Table des matières

Code pour vérifier un paiement dans le bloc #266632	1.13.10

Cet article est une introduction technique au système de monnaie électronique Bitcoin. Il présente les principes de conception des transactions et de la blockchain. Il est basé sur l'article fondateur de Satoshi Nakamoto et le site Wiki de bitcoin.org de la communauté Bitcoin.

Les détails techniques sont développés en annexe. Une première annexe détaille le calcul du jeton horodateur illustré par du code JavaScript pour Node.js. Une seconde annexe détaille la structure d'une transaction et le code JavaScript pour Node.js de vérification d'un paiement.

English version

Historique des révisions :

- 3 juillet 2016 : Première publication
- 9 juillet 2016 : Révision afin de préciser la notion de jeton d'horodatage
- 16 octobre 2016 : Révision de la section *3. Les transactions* et de la section *10. La fraude de la double-dépense*
- 6 mars 2017 : Traduction en français de l'annexe 2

Les objectifs du système Bitcoin

L'objectif fonctionnel

L'objectif exprimé dans l'article de Satoshi Nakamoto en 2008 est de proposer une monnaie électronique sans intermédiaire : ni tiers de confiance pour le paiement, ni autorité de contrôle pour la création monétaire. Il s'appuie pour cela sur un registre public des transactions infalsifiable et vérifiable par tous, mais dont les parties prenantes restent anonymes. Le contexte d'application de cette monnaie est le commerce électronique sur internet.

L'objectif technique

Pour réaliser ce système, le défi majeur est de remplacer un serveur central par un réseau d'agents qui mutualisent leurs ressources pour créer un système décentralisé d'horodatage des transactions, afin d'obtenir un enregistrement séquentiel et irrévocable des transactions. Il doit notamment contrecarrer la fraude de la double-dépense, laquelle consiste à utiliser les mêmes bitcoins pour payer simultanément deux commerçants.

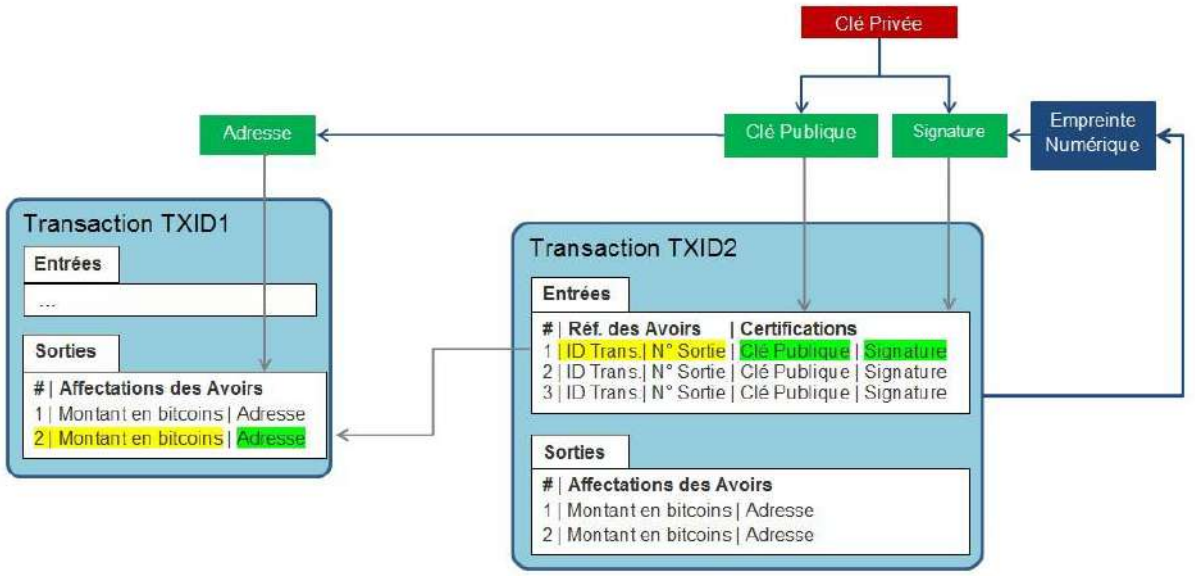
Les éléments constitutifs de la blockchain du réseau Bitcoin sont :

- Les transactions chaînées par un algorithme de signature digitale ;
- Les blocs qui constituent les "pages" du registre des transactions ;
- Les jetons qui garantissent l'ordre des blocs du registre (blockchain), en utilisant un algorithme d'empreinte numérique (hash) et de preuve-de-travail (proof-of-work) ;
- Un logiciel ouvert et gratuit qui implémente un protocole pour un système multi-agents (nodes) ;
- Les agents horodateurs mis en place par les "mineurs" (gold miners), créateurs de blocs et de jetons (timestamps) ;
- Un principe de consensus, basé sur la preuve de travail, qui rend le système sûr tant que les mineurs honnêtes détiennent plus de 50% de la puissance de calcul total.

