permet la vérification du motif d'intégrité. La validité d'un MAC garantit à la fois l'intégrité du message (il n'a pas été modifié après le calcul du MAC) et une forme d'authenticité appelée authenticité de message (le MAC a été calculé par un détenteur de K).
- **Authentification d'une entité :** Certains mécanismes permettant à un utilisateur d'un système d'information de s'authentifier reposent sur la connaissance commune d'une clé secrète K et sur l'emploi de primitives de la cryptographie symétrique.

Sécurité : En cryptographie, la sécurité des mécanismes utilisés n'est pas inconditionnelle, à de très rares exception près. Par exemple, en cryptographie à clé secrète, la taille de la clé étant fixe, il est toujours théoriquement envisageable pour un attaquant de tester toutes les valeurs possibles de la clé, jusqu'à identifier la bonne. Cette attaque est appelée recherche exhaustive. Les tailles de clés doivent donc être choisies de manière à éviter que cette attaque puisse être réalisée dans la pratique, c'est-à-dire en un temps de calcul réaliste. Selon les primitives utilisées, d'autres attaques peuvent s'avérer plus performantes. La confiance dans les primitives de la cryptographie à clé secrète repose sur leur résistance aux tentatives de cryptanalyse à travers le temps.

1.1.3 La cryptographie asymétrique

La cryptographie asymétrique repose sur une idée exposée par Diffie et Hellman en 1976 dans [DH76] : le chiffrement et le déchiffrement sont deux opérations fonctionnellement différentes. Il n'y a donc aucune nécessité pour que les clés servant au chiffrement et au déchiffrement soient les mêmes. Dès lors que l'on utilise deux clés différentes pour le chiffrement et pour le déchiffrement et que la clé de déchiffrement ne peut pas être déduite de la clé de chiffrement, cette dernière peut être publique.
De ce fait, la cryptographie asymétrique est également appelée cryptographie à clé publique. La clé de déchiffrement (ou clé privée) n'est connue que du destinataire. Ce dernier est l'unique détenteur de la clé privée. Différentes fonctions de sécurité peuvent être obtenues en utilisant des outils asymétriques.

— **Échange de clés :** L'application découverte par Diffie et Hellman dans leur article était un mécanisme d'échange de clé secrète. En effet, un des problèmes posés par la cryptographie symétrique réside dans l'établissement de clés partagées entre deux entités. Le protocole d'échange de clés de Diffie et Hellman est encore le plus utilisé aujourd'hui.
— **Chiffrement asymétrique et chiffrement hybride :** La découverte d'outils mathématiques permettant d'instancier l'idée de Diffie et Hellman est arrivée un peu plus tard. En 1978, Rivest, Shamir et Adleman ont inventé l'algorithme RSA [RSA78]. D'autres algorithmes ont suivi, comme celui d'ElGamal [Gam84]. Pour des raisons de débit, le chiffrement à clé publique n'est généralement pas utilisé pour chiffrer directement des données. On utilise plutôt de telles primitives pour chiffrer des clés symétriques tirées aléatoirement. Ces clés servent ensuite au chiffrement de données à l'aide d'un algorithme de chiffrement symétrique. On parle alors de chiffrement hybride.
— **Signature électronique :** Comme la signature manuscrite, la signature électronique permet de lier un document à un signataire. Dans les deux cas, les signatures peuvent être vérifiées publiquement. Les schémas de signature électronique sont donc fondamentalement asymétriques : la clé privée du signataire intervient dans l'algorithme de signature, et la clé publique correspondante sert à la vérification. La primitive RSA peut également être utilisée pour des applications de signatures, notamment en utilisant le format RSA-PSS [RSA02]. D'autres normes telles que DSA [NIS09] ou sa variante utilisant des courbes elliptiques ECDSA, décrite dans le même document, sont aussi fréquemment rencontrées.

Sécurité : Les mécanismes de la cryptographie à clé publique utilisent des outils mathématiques utilisant des fonctions fondamentalement asymétriques. Ces fonctions sont faciles à calculer et considérées comme difficiles à inverser. Leur sécurité est liée à celle de problèmes mathématiques conjecturés difficiles. Citons par exemple la factorisation de grands entiers pour RSA, ou le calcul de logarithmes discrets dans des groupes multiplicatifs pour DSA, ElGamal ou encore l'échange de clés Diffie-Hellman. La résolution de tels problèmes permet de retrouver la clé privée à partir de la clé publique. Les meilleures algorithmes permettant de résoudre ces problèmes sont plus efficaces qu'une recherche

exhaustive sur la clé privée. A tailles de clés égales, la sécurité potentiellement offerte par les algorithmes symétriques est donc meilleure que la sécurité des algorithmes asymétriques. Réciproquement, pour un niveau de sécurité équivalent, les clés asymétriques sont donc plus longues que les clés symétriques.

1.2 Les fonctions de hachage

1.2.1 Origine

Les fonctions de hachage trouvent leur origine dans les premiers développements de la cryptographie et de la sécurité informatique, visant à convertir efficacement des données en empreintes numériques uniques et irréversibles. Elles ont émergé en réponse aux besoins croissants de sécurisation des données et d'assurance de leur intégrité, devenant ainsi des outils essentiels pour garantir l'authenticité et la vérification des informations dans les systèmes informatiques modernes.

1.2.2 Définition

Une fonction de hachage est un algorithme qui prend en entrée des données de taille variable et les transforme en une empreinte numérique de taille fixe, appelée hash ou haché. Cette empreinte est unique pour chaque ensemble de données, ce qui signifie que toute modification apportée aux données entraînera un changement significatif dans la valeur de l'haché. De plus, les fonctions de hachage sont conçues de manière à rendre très improbable la génération de deux ensembles de données différents qui produisent le même haché (ce qu'on appelle une collision).

1.3 Les fonctions de hachage cryptographiques

1.3.1 Définition

Les fonctions de hachage cryptographiques sont des algorithmes qui transforment des données de taille variable en une empreinte numérique de taille fixe, appelée haché. Elles sont conçues pour être rapides à calculer dans un sens (de données vers haché) et difficile voire impossible à inverser. Leur principale fonction est d'assurer l'intégrité des données en garantissant que toute modification des données entraîne un changement significatif dans le haché produit.

1.3.2 Caractéristiques

Les fonctions de hachage cryptographiques sont des outils essentiels pour garantir l'intégrité et l'authenticité des données dans de nombreux systèmes informatiques. Elles transforment des données de taille variable en une empreinte numérique de taille fixe, offrant ainsi un moyen efficace de vérifier les données sans avoir à les stocker ou à les transmettre intégralement. Pour être considérées comme sécurisées et fiables, ces fonctions doivent satisfaire plusieurs caractéristiques clés :

Détermination : Pour une même entrée, la fonction de hachage produit toujours le même haché.

Efficacité (Compilation rapide) : Les fonctions de hachage doivent être rapides à calculer pour toute entrée donnée, garantissant ainsi leur utilisation efficace dans diverses applications.

Pré-image résistante (Non inversible) : Il doit être pratiquement impossible de retrouver l'entrée originale à partir du haché. Cette propriété garantit que même si un attaquant obtient le haché, il ne peut pas déterminer les données d'origine.