Les transactions

Nous présentons ici les transactions les plus communément utilisées (Pay to Public Key Hash).

Le schéma ci-dessous présente la structure d'une transaction et les verrous posés via la clé privée de l'utilisateur.



La structure d'une transaction

Une transaction Bitcoin enregistre une opération de transfert d'avoirs entre les utilisateurs du réseau Bitcoin. La transaction Bitcoin est constituée de deux tables, la tables des entrées, qui liste les avoirs dépensés par un ou plusieurs payeurs (inputs) et la table des sorties (outputs), qui liste les nouveaux avoirs affectés à un ou plusieurs destinataires de paiement. Lorsqu'un destinataire de paiement veut à son tour utiliser un avoir en tant que payeur, une nouvelle transaction est créée, et cet avoir apparaîtra dans la table des avoirs dépensés.

C'est la référence de l'avoir qui apparait dans la table des entrées. Cette référence est constituée de l'identifiant de la transaction qui l'a affecté, et de son numéro d'ordre dans la table des sorties.

La somme total des avoirs affectés doit être inférieure à la somme des avoirs dépensés. S'il y a une différence, cette différence sera affectée automatiquement au mineur qui aura enregistré cette transaction dans un bloc. Cette différence sera considérée comme un frais de transaction.

Identifiant de Transaction

L'identifiant de transaction (txid) est l'empreinte cryptographique d'une transaction. Il est utilisé comme identifiant unique de la transaction. Comme chaque transaction référence la transaction précédente, les agents peuvent vérifier qu'un avoir n'est pas dépensé deux fois dans la blockchain.

Cumuler et fractionner les avoirs

Dans sa forme la plus simple, avec un unique payeur et un unique destinataire de paiement, la transaction permet de transférer la totalité d'un avoir d'un utilisateur à l'autre. Cependant, la représentation des transactions est suffisamment flexible pour couvrir d'autres cas de paiement :

- En spécifiant plusieurs avoirs du même payeur, la transaction permet de cumuler des avoirs pour atteindre la somme attendue par le destinataire de paiement.
- En ajoutant le payeur parmi les destinataires de paiement, la transaction permet d'utiliser un avoir précédemment perçu, qui excède la somme attendue par le destinataire de paiement, et créer un nouvel avoir pour rendre la monnaie au payeur.
- En spécifiant plusieurs destinataires de paiement, il est aussi possible de combiner deux achats en une seule transaction ; par exemple pour payer un billet d'avion et une chambre d'hôtel.
- Enfin, en spécifiant plusieurs payeurs, il est possible d'associer les avoirs de ces payeurs pour un paiement unique.

La transaction initiale

Un seul type de transaction déroge à cette représentation, ce sont les transactions initiales (coinbase transaction). Ces transactions créent uniquement un avoir ex-nihilo pour rémunérer les "mineurs" qui allouent leur puissance de calcul pour créer la blockchain.

Le porte-monnaie électronique

La structure de la transaction est gérée par le porte-monnaie électronique bitcoin (wallet). Celui-ci se charge de collecter tous les avoirs disponibles d'un utilisateur ; c'est à dire les avoirs qui ne sont pas encore dépensés. A la façon d'un compte bancaire, il permet d'obtenir son solde en bitcoins. Pour effectuer un paiement, le logiciel se charge de créer la transaction en fonction du montant à payer et des avoirs disponibles.

La clé privée

La clé privée est un code cryptographique généré aléatoirement et tenu secret par l'utilisateur.

A cette clé privée est associée un clé publique, qui jouera le rôle d'un numéro de compte associé à un avoir. Pour faciliter son utilisation, la clé publique est codée sous une forme alphanumérique plus lisible que son format brut. Sous cette forme, la clé est appelée "adresse Bitcoin". C'est ce format qui est effectivement utilisé et communiqué par le destinataire de paiement au payeur.

Ainsi pour recevoir un paiement, une clé privée est générée, une clé publique est calculée et transformée en adresse bitcoin, qui sera communiquée au payeur pour être inscrite parmi les avoirs affectés de la transaction créée par le payeur.

Pour effectuer un paiement, un utilisateur crée une nouvelle transaction, qui référencera des avoirs collectés pour une dépense. Pour chaque avoir collecté, une ligne de la table des entrées sera créée. Cette ligne référence l'avoir dépensé par l'identifiant de la transaction qui a affecté cet avoir et le numéro de l'avoir dans la table des sorties. Puis deux informations de certification sont ajoutées:

- la clé publique : elle permet de vérifier que la clé publique utilisée pour la signature correspond bien à l'adresse bitcoin. Si la signature est créé avec une autre paire de clé publique/privée, les agents du réseau invalideront la transaction
- la signature : elle joue deux rôles,
  - elle atteste que l'utilisateur détenteur de la clé privée a confirmé cette dépense (principe de non-répudiation [1]). En effet, la signature ne peut être calculée qu'avec la clé privée, mais peut être vérifiée avec la clé publique.
  - elle verrouille la dépense pour cette transaction (principe d'intégrité[2]), en effet la signature est calculée avec l'empreinte numérique de la nouvelle transaction, en fait légèrement modifiée, les champs de certification sont remplacés par des 0.

Ainsi seul le détenteur de la clé privée pourra générer l'adresse pour une nouvelle affectation, et sera en capacité de fournir les informations qui permettent la dépense. Les transactions sont chaînées entre elles, car elles incorporent l'identifiant des transactions précédentes, permettant de remonter l'origine d'un avoir jusqu'à la transaction initiale qui a créé les bitcoins ex-nihilo.

L'algorithme de signature cryptographique permet à chaque agent du réseau, qui diffuse cette transaction ou qui l'enregistre dans un bloc, de vérifier la cohérence, de l'adresse, de la clé publique et de la signature, sans avoir accès à la clé privée.

> [1]. La non répudiation garantit qu'une transaction ne peut être niée. ↩

> 2

> [2]. L'intégrité garantit que les données sont bien celles que l'on croit être. ↩

Contrats intelligents

Les clés publiques et les signatures sont en fait stockées comme des fragments de scripts qui seront exécutés par les agents pour effectuer la vérification. Cette vérification n'est donc pas codée par les agents eux-mêmes mais par les transactions. L'exécution de ces scripts aura pour résultat les vérifications décrites précédemment. Ces scripts sont appelés des contrats intelligents (smart contract).

Ces scripts permettent de définir d'autres modalités de paiement telles que la signature multiple pour autoriser la dépense d'un avoir.

L'une de ces modalités consiste à terminer la chaîne des paiements en injectant une donnée dans la transaction. L'avoir en bitcoins se trouve alors "transformé" en donnée (bitcoin burning). La donnée injectée devient une information certifiée et publiquement disponible (proof-of-existence). Par exemple, cette donnée peut être l'empreinte numérique d'un document, l'intégrité de ce document est alors certifiée sans l'implication d'un tiers de confiance.

Les blocs

Les transactions sont enregistrées dans des blocs. Les blocs sont eux-mêmes chainés les uns aux autres pour former la blockchain. Les blocs sont diffusés le plus largement possible sur le réseau Bitcoin, afin d'être validé par les agents et ajoutés à la blockchain. Une transaction est valide si elle est enregistrée dans un bloc ajouté à la blockchain depuis suffisamment longtemps, c'est à dire avec un nombre suffisant de blocs successeurs, donc une chaîne de blocs qui a fait l'objet d'un consensus.

Une bloc est structuré en 2 parties : un en-tête et un corps. L'en-tête permet de restituer le jeton d'horodatage.

```
Structure d'un bloc :
+-------------------------------------------------------------------+
| En-tête du bloc :                                                 |
| - Jeton d'horodatage du bloc précédent                            |
| - Racine de l'arbre de Merkle (Empreinte numérique des transactions)|
| - Horodatage (date et heure)                                      |
| - La cible de la preuve de travail                                |
| - Le "nonce" du jeton (preuve de travail)                         |
+-------------------------------------------------------------------+
| Corps du bloc :                                                   |
| - La table des transactions                              |
+-------------------------------------------------------------------+
```

Les jetons d'horodatage

Le jeton d'horodatage est une empreinte numérique qui authentifie un bloc comme chainon de la blockchain.

Le jeton d'horodatage est créé avec les données suivantes:

- le jeton du bloc précédent,
- une empreinte numérique qui reflète le contenu du bloc (racine de l'abre de Merkle des transactions),
- le "nonce" (une valeur numérique qui satisfait la preuve de travail),
- l'horodatage,
- la cible de la preuve de travail.

Le nonce est une valeur numérique qui doit satisfaire un critère lequel exige un temps de calcul aléatoire et long. Ce critère est appelé la preuve de travail. Une fois cette valeur calculée, vérifié que le nonce satisfait au critère est immédiat.

Tout changement dans le contenu d'un bloc invaliderait le jeton, et imposerait un nouveau calcul du jeton. Comme chaque jeton est calculé avec le jeton précédent, il faudrait aussi recalculer les jetons des blocs suivants. La preuve-de-travail diminue le risque de réécrire un bloc, d'insérer un bloc, ou de retirer un bloc, au fur et à mesure que des blocs sont ajoutés.