Résistance aux collisions (Unique, 0 collision) : Il doit être extrêmement difficile de trouver deux entrées distinctes qui produisent le même haché. Cela empêche que deux ensembles de données différents soient traités de manière identique, assurant ainsi la sécurité des systèmes.

Diffusion (Non predictable) : Une petite modification des données d'entrée doit entraîner un changement significatif et imprévisible dans le haché produit. Cette propriété, souvent appelée effet avalanche, assure que même des changements mineurs dans les données entraînent des hachés très différents.

Uniformité : Les hachés produits par une fonction de hachage cryptographique doivent être distribués de manière uniforme sur l'espace de sortie, ce qui garantit que chaque valeur de haché possible est aussi probable qu'une autre.

Résistance à la seconde pré-image : Il doit être pratiquement impossible de trouver une seconde entrée différente de la première qui produit le même haché, renforçant ainsi la sécurité des données.

Taille fixe : La fonction de hachage produit une empreinte numérique de taille fixe, quelle que soit la taille de l'entrée. Cela est crucial pour la comparaison et le stockage des hachés.

Ces caractéristiques sont essentielles pour assurer la sécurité et l'intégrité des données dans les systèmes cryptographiques. Elles garantissent que les fonctions de hachage sont fiables, sécurisées, et adaptées à une variété d'applications.

Chapitre 2

Les fonctions de hachage

2.1 MD5

MD5 (Message-Digest Algorithm 5) est une fonction de hachage cryptographique largement utilisée qui produit une empreinte de 128 bits (16 octets) à partir de données de n'importe quelle longueur. Bien qu'elle ait été populaire pour vérifier l'intégrité des fichiers et pour stocker les mots de passe, MD5 est désormais considérée comme peu sûre en raison de sa vulnérabilité aux attaques par collision et aux attaques de préimage. Des méthodes plus modernes, telles que SHA-256, sont recommandées pour les applications nécessitant une sécurité robuste.

2.2 SHA

SHA (Secure Hash Algorithm) est une famille de fonctions de hachage cryptographiques conçue par la NSA et publiée par le NIST. Les versions les plus utilisées incluent SHA-1, SHA-2 et SHA-3, chacune offrant des niveaux de sécurité et de performance adaptés à différentes applications.

SHA-1 : Produit une empreinte de 160 bits. Bien qu'autrefois populaire, il est désormais considéré comme vulnérable aux attaques par collision et n'est plus recommandé pour les applications de sécurité.

SHA-2 : Inclut plusieurs variantes (SHA-224, SHA-256, SHA-384, SHA-512) produisant des empreintes de différentes tailles. SHA-256 et SHA-512 sont les plus couramment utilisés pour leur robustesse et leur sécurité.

SHA-3 : La version la plus récente, basée sur le Keccak, offrant une alternative sûre aux algorithmes de la famille SHA-2. Elle est utilisée pour des applications nécessitant des niveaux de sécurité extrêmement élevés.

Les algorithmes SHA Produisent des empreintes de taille fixe (160 bits pour SHA-1, 256 bits pour SHA-256, etc.). Ils sont largement utilisés pour garantir l'intégrité et l'authenticité des données dans diverses applications, telles que les certificats numériques, les signatures électroniques et la vérification des fichiers.

2.3 BYCRYPT

bcrypt est une fonction de hachage de mots de passe conçue pour être lente afin de rendre les attaques par force brute plus difficiles. Elle utilise l'algorithme de chiffrement Blowfish et inclut un facteur de coût qui peut être ajusté pour augmenter le temps de calcul du hachage, rendant ainsi les attaques par dictionnaire et par force brute plus coûteuses en termes de temps. bcrypt produit un haché de 60 caractères et est couramment utilisé pour sécuriser les mots de passe dans les applications web et les systèmes de gestion des utilisateurs.

2.4 ARGON2

Argon2 est une fonction de hachage de mots de passe reconnue pour sa sécurité et son efficacité, ayant remporté la Password Hashing Competition (PHC) en 2015. Il existe en trois variantes : Argon2d, Argon2i, et Argon2id.

Argon2d : Optimisé contre les attaques basées sur la mémoire, résistant aux attaques par GPU.
Argon2i : Utilise un accès mémoire indépendant des entrées, résistant aux attaques par canaux auxiliaires.
Argon2id : Combine les avantages des deux, offrant une sécurité équilibrée.

Caractéristiques Clés d'Argon2 :
Facteur de Coût Ajustable : Augmente la difficulté des attaques par force brute.
Résistance aux Attaques par Force Brute : Rend les attaques plus coûteuses en termes de temps et de ressources.
Parallélisme : Améliore la vitesse de calcul sur les systèmes multi-cœurs.
Sécurité : Protège contre les attaques par canaux auxiliaires et par dictionnaire.
Sortie : Produit un haché de taille fixe, souvent 256 bits.

Argon2 est largement adopté pour le hachage sécurisé des mots de passe, offrant une protection robuste et une flexibilité adaptée aux besoins des applications modernes.

Chapitre 3

Applications des fonctions de hachage

Les fonctions de hachage sont largement utilisées dans de nombreux domaines de l'informatique et de la sécurité pour diverses applications. Voici quelques utilisations courantes des fonctions de hachage :

3.1 Intégrité des données

La vérification de l'intégrité d'une donnée est parmi les principales utilisations d'une fonction de hachage. Un utilisateur doit être capable de vérifier si une donnée n'a pas été modifiée depuis sa création ou pendant sa transmission à travers un canal de communication. Beaucoup de sites de téléchargement de logiciels affichent sur leur page principale les empreintes des fichiers proposés au téléchargement, calculées au moyen d'une fonction de hachage. Il suffit pour l'utilisateur de télécharger un fichier, de calculer son haché et de comparer la valeur calculée à celle affichée sur le site. Si la fonction de hachage utilisée est connue pour être résistante aux secondes préimages, alors l'utilisateur peut être sûr avec une grande probabilité qu'il détient le bon fichier.

3.2 Protection des mots de passe

Dans beaucoup de systèmes informatiques, l'authentification d'un utilisateur se fait à partir d'un mot de passe. Les mots de passe de tous les utilisateurs doivent être stockés dans le système et à chaque requête une vérification est faite pour permettre l'accès aux utilisateurs légitimes. Toutefois, le stockage des mots de passe en clair peut engendrer de sérieux problèmes. Un attaquant qui arrive à récupérer le contrôle du système sera ensuite capable de s'authentifier sous l'identité de n'importe quel utilisateur. Pour éviter une telle situation, l'haché de chaque mot de passe est calculé et conservé dans le système au lieu du mot de passe lui-même. De cette façon, pour chaque tentative d'authentification, le mot de passe que l'utilisateur insère est haché et le résultat est comparé avec la valeur sauvegardée dans le système. Si la comparaison est réussie alors l'utilisateur a le droit de profiter des services du système, autrement l'accès est interdit. Même si un attaquant arrive à gagner le contrôle du système, il arrivera seulement à récupérer les hachés des mots de passe. Pour retrouver les valeurs cachées derrière les empreintes, il devra être capable de retrouver des préimages de ces valeurs. Pour cela, une fonction de hachage résistante aux préimages doit

être utilisée. Il n'est pas rare qu'une grande partie des utilisateurs choisissent comme mots de passe des mots courants, ne contenant donc pas suffisamment d'entropie. Pour éviter alors les attaques de type dictionnaire, où tous les mots du dictionnaire sont testés les uns après les autres, dans plusieurs systèmes la fonction de hachage est itérée plusieurs fois sur le mot de passe. Ce nombre est par exemple fixé à 1000 pour beaucoup d'applications. Des instructions et des recommandations sur la manière de protéger des mots de passe pour les applications cryptographiques peuvent par exemple être trouvées dans PKCS 5 (RFC 2898).