Un système multi-agents

Le réseau Bitcoin est le réseau internet utilisé comme un réseau de pair-à-pair. Tous les participants du réseau Bitcoin ont ainsi le même statut ; aucun participant ne peut se prévaloir d'une quelconque légitimité supérieure. Chaque participant est considéré comme un pair vis-à-vis des autres.

Chaque participant du réseau utilise un logiciel qui est à la fois :

- une interface Homme-Machine pour gérer son porte-monnaie de bitcoins, c'est à dire ses avoirs enregistrés dans la blockchain, et pour effectuer des paiements via les transactions ;
- un agent autonome, c'est à dire un "bot" qui réagit aux messages véhiculés par les autres agents.

Le rôle des agents (full node) est de supporter ce réseau de pair-à-pair :

- **Assurer la diffusion des messages**. Deux types principaux de messages sont diffusés pour gérer la monnaie Bitcoin, la transaction et le bloc. La diffusion des messages est faite par propagation. Lors de la connexion au réseau, chaque agent sélectionne de façon aléatoire un pool d'agents de contact et diffuse chaque message reçu à ses contacts qui relayeront à leur tour le message, jusqu'à atteindre de proche en proche tous les agents connectés. Ce mécanisme offre une forte fiabilité, si des messages sont perdus ou retardés par certains, ils seront quand même propagés.
- **Copier localement le registre des transactions** afin que chaque participant puisse connaitre l'état de la blockchain, et informer ses pairs qui peuvent à tout moment quitter ou rejoindre le réseau.
- **Implémenter le protocole**, vérifier et relayer les messages, notamment la difficulté attendue de la preuve de travail ajustée pour réguler la création des blocs.

Certains participants peuvent utiliser le logiciel dans un mode allégé. Ils ne stockeront pas une instance complète de la blockchain. Ils devront interroger les autres agents pour obtenir une vue partielle des blocs afin de vérifier des transactions dont ils sont destinataires (simple payment verification).

Les mineurs

Les mineurs sont des individus ou organisations qui mettent en place des agents horodateurs.

Comme les autres agents, chaque agent horodateur collecte les blocs diffusés sur le réseau pour créer localement sa propre instance de blockchain. Chaque agent horodateur collecte en plus les transactions pour créer un nouveau bloc en calculant le jeton associé. Dès qu'un nouveau bloc dispose de son jeton, l'agent le diffuse le plus largement possible de façon à prendre à témoin les autres agents pour qu'ils adoptent le plus rapidement possible ce bloc à leur propre instance de blockchain.

Le caractère aléatoire du calcul pour satisfaire la preuve de travail met les agents en compétition.

Pour inciter les mineurs à investir des ressources et de l'électricité, les mineurs sont habilités à ajouter une transaction spéciale dans un bloc. Cette transaction permet d'attribuer une récompense au mineur par prélèvement de frais sur les transactions collectées ou bien par création monétaire ex nihilo.

Le principe du consensus

Dans le cas ou deux blocs parviennent quasiment simultanément à un agent horodateur deux instances de blockchain sont mises en concurrences. Dans ce cas le protocole prévoit que l'agent horodateur conserve les deux instances de blockchain ainsi créées, et s'applique à étendre la première.

Le principe du consensus consiste à opter pour la plus longue de ses instances, dès qu'elle apparaitra. Elle représentera en effet le plus grand investissement en calcul de la communauté des agents horodateurs. Tant que les agents horodateurs honnêtes détiennent plus de 50% de la puissance de calcul du réseau elle sera sure.

Tout agent malhonnête qui tenterait de proposer une instance alternative sera mis en minorité vis à vis de ses pairs. En effet le système des jetons d'horodatage, l'empêchera d'ajouter les blocs venus du groupe majoritaire. De même ses propres blocs ne seront pas ajoutés à la plus longue des blockchains.

La plus longue instance de blockchain apparait alors comme l'instance de blockchain qui fait consensus; est donc l'instance reconnue à la majorité. Si un destinataire de paiement s'assure que la transaction sera bien enregistrée dans un bloc suivi de suffisamment de blocs, il obtiendra la certitude que cette transaction est bien enregistrée dans la plus longue instance qui a fait l'objet de ce consensus; et qu'elle sera conservée. C'est la raison pour laquelle un mineur ne peut disposer immédiatement de la transaction spéciale qui le rétribue pour son travail. Cette transaction devient valide lorsque suffisamment de blocs ont été ajoutés à la suite du bloc qu'il a créé.

Le principe du consensus se substitue à une autorité de contrôle qui veillerait à ce que chaque mineur respecte le protocole. On peut noter ici que le comportement de la majorité des mineurs fait loi ; même si ce comportement est jugé arbitraire (comme se reporter sur l'instance de blockchain du premier bloc reçu) voire même dans certains cas erroné (si la majorité des mineurs utilise un logiciel commun bogué). Ce n'est donc par la spécification du protocole qui compte mais l'implémentation du protocole par le logiciel utilisé par la majorité des mineurs. C'est pourquoi pour amorcer le système, le partage d'un logiciel gratuit et ouvert est une condition nécessaire.

L'autorégulation

Le système se régule en fonction du trafic, et de la puissance de calcul des agents horodateurs. L'ajustement de la preuve-de-travail tous les 2016 blocs est une règle du protocole. Comme toute règle consensuelle, si un mineur ne respecte cette preuve-de-travail ajustée, il sera mis en minorité, et ses blocs seront rejetés par la majorité. La fréquence de 10 minutes correspond à une projection, estimée en 2008, des capacités de stockage des ordinateurs dans les années à venir, notamment pour conserver les en-têtes de blocs en mémoire vive.

Si la difficulté de la preuve-de-travail augmente, exigeant une puissance de calcul de plus en plus grande, les mineurs peuvent alors mutualiser leur ressource dans des pools de mineurs pour atteindre la puissance de calcul requise.

Enfin le nombre moyen de transactions par bloc est induit par les frais prélevés sur les transactions, car ceux-ci compensent le coût en électricité requis par la preuve-de-travail. La taille maximale d'un bloc est cependant fixée à 1 Mo afin de garantir une diffusion rapide des blocs sur le réseau.

La gouvernance

Bitcoin étant un système fortement décentralisé, il n'y a pas de gouvernance, ce qui peut poser problème si le protocole doit évoluer. De façon générale, les nouvelles versions du logiciel Bitcoin Core garde une compatibilité ascendante pour ne pas mettre en minorité les nouveaux agents et assurer son adoption.

La fraude de la double-dépense

Principe de la fraude

La double dépense consiste à émettre deux transactions qui dépensent le même avoir : la première transaction est émise pour payer un premier destinataire, la seconde transaction est émise pour payer un complice ou le pirate lui-même, afin de récupérer la somme dépensée.

Afin que la fraude ne soit pas immédiatement découverte, ces deux transactions doivent coexister dans deux instances concurrentes de la blockchain. La seconde instance doit devenir la plus longue blockchain pour que la fraude réussisse. Dans ce cas, la première transaction ne pourra pas être recyclée dans un autre bloc car elle sera jugée invalide car incompatible avec la seconde transaction et sera donc finalement rejetée. Si entre les deux transactions, le premier destinataire a accepté le paiement, il ne découvrira qu'après coup que cette transaction a été rejetée. Puisque les participants sont anonymes, il ne pourra se retourner contre le fraudeur.

Contrecarrer la fraude

Pour veiller à ne pas être victime de la fraude, le destinataire de paiement doit s'assurer que la transaction émise par le payeur est bien enregistrée dans un bloc de la plus longue des instances et que cette instance est pérenne. C'est pourquoi, il doit attendre qu'un nombre suffisant de blocs succèdent au bloc qui enregistre sa transaction, avant d'accepter le paiement.

La structure d'un bloc par l'arbre de Merkle permet de restituer un bloc allégé ; le destinataire de paiement peut ainsi vérifier une transaction sans avoir à télécharger le contenu complet des blocs.

Si le nombre de bloc successeurs est suffisamment élevé, et que le pirate ne dispose pas de plus de 51% de la puissance de calcul total alors le risque de double-dépense tend vers zéro (cf. la démonstration par Satoshi Nakamoto basé sur la probabilité de Poisson).

Annexe : Implémentation d'un en-tête de bloc

L'empreinte numérique

Une fonction de hachage cryptographique transforme un message en un code numérique appelé l'empreinte numérique du message. Ce code reflète entièrement le message. La même fonction de hachage appliquée à un message altéré produira un code numérique distinct de son code initial.

L'algorithme SHA-256

L'algorithme SHA-256 (Secured Hash Algorithm) spécifie une méthode de hachage qui produit une empreinte numérique sur 256 bits. Elle est dite sécurisée car l'altération d'un seul bit du message, produit une empreinte numérique distincte. Il est donc impossible d'altérer un message tout en conservant la même empreinte.

La fonction *hash* ci-dessous applique l'algorithme SHA-256 au message passé en paramètre, le résultat est affiché dans sa représentation hexadécimale :

```
// Calculate a hash code from a binary buffer
var crypto = require("crypto");
var hash = function(buffer) {
    var f = crypto.createHash('sha256');
    var h = f.update(buffer);
    return h.digest();
};

> var message = new Buffer('hello world');
> message.toString('hex');
'68656c6c6f20776f726c64'
> var hashCode = hash(message);
> hashCode.toString('hex');
'b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9'
```

Le "nonce"

Un nonce est une valeur arbitraire adjointe à un message pour influer sur l'empreinte numérique obtenue ; on applique la fonction de hachage au message concaténé à un nonce codé sur 32 bits avec la convention Little-Endian.