3.3 Signatures électroniques

Une des applications les plus importantes des fonctions de hachage est leur utilisation dans les signatures électroniques. La signature électronique est un mécanisme analogue à la signature manuscrite permettant de garantir l'intégrité d'un document électronique et prouvant au lecteur du document l'identité de son auteur. De tels mécanismes utilisent les procédés de la cryptographie asymétrique. Si Alice veut garantir l'authenticité d'un message à Bob, elle le signe avec sa clé privée dA. Ensuite, Bob peut utiliser la clé publique d'Alice, eA, pour vérifier la signature. Cette opération consiste à vérifier que la signature a bien été produite par dA et le message original. Un algorithme asymétrique comme RSA, ElGamal, DSA ou encore le ECDSA basé sur les courbes elliptiques, peut être utilisé pour atteindre cet objectif. Un problème courant intervient quand la taille des données à signer est grande. Dans ce cas, la procédure de chiffrement avec un système asymétrique peut devenir très longue. De plus, la taille de la signature elle-même peut devenir déraisonnablement grande. Pour cette raison, les fonctions de hachage sont utilisées. Lorsque Alice veut envoyer un message signé m à Bob, elle suit la procédure suivante. Elle calcule l'empreinte de m à l'aide d'une fonction de hachage H, H(m). Ensuite, elle applique la signature sur le haché. Enfin, le résultat (H(m)) est envoyé à Bob. Bob peut maintenant vérifier qu'Alice a bien produit m en utilisant sa clé publique. Pour ce type d'utilisation, on demande qu'il ne soit pas possible de produire des collisions ou des secondes préimages pour H. Si Charlie est capable de générer deux messages, m, m tels que m m et H(m) = H(m), il peut donner à signer à Alice m au lieu de m et produire ainsi une signature légitime à partir d'un message contrefait, sans qu'Alice puisse s'en apercevoir. De même, si Alice est capable de produire des collisions ou des secondes préimages, elle peut utiliser une collision pour nier plus tard qu'elle a produit un message.

3.4 Protocoles d'engagement

Un protocole d'engagement permet à une personne de s'engager sur une valeur ou sur un message m sans dévoiler son contenu dans l'immédiat. Pour ceci, il suffit de calculer le haché du message et ensuite de diffuser H(m). Le haché du message ne révèle aucune information sur le message lui-même. Par contre, quand le message sera révélé, il suffira de calculer son haché et de le comparer avec la valeur préalablement diffusée. On peut supposer que le secret que nous souhaitons temporairement protéger est un message simple, par exemple une valeur numérique, réponse à un problème de mathématiques difficile. Pour éviter qu'une personne connaissant le haché du secret puisse essayer quelques

valeurs susceptibles d'être la bonne réponse, il est souvent courant de hacher le message concaténé avec une valeur secrète aléatoire k, c'est-à-dire calculer H(m||k). À la révélation du secret, la transmission de m et k suffiront pour vérifier la vérité. Si la fonction de hachage qui a été utilisée pour cette application est résistante aux collisions et aux secondes préimages, alors on peut être sûr de la valeur révélée. Une autre application du même type est l'horodatage ou timestamping en anglais. Il s'agit d'un mécanisme qui associe une heure et une date à un document numérique. Parfois, nous pouvons avoir besoin de prouver qu'un document a déjà été créé à un certain moment dans le temps. Ceci peut par exemple arriver pour prouver que nous possédions la description d'une certaine invention avant de la dévoiler au public. Dans ce cas également, une fonction de hachage peut être utilisée pour protéger le message.

3.5 Générateurs pseudo-aléatoires

Un générateur pseudo-aléatoire de nombres est un algorithme capable de produire une suite de bits qui a les propriétés d'une suite générée aléatoirement, comme l'indépendance et la distribution uniforme des bits. La production de telles suites est nécessaire pour diverses applications cryptographiques, comme par exemple la signature électronique ou la génération des clés pour un chiffrement asymétrique. Les bonnes propriétés statistiques des fonctions de hachage sont souvent exploitées pour cela. Par exemple, si H est une fonction de hachage, initialisée avec une graine c, il est possible de construire une suite pseudo-aléatoire à partir de la suite récurrente xi = H(xi-1||0), où x0 = c, de laquelle on extrait des bits pseudo-aléatoires.

3.6 Les fonctions de hachage à base de chiffrement par blocs

Les fonctions de hachage à base de chiffrement par blocs utilisent un algorithme de chiffrement par blocs pour produire une valeur de hachage. Ces fonctions de hachage sont souvent plus lentes que les fonctions de hachage spécifiquement conçues pour cela, mais elles offrent une sécurité plus forte. Les fonctions de hachage à base de chiffrement par blocs peuvent être implémentées en utilisant différents modes d'opération de chiffrement par blocs, tels que CBC (Cipher Block Chaining), CFB (Cipher Feedback), OFB (Output Feedback) et CTR (Counter). Chacun de ces modes d'opération utilise une technique différente pour transformer le texte en clair en un texte chiffré. Une des fonctions de hachage les plus connues basées sur un bloc de chiffrement est SHA-256 (Secure Hash Algorithm 256 bits). SHA-256 utilise le chiffrement par blocs pour produire un hachage de 256 bits à partir d'une entrée de taille variable. Un autre exemple est Whirlpool, qui utilise également un algorithme de chiffrement par blocs pour produire un hachage de 512 bits. Ces fonctions de hachage basées sur les blocs de chiffrement offrent une sécurité solide contre les attaques de collision et ont été largement utilisées dans de nombreux systèmes de sécurité et protocoles de communication. Cependant, leur mise en œuvre peut être plus complexe que celle des fonctions de hachage à sens unique simples.

3.7 Les fonctions de hachage à base de Cellular Automaton

Les fonctions de hachage basées sur les automates cellulaires (CA) constituent une classe spécifique qui utilise ces modèles mathématiques pour réaliser des opérations de hachage. Les automates cellulaires sont des grilles de cellules qui évoluent dans le temps selon des règles prédéfinies. Dans les fonctions de hachage basées sur les CA, une grille de cellules représente l'état du message d'entrée, tandis que les règles de transition de l'automate cellulaire déterminent le processus de hachage. Ce processus se déroule en plusieurs étapes, chaque itération de l'automate cellulaire représentant une étape du calcul de hachage. Les fonctions de hachage basées sur les CA offrent plusieurs avantages, notamment une distribution uniforme des empreintes de hachage, une forte résistance aux collisions et une efficacité élevée. Elles sont également robustes contre les attaques par force brute et les attaques différentielles. Cependant, bien que prometteuses, ces fonctions de hachage sont moins répandues que des standards comme SHA-2 ou SHA-3, et souvent considérées comme moins évaluées en termes de sécurité. Leur implémentation peut également être plus complexe en raison de l'utilisation d'automates cellulaires pour les calculs de hachage.

Chapitre 4

Sécurité des fonctions de hachage cryptographiques

4.1 Notions de sécurité

Dans le domaine de la sécurité informatique, les fonctions de hachage sont utilisées comme brique de base de différents protocoles cryptographiques. La sécurité de ces protocoles est basée essentiellement sur la résistance de la fonction de hachage à différent types d'attaques.

4.2 Attaques

Les fonctions de hachage cryptographiques peuvent être vulnérables à diverses attaques, ce qui peut avoir des conséquences graves sur la sécurité des données et des systèmes. Voici les principales attaques associées aux fonctions de hachage :

4.2.1 Attaques par Collision

Les attaques par collision visent à trouver deux entrées distinctes qui produisent le même haché. Les algorithmes vulnérables à ces attaques peuvent permettre à un attaquant de modifier un message sans changer son haché, compromettant ainsi l'intégrité des données.

4.2.2 Attaques de Préimage

Les attaques de préimage consistent à trouver une entrée qui correspond à un haché donné. Si une fonction de hachage est vulnérable à cette attaque, un attaquant peut générer des données malveillantes qui produisent le même haché qu'un message légitime.

4.2.3 Attaques de Seconde Préimage

Les attaques de seconde préimage cherchent à trouver une autre entrée qui produit le même haché qu'une entrée donnée. Bien que plus difficiles que les attaques par collision,

elles peuvent toujours compromettre l'intégrité des données si la fonction de hachage est faible.

4.2.4 Attaques par Force Brute

Les attaques par force brute impliquent de tester toutes les combinaisons possibles pour trouver une entrée correspondant à un haché spécifique. La résistance à cette attaque dépend de la taille et de la complexité du haché.

4.2.5 Attaques par Dictionnaire

Les attaques par dictionnaire utilisent des listes précompilées de hachés pour retrouver les entrées d'origine. Les fonctions de hachage faibles rendent ces attaques plus pratiques et efficaces.

4.2.6 Attaques par Birthday

Les attaques par birthday exploitent le paradoxe des anniversaires pour trouver des collisions plus rapidement que par force brute. Elles sont particulièrement efficaces contre les fonctions de hachage produisant des empreintes de taille relativement petite.

4.2.7 Attaques par Canaux Auxiliaires

Ces attaques exploitent les informations divulguées par l'implémentation physique des fonctions de hachage, comme le temps d'exécution, la consommation d'énergie ou les émissions électromagnétiques, pour retrouver les données d'origine ou les clés de chiffrement.

4.2.8 Attaques de Déni de Service

Les attaques de déni de service visent à surcharger un système en exploitant les faiblesses des fonctions de hachage, par exemple en générant de nombreuses collisions pour ralentir ou bloquer le service.

4.3 Conséquences

Les Conséquences sont nombreuses, tels que :

4.3.1 Altération des données

Si une fonction de hachage est faible et sujette aux collisions, un attaquant peut modifier un message sans modifier son haché, ce qui peut entraîner une altération ou une falsification des données. Par exemple, dans les systèmes de vérification de fichiers, une collision pourrait permettre à un attaquant de remplacer un fichier légitime par un fichier malveillant tout en conservant le même haché.

4.3.2 Violation de l'intégrité

Si une fonction de hachage est vulnérable aux attaques de préimage, un attaquant peut trouver une entrée correspondante à un haché donné, ce qui peut compromettre l'intégrité des données. Cela signifie que l'attaquant peut générer des données malveillantes qui correspondent à un haché préexistant, mettant ainsi en péril les systèmes de vérification de l'intégrité des fichiers, les certificats numériques, et autres.

4.3.3 Attaques par contrefaçon

Si une fonction de hachage est faible et peut être inversée facilement, un attaquant peut créer des messages contrefaits avec le même haché que des messages légitimes, ce qui peut mener à des attaques d'usurpation d'identité ou de falsification. Dans le cas de signatures numériques, cela permettrait à un attaquant de faire passer des documents frauduleux pour des documents authentiques.

4.3.4 Compromission des protocoles cryptographiques

De nombreux protocoles cryptographiques, tels que TLS/SSL, reposent sur des fonctions de hachage pour assurer la sécurité. Des faiblesses dans ces fonctions peuvent compromettre la sécurité des communications chiffrées, exposant ainsi les données à des interceptions et des modifications non autorisées.

4.3.5 Perte de confiance

La découverte de faiblesses dans une fonction de hachage peut entraîner une perte de confiance dans les systèmes et les applications qui en dépendent. Cela peut avoir des répercussions économiques et réputationnelles importantes pour les entreprises et les organisations.

4.3.6 Révocation de certificats numériques

Les certificats numériques reposent sur des fonctions de hachage pour garantir l'intégrité et l'authenticité. Si ces fonctions sont compromises, les certificats pourraient être falsifiés, ce qui nécessite une révocation massive de certificats compromis et une réémission, entraînant des perturbations importantes dans les communications sécurisées.

4.3.7 Attaques par force brute et par dictionnaire

Les attaques par force brute et par dictionnaire exploitent les faiblesses des fonctions de hachage pour retrouver les données d'origine à partir des hachés. Si une fonction de hachage est faible, ces attaques deviennent plus pratiques et moins coûteuses, ce qui peut compromettre la sécurité des mots de passe et des données sensibles.

4.3.8 Déni de service

Une fonction de hachage faible peut être exploitée pour lancer des attaques de déni de service. Par exemple, en générant de nombreuses collisions, un attaquant peut surcharger un système de vérification basé sur des hachés, ralentissant ou bloquant le service.

4.4 Avantages

Les fonctions de hachage cryptographiques offrent plusieurs avantages importants pour la sécurité des données et des systèmes :

— Intégrité des Données : Les fonctions de hachage permettent de vérifier l'intégrité des données en produisant un haché unique pour chaque entrée. Toute modification des données entraîne un changement du haché, facilitant ainsi la détection des altérations.
— Identification Unique : Les hachés servent de signatures numériques uniques, permettant d'identifier et de comparer rapidement des fichiers ou des messages sans avoir besoin de les stocker intégralement.
— Protection des Mots de Passe : Les fonctions de hachage sécurisent les mots de passe en stockant des hachés plutôt que les mots de passe en clair, réduisant le risque en cas de fuite de données.
— Efficacité : Les fonctions de hachage sont rapides et efficaces à calculer, même pour de grandes quantités de données, ce qui les rend adaptées à une utilisation dans des applications de haute performance.
— Compatibilité : Les fonctions de hachage sont largement utilisées et supportées par de nombreux standards et protocoles de sécurité, assurant leur compatibilité et leur interopérabilité dans divers systèmes.

4.5 Limites

— Les fonctions de hachage peuvent produire des collisions, ce qui signifie que deux entrées différentes peuvent produire la même valeur de hachage. Cela peut être exploité par des attaquants pour créer des valeurs de hachage falsifiées et altérer les données originales.
— Les fonctions de hachage peuvent être sujettes à des attaques de pré-image, de seconde pré-image et de collision, en fonction de leur conception. Cela peut compromettre la sécurité des données sensibles protégées par ces fonctions de hachage.
— Les fonctions de hachage peuvent être vulnérables à des attaques par force brute ou par dictionnaire, qui consistent à essayer toutes les combinaisons possibles de données jusqu'à ce que la valeur de hachage correspondante soit trouvée.
— Les fonctions de hachage ne sont pas une solution de sécurité globale ; elles ne peuvent pas remplacer d'autres mesures de sécurité telles que le chiffrement. Elles doivent être utilisées en complément d'autres pratiques de sécurité pour assurer une protection complète des données.