> La convention Little-Endian ordonne les octets du poids le plus faible vers le poids le plus fort, la convention Big-Endian ordonne les octets du poids le plus fort vers le poids le plus faible. Par exemple le nombre 1 est représenté en hexadécimal sur 32 bits par 00000001 avec la convention Big-Endian et 01000000 avec la convention Little-Endian.
>
> Le nombre 1 sur 32 bits avec la convention Big-Endian :
>
> ```
> Octet 3 | Octet 2 | Octet 1 | Octet 0
> --------+---------+---------+--------
>      00 |      00 |      00 |      01
> ```
>
> Le nombre 1 sur 32 bits avec la convention Little-Endian :
>
> ```
> Octet 0 | Octet 1 | Octet 2 | Octet 3
> --------+---------+---------+--------
>      01 |      00 |      00 |      00
> ```

Dans le code ci-dessous, nous représentons le nonce avec un Buffer d'octets, indépendamment de la convention Little-Endian/Big-Endian propre au processeur, puis nous concaténons le message avec le nonce, pour obtenir une nouvelle empreinte numérique.

```
var numberToInt32LE = function (n) {
    var buffer = new Buffer(4);
    buffer.writeUInt32LE(n,0);
    return buffer;
};

> var nonce = numberToInt32LE (1000);
> nonce.toString('hex');
'e8030000'

> var message = new Buffer('hello world');
> var hashCode = hash(Buffer.concat([message,nonce]));
> hashCode.toString('hex');
'51b2583117a8e0c8551883127bba6bdc3fe6603ac58ce4ad5c1c2a6def9be485'
```

La preuve-de-travail

La preuve-de-travail vise

- à rendre difficile le calcul d'une empreinte numérique en exigeant du temps d'exécution du processeur ;
- à rendre aléatoire le temps nécessaire à ce calcul (cf. le principe du consensus).

L'algorithme SHA-256 produit un nombre qui peut être représenté sur 256 bits, soit un nombre entre 0 et $2^{256} - 1$. Si on considère la fonction de hachage basée sur l'algorithme SHA-256 comme une fonction pseudo-aléatoire, la probabilité d'obtenir un nombre qui commence par N bits à 0 pour sa représentation sur 256 bits est de $1/(2^N)$. Cela revient à segmenter l'intervalle des nombres de 0 à $2^{256} - 1$, en 2, 4, 8, ...2^N segments, et à évaluer la probabilité d'obtenir un nombre du premier segment.

Le tableau ci-dessous donne un échantillon de quelques probabilités :

Nombre de bits à zéro	Probabilité
1	1 / 2
2	1 / 4
3	1 / 8
4	1 / 16
5	1 / 32
16	1 / 65536

Une preuve-de-travail spécifie le nombre N de bits à 0 que doit satisfaire une empreinte numérique par l'adjonction d'un nonce. Pour satisfaire cette contrainte, il faut rechercher un nonce, avec une probabilité de succès de $1/(2^N)$. Cette recherche peut être faite simplement par une itération du nonce jusqu'à satisfaire la contrainte, ou par toute autre méthode, par exemple en tirant des valeurs au "hasard". Quelle que soit la méthode, elle nécessitera du temps d'exécution du processeur, un temps d'autant plus long que la probabilité de succès sera faible.

Pour tester l'empreinte numérique, nous utiliserons la fonction *toReverseHexaNotation* qui inverse la représentation hexadécimal d'un nombre codé sur 32 bits ou 256 bits avec la convention Little-Endian

```
Buffer.prototype.toReverseHexaNotation = function ()
{
    var hexa = "";
    for (var i = this.length-1; i >= 0; i--) {
       var digits =  this[i].toString(16);
        hexa += ("0" + digits).slice(-2); // Add "0" for single digit
    }
    return hexa;
};

/* Return a nonce value for a number of expected bits.
 * Warning: May take a while if the proof-of-work is difficult
 * proof is a string of '0' characters, each character is
 * a hexadecimal digit, so it is 4 zero-bits
 */
var calculateProofOfWork = function(proof, message) {
   var len = proof.length;
   for(var nonce=0;;nonce++){
        var nonceLE= numberToInt32LE(nonce);
        var hashCode = hash(Buffer.concat([message, nonceLE]));
        var result = hashCode.toReverseHexaNotation();
        if (result.substr(0, len) == proof)
            return nonceLE;
    }
};

> var message = new Buffer('hello world');
> var nonce = calculateProofOfWork('00', message);
> nonce.toString('hex');
'0e000000'
```

Une fois la valeur du nonce trouvée, la vérification de la preuve-de-travail est immédiate :

```
> var hashCode = hash(Buffer.concat([message, nonce]));
> hashCode.toReverseHexaNotation();
'003a669f5c15dd75046bae1661f6a7326bbb80ec217080bff5c97286dc03d7c6'
```

Le temps et l'effort de calcul pour trouver le nonce augmente à mesure que la probabilité de succès décroit :

```
var testProofOfWork = function (message, loops) {
    var proof = "0";
    for (var i = 0; i < loops; i++) {
        var t1 = Date.now();
        var nonce = calculateProofOfWork(proof, message);
        var t2 = Date.now();
        var hashCode = hash(Buffer.concat([message, nonce]));
        console.log ("hash:", hashCode.toReverseHexaNotation(),
                     "nonce:", nonce.toReverseHexaNotation(),
                     "elapsedTime:", t2-t1);
        proof += "0";
    }
};
```

```
> testProofOfWork(message, 5);
hash: 003a669f5c15dd75...80bff5c97286dc03d7c6 nonce: 0000000e elapsedTime: 1
hash: 003a669f5c15dd75...80bff5c97286dc03d7c6 nonce: 0000000e elapsedTime: 1
hash: 0001c1c7a764d470...76d596b136ef255ba911 nonce: 00001f0a elapsedTime: 254
hash: 00008e8dc456ce49...a36d96dfafab3f18fba0 nonce: 0000fca8 elapsedTime: 1486
hash: 000003aaa63ca127...35ba4de259c2625b617f nonce: 0008e973 elapsedTime: 12241
```

La cible de la preuve-de-travail

Le protocole Bitcoin a fait évoluer la preuve-de-travail. Elle n'est plus spécifiée comme un nombre de bits à 0 mais comme une valeur maximale que l'empreinte numérique peut atteindre. Cette valeur maximale est appelée la cible.

Cette cible est représentée sous une forme compacte de 32 bits, appelée *bits*. Cette représentation est basée sur les fonctions OpenSSL *BN_bn2mpi()* et *BN_mpi2bn()* :

- Les 23 bits de poids faible représentent une valeur.
- Le 24ième bit représente le signe négatif.
- L'octet de poids fort est utilisé pour établir un décalage à gauche en nombre d'octets.

La valeur correspondante s'obtient par le calcul suivant :

```
var bigInt = require("big-integer");

var bitsToTarget = function (bits) {
    bits = bigInt(bits);
    var sign = bits.and(0x00800000).shiftRight(24).toJSNumber();
    var exponent = bits.and(0xFF000000).shiftRight(24).toJSNumber();
    var mantissa = bits.and(0x007FFFFF);
    var target = mantissa.times(Math.pow(-1,sign)).shiftLeft(8 * (exponent-3));
    return target;
}

> bitsToTarget(0x04012345).toString(16);
'1234500'
```

La valeur limite choisie pour la cible, soit l'effort minimum de preuve-de-travail exigé, commence par 32 bits à 0, soit :

```
0x00000000FFFF0000000000000000000000000000000000000000000000000000
```

soit dans sa forme compacte sur 32 bits :

```
0x1d00ffff
```

L'indice de difficulté

L'indice de difficulté est calculé par le rapport de la valeur limite de la preuve-de-travail avec la cible courante. Cet indice donne une estimation de l'effort de calcul nécessaire pour satisfaire la preuve-de-travail. Plus l'indice est grand, plus l'effort de calcul est important. Une valeur de l'indice égale à 1 correspond à l'effort minimum.

```
> var bigInt = require("big-integer");
> var limit = bigInt("00000000FFFF0000000000000000000000000000000000000000000000000000"
,16);
> var target = bigInt("00000000000404CB000000000000000000000000000000000000000000000000
0",16);
> var difficultyIndex = limit.divide(target).toJSNumber();
16307
```

L'ajustement de la preuve-de-travail

La cible de la preuve-de-travail doit être ajustée de façon à générer un bloc en moyenne toutes les 10 minutes.

Si un bloc est généré en moyenne toutes les 10 minutes, alors sur une période de 14 jours, 2016 blocs doivent être générés (2016 blocs = 14 jours * 24 heures * 6 blocs par heure). Ainsi tous les 2016 blocs, la cible est ajustée en la multipliant par un facteur. Ce facteur est déduit de la différence de temps de création effectif des 2016 blocs avec la durée attendue de 14 jours.

Le jeton d'horodatage

Le jeton d'horodatage de chaque bloc est calculé avec l'empreinte numérique de la concaténation des éléments de son en-tête. Cet en-tête est constitué :

- du numéro de version de la structure du bloc (*version*) ;
- du jeton d'horodatage du bloc précédent (*previousToken*) ;
- de l'empreinte numérique du contenu du bloc (*merkleRootHash*) ;
- de l'horodatage du bloc sur 32 bits (*time*);
- de la valeur compacte de la cible de la preuve-de-travail (*bits*) ;
- du nonce qui permet de satisfaire la preuve-de-travail (*nonce*).