Chapitre 5

Les fonctions de hachage sont-elles suffisantes pour la cryptographie ?

5.1 Réponse

Les fonctions de hachage cryptographiques jouent un rôle crucial dans la sécurité des systèmes informatiques. Cependant, se posent des questions quant à leur suffisance pour garantir une sécurité cryptographique complète. Ce chapitre examine cette question en profondeur, en se basant sur les attaques connues, leurs conséquences, ainsi que sur les avantages et les limites des fonctions de hachage.

5.1.1 Pourquoi les fonctions de hachage ne sont pas suffisantes

Les fonctions de hachage, bien qu'extrêmement utiles, ne sont pas suffisantes pour assurer à elles seules une sécurité cryptographique complète pour plusieurs raisons :

1. **Vulnérabilités aux Attaques :**
   (a) **Attaques par Collision :** Trouver deux entrées différentes produisant le même haché compromet l'intégrité des données. La complexité d'une attaque par collision est de $O(2^{n/2})$ pour une fonction de hachage de n bits. Par exemple, pour SHA-1 (160 bits), cela représente environ 2^{80} opérations.
   (b) **Attaques de Préimage et de Seconde Préimage :** Ces attaques, avec des complexités de $O(2^n)$ et $O(2^n)$ respectivement, peuvent permettre de générer des données malveillantes qui produisent le même haché qu'un message légitime.
2. **Limites Structurelles :**
   (a) **Collisions Inévitables :** En raison du pigeonhole principle, les collisions sont inévitables lorsque l'ensemble des entrées dépasse celui des sorties possibles.
   (b) **Attaques par Force Brute et Dictionnaire :** La résistance aux attaques par force brute dépend de la longueur du haché. Par exemple, un haché de 256 bits (comme SHA-256) nécessiterait 2^{256} opérations pour une attaque par force brute, ce qui est actuellement impraticable mais pourrait devenir faisable avec des avancées en informatique quantique.
3. **Faiblesses dans la Pratique :**

(a) **Attaques par Canaux Auxiliaires :** Exploitant des aspects physiques de l'implémentation (comme la consommation d'énergie), ces attaques peuvent contourner la résistance théorique des fonctions de hachage.

(b) **Attaques par Birthday :** Utilisant le paradoxe des anniversaires, ces attaques trouvent des collisions plus rapidement, réduisant la complexité à $O(2^{n/2})$.

5.2 Comment exploiter les fonctions de hachage en cryptographie

Bien que les fonctions de hachage seules ne suffisent pas, elles peuvent être utilisées efficacement en combinaison avec d'autres mécanismes cryptographiques pour créer des systèmes robustes. L'un des exemples les plus marquants est l'utilisation des fonctions de hachage dans la technologie de la blockchain.

5.3 Cryptomonnaies et les technologies de la Blockchain

Dans le contexte de la transformation numérique mondiale, les cryptomonnaies et les technologies de la blockchain se sont imposées comme des innovations majeures, bouleversant les secteurs financiers et technologiques. Cette thèse explore les fondements, les applications et les implications des cryptomonnaies, en mettant l'accent sur le Bitcoin, la blockchain, les problématiques résolues par cette technologie, ainsi que des concepts spécifiques comme le SHA-256 et les arbres de Merkle.

5.3.1 Cryptomonnaies

Définition et Fonctionnement

Les cryptomonnaies sont des devises numériques utilisant des protocoles cryptographiques pour sécuriser les transactions financières, contrôler la création de nouvelles unités et vérifier le transfert d'actifs. Contrairement aux monnaies traditionnelles, elles fonctionnent de manière décentralisée sur des réseaux peer-to-peer, sans nécessiter d'autorité centrale comme une banque ou un gouvernement. Leur caractère décentralisé et sécurisé repose sur des technologies avancées de cryptographie et de consensus distribué.

Avantages et Impact

Les cryptomonnaies offrent plusieurs avantages, tels que des transactions rapides, des frais réduits et une transparence accrue. Elles permettent également l'inclusion financière de populations non bancarisées et offrent des solutions innovantes pour les transferts transfrontaliers. Leur impact se manifeste dans divers secteurs, y compris la finance, le commerce en ligne, et même l'innovation technologique.

5.3.2 Bitcoin

Origines et Principe

Le Bitcoin est la première et la plus connue des cryptomonnaies, créée en 2009 par une personne ou un groupe de personnes sous le pseudonyme de Satoshi Nakamoto. Il permet des transactions en ligne sans passer par une institution financière, offrant ainsi un moyen de transfert de valeur sécurisé, transparent et immuable. Les transactions Bitcoin sont enregistrées sur une blockchain, une base de données décentralisée et distribuée qui garantit l'intégrité et l'immutabilité des transactions. Le Bitcoin a ouvert la voie à de nombreuses autres cryptomonnaies et à la popularisation de la technologie blockchain.

Importance et Adoption

Le Bitcoin a révolutionné la finance numérique en introduisant des concepts tels que la rareté numérique et la décentralisation monétaire. Son adoption a connu une croissance exponentielle, attirant l'attention des investisseurs, des institutions financières et même des gouvernements. Malgré sa volatilité, le Bitcoin reste une référence incontournable dans le monde des cryptomonnaies.

5.3.3 Blockchain (Chaîne de Blocs)

Définition et Fonctionnement

La blockchain est une technologie de registre distribué qui enregistre les transactions de manière transparente, sécurisée et immuable. Chaque bloc de la chaîne contient une liste de transactions et est lié au bloc précédent via un haché cryptographique, formant ainsi une chaîne continue. Cette structure décentralisée empêche la falsification des données et garantit que toutes les transactions sont vérifiées par un réseau de nœuds indépendants.

Caractéristiques et Sécurité

Les principales caractéristiques de la blockchain incluent la décentralisation, la transparence, l'immutabilité et la sécurité. La sécurité de la blockchain repose sur des mécanismes cryptographiques robustes et des protocoles de consensus tels que la preuve de travail (PoW) ou la preuve d'enjeu (PoS). Ces mécanismes rendent les attaques et la falsification des données extrêmement difficiles.

5.3.4 Problématiques Résolues par la Blockchain

Transparence et Traçabilité

La blockchain améliore la transparence en permettant à toutes les parties prenantes de vérifier les transactions de manière indépendante. Chaque transaction enregistrée est visible et vérifiable, ce qui réduit les risques de fraude et de corruption.

Sécurité et Intégrité

La sécurité est renforcée grâce à l'immutabilité des données, rendant les tentatives de falsification ou de fraude extrêmement difficiles. Les transactions sont vérifiées par des nœuds multiples, assurant ainsi l'intégrité des données.

Réduction des Coûts et des Délais

La blockchain réduit les coûts et les délais en éliminant les intermédiaires dans les transactions financières et les chaînes d'approvisionnement. Cela permet des transactions plus rapides et moins coûteuses, augmentant ainsi l'efficacité des processus commerciaux.

Applications Diverses

La blockchain trouve des applications dans de nombreux domaines au-delà des cryptomonnaies. Dans la finance, elle permet des paiements transfrontaliers rapides et sécurisés, ainsi que la tokenisation d'actifs. Dans la logistique et la chaîne d'approvisionnement, elle offre une traçabilité complète des produits, de la production à la consommation. D'autres applications incluent les contrats intelligents, les votes électroniques, la protection des droits de propriété intellectuelle et la gestion des identités numériques.

5.3.5 La Fonction de Hachage SHA-256 dans Bitcoin

Rôle et Importance

SHA-256 (Secure Hash Algorithm 256 bits) est une fonction de hachage cryptographique utilisée dans le réseau Bitcoin pour sécuriser les transactions et maintenir l'intégrité de la blockchain. Chaque transaction est hachée et ces hachés sont ensuite combinés dans un arbre de Merkle.

Minage et Sécurité

Le processus de minage implique la résolution de problèmes cryptographiques en trouvant un haché qui satisfait certaines conditions, garantissant que les blocs ajoutés à la blockchain sont valides. SHA-256 assure que toute modification des données entraînerait un changement significatif du haché, rendant la falsification pratiquement impossible.

5.3.6 Arbre de Merkle et Merkle Root

Définition et Fonctionnement

Un arbre de Merkle est une structure de données utilisée pour organiser et vérifier l'intégrité des transactions dans un bloc de la blockchain. Chaque feuille de l'arbre représente un haché de transaction, et ces hachés sont ensuite combinés par paires et hachés à nouveau jusqu'à ce qu'un seul haché, appelé Merkle Root, soit produit.

Utilité et Sécurité

Le Merkle Root est inclus dans l'en-tête du bloc et représente un résumé cryptographique de toutes les transactions dans ce bloc. Cette structure permet une vérification efficace et sécurisée des transactions sans avoir à vérifier chaque transaction individuellement.

5.3.7 Relation entre Blockchain, Merkle et SHA-256

Intégration et Synergie

La relation entre la blockchain, les arbres de Merkle et SHA-256 est fondamentale pour le fonctionnement sécurisé et efficace des cryptomonnaies de Bitcoin. La blockchain utilise des blocs pour enregistrer les transactions, et chaque bloc contient un Merkle Root qui résume toutes les transactions de ce bloc.

Sécurité et Vérification

Les arbres de Merkle utilisent SHA-256 pour hacher les transactions et combiner ces hachés de manière hiérarchique, créant ainsi une chaîne cryptographique sécurisée. Cette combinaison permet de garantir l'intégrité et l'immutabilité des données, facilitant la vérification des transactions et empêchant les tentatives de falsification ou de fraude.

Synthèse cryptomonnaies

Pour synthétiser, les cryptomonnaies et la technologie de la blockchain représentent une avancée significative dans le domaine de la finance et de la technologie. Leur capacité à offrir transparence, sécurité, et efficacité les rend indispensables dans un monde de plus en plus numérique. L'étude de ces technologies, ainsi que des concepts comme SHA-256 et les arbres de Merkle, est essentielle pour comprendre leur fonctionnement et leur impact potentiel sur divers secteurs

Chapitre 6

Complexité des fonctions de hachage

Les fonctions de hachage sont essentielles en cryptographie pour garantir l'intégrité et l'authenticité des données. Leur complexité, tant en termes de temps que d'espace, joue un rôle crucial dans leur efficacité et leur sécurité. Ce chapitre analyse les différentes dimensions de la complexité des fonctions de hachage, en s'appuyant sur des mesures quantitatives pour une évaluation approfondie.

6.1 Complexité Temporelle

La complexité temporelle, ou le temps de calcul nécessaire pour produire un haché, est une mesure clé de l'efficacité d'une fonction de hachage.

- **MD5 :** 320 ns par byte sur un CPU moderne.
- **SHA-1 :** 400 ns par byte sur un CPU moderne.
- **SHA-256 :** 600 ns par byte sur un CPU moderne.
- **SHA-3 :** 800 ns par byte sur un CPU moderne.
- **bcrypt :** Dépend du facteur de coût (e.g., 2 a la puissance 10), typiquement de l'ordre de centaines de millisecondes pour un mot de passe court.
- **Argon2 :** Dépend des paramètres (e.g., temps, mémoire, et degré de parallélisme), mais généralement plusieurs centaines de millisecondes pour une configuration sécurisée.

6.2 Complexité Spatiale

La complexité spatiale, ou la quantité de mémoire utilisée par la fonction de hachage, est également une considération importante, surtout dans les environnements contraints.

- **MD5 et SHA-1 :** Environ 128 bytes.
- **SHA-256 :** Environ 256 bytes.
- **SHA-3 :** Environ 200 bytes pour SHA3-256.
- **bcrypt :** Peut varier, mais typiquement 4 KB à 8 KB.
- **Argon2 :** Peut être configurée (e.g., 64 MB pour des paramètres de sécurité élevés).

Chapitre 7

Réalisation

Pour illustrer concrètement les concepts théoriques de cette thèse, une démonstration a été réalisée en utilisant JavaScript. L'objectif de cette démonstration est de présenter le processus de hachage pour plusieurs algorithmes couramment utilisés : MD5, SHA-256, SHA-3 et bcrypt. Cette section décrit en détail la réalisation, les résultats obtenus et leur analyse.

7.1 Choix des Algorithmes de Hachage

Les algorithmes de hachage choisis pour cette démonstration représentent une gamme variée en termes de sécurité et d'usage :

- MD5 : Bien que considéré comme obsolète en raison de ses vulnérabilités aux collisions, MD5 est encore largement utilisé pour des contrôles d'intégrité simples.
- SHA-256 : Un standard de l'industrie pour les applications nécessitant une haute sécurité, utilisé notamment dans le protocole Bitcoin.
- SHA-3 : La dernière addition à la famille des fonctions de hachage SHA, offrant une sécurité renforcée et une conception différente des versions précédentes.
- bcrypt : Conçu spécifiquement pour le hachage des mots de passe, il intègre un facteur de coût permettant de ralentir les attaques par force brute.

7.2 Environnement de Développement

La démonstration a été réalisée en utilisant JavaScript, un langage de programmation polyvalent largement utilisé dans le développement web. Pour effectuer les opérations de hachage, plusieurs bibliothèques cryptographiques open source ont été utilisées :

- CryptoJS pour MD5, SHA-256, et SHA-3.
- bcrypt.js pour bcrypt.

Ces bibliothèques permettent de réaliser des opérations cryptographiques complexes de manière efficace et sécurisée.

7.3 Description de la Démonstration

La démonstration consiste en une application web simple où l'utilisateur peut entrer un texte dans un champ de saisie. Après soumission, le texte est haché en utilisant chacun des algorithmes mentionnés ci-dessus, et les résultats sont affichés de manière conviviale.

Interface Utilisateur

L'interface utilisateur se compose d'un champ de saisie pour le texte, d'un bouton de soumission et d'une section d'affichage des résultats. Chaque résultat de hachage est clairement étiqueté et présenté de manière à être facilement comparé avec les autres.

Processus de Hachage

Lorsqu'un utilisateur entre un texte et clique sur le bouton de soumission, le texte est traité par les différentes fonctions de hachage de la manière suivante :

- MD5 : Le texte est haché en utilisant l'algorithme MD5 et le résultat est affiché immédiatement.
- SHA-256 : De même, le texte est haché avec SHA-256 et le résultat est affiché.
- SHA-3 : Le texte est haché avec SHA-3 et le résultat est également affiché.
- bcrypt : Pour bcrypt, le texte est haché avec un facteur de coût par défaut, et le résultat est affiché.

Affichage des Résultats

Les résultats sont affichés sous forme de texte dans une section dédiée de l'interface. Chaque résultat est précédé du nom de l'algorithme de hachage utilisé. Par exemple :

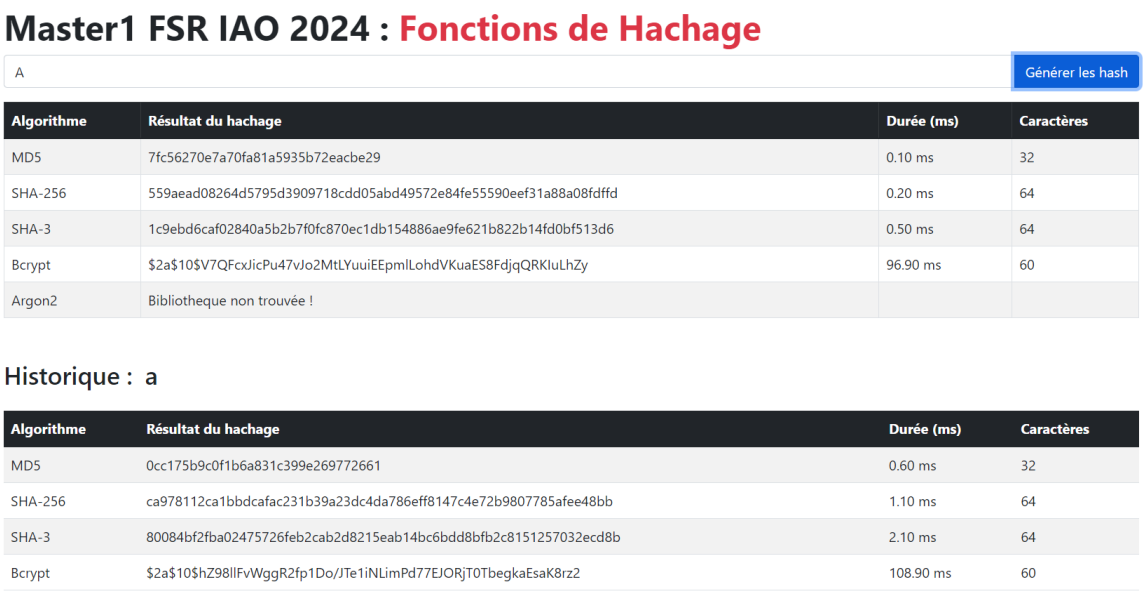


FIGURE 7.3.1 – Démonstration : haché de a et A

Chaque valeur de hachage est unique au texte entré, illustrant ainsi le fonctionnement des différentes fonctions de hachage.

Analyse des Résultats

Les résultats obtenus montrent clairement les différences entre les algorithmes de hachage :

- MD5 : produit un haché de 32 caractères hexadécimaux. Bien que rapide 0., il est vulnérable aux attaques par collision.
- SHA-256 : produit un haché de 64 caractères hexadécimaux, offrant une sécurité beaucoup plus élevée contre les collisions et les attaques de préimage.
- SHA-3 : offre une sécurité renforcée et une structure différente, produisant également un haché de 64 caractères hexadécimaux.
- bcrypt : produit un haché avec un facteur de coût intégré, rendant les attaques par force brute plus difficiles et lentes.

Ces résultats démontrent l'importance de choisir un algorithme de hachage adapté aux besoins de sécurité spécifiques de chaque application.

Conclusion Générale

En conclusion, bien que les fonctions de hachage cryptographiques soient des outils puissants pour assurer l'intégrité et l'authentification des données, elles présentent des vulnérabilités qui limitent leur efficacité en tant que solutions de sécurité autonomes. Les attaques par collision, de préimage et par canaux auxiliaires montrent que les fonctions de hachage ne peuvent pas, à elles seules, garantir une sécurité cryptographique totale. Cependant, en les intégrant judicieusement avec d'autres technologies, comme les blockchains, il est possible de créer des systèmes robustes et sécurisés. L'exemple du Bitcoin, utilisant SHA-256 et les arbres de Merkle, illustre parfaitement comment les fonctions de hachage peuvent être exploitées pour renforcer la sécurité et l'immutabilité des transactions. Ce rapport souligne l'importance de combiner diverses techniques cryptographiques pour répondre aux défis de la sécurité moderne et développer des solutions plus résilientes face aux menaces émergentes.

Bibliographie

[1] Andrew Tanenbaum (Université libre d ?Amsterdam), « Systèmes d ?exploitation », 3ème édition, Nouveaux Horizons, 2008.

[2] https ://developer.mozilla.org/fr/docs/Glossary

[3] Analysis and Design of Cryptographic Hash Functions Bart PRENEEL February 2003

[4] https://www.ssl.com/fr/faq/qu%27est-ce-qu%27une-fonction-de-hachage-cryptographique/

[5] https://www.chireux.fr/mp/TIPE/ADS/fonctions%20de%20hachage.pdf

[6] https://thepressfree.com/definition-des-fonctions-de-hachage-cryptographique/

[7] Hash Functions Based on Block Ciphers Xucjia Lai and Jamcs L. Massey Signal and Information Processing Laboratory Swiss Federal Institute of Technology CI-I-8092 Ziirich, Switzcrland

[8] https://openclassrooms.com/fr/courses/1757741-securisez-vos-donnees-avec-la-cryptographie/6031693-hachez-vos-donnees-menu-menu

Chapitre 8

Annexes

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>NOUHAILA & LEFORT</title>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/5.3.0/js/bootstrap.bundle.min.js"></script>
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/5.3.0/css/bootstrap.min.css">
</head>
<body>
    <div class="container">
        <h1 class="mt-5"> <b>Master1 FSR IAO 2024 :</b> <b class="text-danger">Fonctions de Hachage </b></h1>
        <div class="input-group mb-3">
            <input type="text" id="inputValue" class="form-control" placeholder="Entrez une valeur">
            <button class="btn btn-primary" onclick="generateHashes()">G n rer les hash</button>
        </div>

        <div class="result">
            <table id="hashResults" class="table table-striped table-bordered">
                <thead class="table-dark">
                    <tr>
                        <th scope="col">Algorithme</th>
                        <th scope="col">R sultat du hachage</th>
                        <th scope="col">Dur e (ms)</th>
                        <th scope="col">Caract res</th>
                    </tr>
                </thead>
                <tbody>
                    <tr>
                        <td>MD5</td>
                        <td><span id="md5Result"></span></td>
                        <td><span id="md5Time"></span></td>
                        <td><span id="md5Size"></span></td>
```

```
                    </tr>
                    <tr>
                        <td>SHA-256</td>
                        <td><span id="sha256Result"></span></td>
                        <td><span id="sha256Time"></span></td>
                        <td><span id="sha256Size"></span></td>
                    </tr>
                    <tr>
                        <td>SHA-3</td>
                        <td><span id="sha3Result"></span></td>
                        <td><span id="sha3Time"></span></td>
                        <td><span id="sha3Size"></span></td>
                    </tr>
                    <tr>
                        <td>Bcrypt</td>
                        <td><span id="bcryptResult"></span></td>
                        <td><span id="bcryptTime"></span></td>
                        <td><span id="bcryptSize"></span></td>
                    </tr>
                    <tr>
                        <td>Argon2</td>
                        <td><span id="argon2Result"></span></td>
                        <td><span id="argon2Time"></span></td>
                        <td><span id="argon2Size"></span></td>
                    </tr>
                </tbody>
            </table>
        </div>

        <!-- Historique des r sultats -->
        <div class="history mt-5">
            <h3 style="display: inline-block; margin-right: 10px;">
Historique :</h3>
            <h3 id="historyInput" style="display: inline-block;">
              <!-- La valeur d'input pr c dents sera ajout e ici -->
            </h3>

            <table id="historyTable" class="table table-striped">
              <thead class="table-dark">
                <tr>
                  <th scope="col">Algorithme</th>
                  <th scope="col">R sultat du hachage</th>
                  <th scope="col">Dur e (ms)</th>
                  <th scope="col">Caract res</th>
                </tr>
              </thead>
              <tbody id="historyBody">
                <!-- Les r sultats pr c dents seront ajout s ici --
>
              </tbody>
            </table>
          </div>


    </div>
```

```
87
88      <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js
        /4.0.0/crypto-js.min.js"></script>
89      <script src="https://cdn.jsdelivr.net/npm/bcryptjs@2.4.3/dist/
        bcrypt.min.js"></script>
90        <script src="https://cdnjs.cloudflare.com/ajax/libs/js-sha3/0.8.0/
        sha3.min.js"></script>
91
92      <script>
93              // Initialisation du tableau d'historique
94              let compteur = 0;
95              let histoTab = new Array(13).fill(null); // Initialisation
        d'un tableau vide de taille 13
96
97              function generateHashes() {
98              // V rifier si des hachages pr c dents existent pour les
         ajouter    l'historique
99              if (compteur > 0) {
100                 addToHistory("MD5", histoTab[0], histoTab[1], histoTab
        [2]);
101                 addToHistory("SHA-256", histoTab[3], histoTab[4],
        histoTab[5]);
102                 addToHistory("SHA-3", histoTab[6], histoTab[7],
        histoTab[8]);
103                 addToHistory("Bcrypt", histoTab[9], histoTab[10],
        histoTab[11]);
104                 addToHistoryInput(histoTab[12]);
105             }
106             compteur++;
107
108             const inputValue = document.getElementById("inputValue").
        value;
109
110             // Mesurer le temps de d but
111             const startTime = performance.now();
112
113             // MD5
114             const md5Hash = CryptoJS.MD5(inputValue).toString();
115             const md5Time = performance.now() - startTime;
116             document.getElementById("md5Result").innerText = md5Hash;
117             document.getElementById("md5Time").innerText = `${md5Time.
        toFixed(2)} ms`;
118             document.getElementById("md5Size").innerText = `${md5Hash.
        length} `;
119
120             // SHA-256
121             const sha256Hash = CryptoJS.SHA256(inputValue).toString();
122             const sha256Time = performance.now() - startTime;
123             document.getElementById("sha256Result").innerText =
        sha256Hash;
124             document.getElementById("sha256Time").innerText = `${
        sha256Time.toFixed(
125                 2
126             )} ms`;
127             document.getElementById("sha256Size").innerText = `${
```

```
    sha256Hash.length} ‘;
128
129             // SHA-3 (Keccak)
130             const sha3Hash = sha3_256(inputValue);
131             const sha3Time = performance.now() - startTime;
132             document.getElementById("sha3Result").innerText = sha3Hash;
133             document.getElementById("sha3Time").innerText = ‘${sha3Time
    .toFixed(2)} ms‘;
134             document.getElementById("sha3Size").innerText = ‘${sha3Hash
    .length} ‘;
135
136             // Bcrypt
137             const bcryptHash = dcodeIO.bcrypt.hashSync(inputValue, 10);
138             const bcryptTime = performance.now() - startTime;
139             document.getElementById("bcryptResult").innerText =
    bcryptHash;
140             document.getElementById("bcryptTime").innerText = ‘${
    bcryptTime.toFixed(
141                 2
142             )} ms‘;
143             document.getElementById("bcryptSize").innerText = ‘${
    bcryptHash.length} ‘;
144
145             document.getElementById("argon2Result").innerText =
146                 "Bibliotheque non trouv e !";
147             // Ajouter les r sultats au tableau d’historique
148             histoTab[0] = md5Hash;
149             histoTab[1] = ‘${md5Time.toFixed(2)} ms‘;
150             histoTab[2] = ‘${md5Hash.length}‘;
151             histoTab[3] = sha256Hash;
152             histoTab[4] = ‘${sha256Time.toFixed(2)} ms‘;
153             histoTab[5] = ‘${sha256Hash.length}‘;
154             histoTab[6] = sha3Hash;
155             histoTab[7] = ‘${sha3Time.toFixed(2)} ms‘;
156             histoTab[8] = ‘${sha3Hash.length}‘;
157             histoTab[9] = bcryptHash;
158             histoTab[10] = ‘${bcryptTime.toFixed(2)} ms‘;
159             histoTab[11] = ‘${bcryptHash.length}‘;
160             histoTab[12] = inputValue;
161             }
162
163             // Fonction pour ajouter au tableau d’historique
164             function addToHistory(algorithm, hashResult, elapsedTime,
    hashLength) {
165             const historyBody = document.getElementById("historyBody");
166
167             // Supprimer jusqu’   4 lignes existantes
168             while (historyBody.rows.length >= 4) {
169                 historyBody.deleteRow(0);
170             }
171
172             // Cr er une nouvelle ligne de tableau pour le nouvel
    historique
173             const newRow = historyBody.insertRow(historyBody.rows.
    length);
```

```
174
175             // Ins rer les cellules dans la nouvelle ligne
176             const algorithmCell = newRow.insertCell(0);
177             const resultCell = newRow.insertCell(1);
178             const timeCell = newRow.insertCell(2);
179             const lengthCell = newRow.insertCell(3);
180
181             // Remplir les cellules avec les donn es
182             algorithmCell.textContent = algorithm;
183             resultCell.textContent = hashResult;
184             timeCell.textContent = elapsedTime;
185             lengthCell.textContent = hashLength;
186             }
187             // Fonction pour ajouter au tableau d'historique
188             function addToHistoryInput(inputValue) {
189             const historyInput = document.getElementById("historyInput"
    );
190
191             // Supprimer toutes les lignes existantes
192             while (historyInput.firstChild) {
193                 historyInput.removeChild(historyInput.firstChild);
194             }
195
196             // Cr er un nouveau <p> pour chaque nouvelle entr e
197             const newEntry = document.createElement("p");
198             newEntry.textContent = inputValue;
199
200             // Ajouter la nouvelle entr e    historyInput
201             historyInput.appendChild(newEntry);
202             }
203     </script>
204 </body>
205 </html>
```