> L'empreinte numérique du contenu du bloc est calculée via un arbre de Merkle, la racine de l'arbre de Merkle est l'empreinte numérique pour l'ensemble des transactions enregistrées.

```
// Header from block 125552
var header = {
    version: 1,
    previousToken: '00000000000008a3a41b85b8b29ad444def299fee21793cd8b9e567eab02cd81',
    merkleRootHash: '2b12fcf1b09288fcaff797d71e950e71ae42b91e8bdb2304758dfcffc2b620e3'
,
    time: new Date ("Sat May 21 2011 17:26:31 GMT+0000 (UTC)"),
    bits: 440711666,
    nonce: 2504433986
};
```

Tous les éléments de l'en-tête sont considérés comme des nombres sur 32 bits ou 256 bits, encodés avec la convention Little-Endian. La fonction *serializeHeader* permet de compiler ces données dans des buffers, avec la convention Little-Endian :

```
var serializeHeader = function (header) {
  var buffers = [];
  buffers.push (numberToInt32LE(header.version));
  buffers.push (hexaNotationToInt256LE(header.previousToken));
  buffers.push (hexaNotationToInt256LE(header.merkleRootHash));
  buffers.push (dateToInt32LE(header.time));
  buffers.push (numberToInt32LE(header.bits));
  buffers.push (numberToInt32LE(header.nonce));
  return Buffer.concat (buffers);
};

var dateToInt32LE = function (date) {
    var time = date.getTime() / 1000; // remove milliseconds
    return numberToInt32LE(time);
};

var hexaNotationToInt256LE = function (hexa)
{
    var bytes = new Array(32);
    for (var i = 0, j = 31, len = hexa.length; i < len; i+=2, j--) {
        bytes[j] = parseInt(hexa[i]+hexa[i+1],16);
    }
    return new Buffer(bytes);
};

> serializeHeader (header).toString('hex');
'0100000081cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000e320b6c2fffc8
d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122bc7f5d74df2b9441a42a14695'
>
```

Une première empreinte numérique d'un bloc est calculée sur la concaténation des éléments de l'en-tête, puis une seconde empreinte est calculée sur ce résultat, pour des raisons historiques et par prévention. Le jeton d'horodatage est cette seconde empreinte numérique.

```
> var hashcode = hash ( hash (serializeHeader (header) ) ); // hash twice
> hashcode.toReverseHexaNotation();
'00000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d'
```

L'en-tête du bloc genesis est particulier car le jeton d'horodatage du bloc précédent est 0.

```
// Header from block 0
var header = {
    version: 1,
    previousToken: '0000000000000000000000000000000000000000000000000000000000000000',
    merkleRootHash: '4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b'
,
    time: new Date ("Sat Jan 03 2009 18:15:05 GMT+0000 (UTC)"),
    bits: 0x1d00ffff,
    nonce: 2083236893
};

> var hashcode = hash ( hash (serializeHeader (header) ) ); // hash twice
> hashcode.toReverseHexaNotation();
'000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f'
```

Annexe 2 : Implémentation des transactions

Sérialisation

Une transaction est un message structuré, qui peut être représenté par :

- un format binaire, utilisé pour le calcul de l'empreinte numérique, la signature et le transfert réseau,
- un format de source arbitraire, qui est un objet JSON dans cet article.

Nous appelons «sérialisation» la transformation d'un format source au format binaire. Le format binaire est le seul format "officiel". En effet, le format source n'est pas spécifié par le protocole. Par conséquent, tout processus applicable à une transaction s'applique au format binaire.

Les règles de conversion pour sérialiser un format source dans un format binaire sont les suivantes :

- la plupart des nombres sont encodés avec la convention Little-Endian sur 32 bits,
- une empreinte numérique est un grand nombre sur 256 bits, codé avec la convention Little-Endian,
- un montant est un entier sur 64 bits, codé avec la convention Little-Endian,
- un tableau commence par le nombre d'entrées, codé sur un octet, suivi de la séquence de toutes ses entrées,
- toute autre donnée commence par la taille de la donnée codée sur un octet.

> La convention Little-Endian ordonne des octets du poids le plus faible vers le poids le plus fort, la convention Big-Endian ordonne des octets du poids le plus fort vers le poids le plus faible. Par exemple, le nombre 1 est sérialisé avec la convention Big-Endian sur 32 bits, en la suite d'octets 00000001 et avec Little-Endian convention sur 32 bits en la suite d'octets 01000000.
>
> Le nombre 1 codé sur 32 bits avec la convention Big-Endian :
>
> ```
> Octet 3 | Octet 2 | Octet 1 | Octet 0
> --------+---------+---------+--------
>      00 |      00 |      00 |      01
> ```
>
> Le nombre 1 encodé sur 32 bits avec la convention Little-Endian :
>
> ```
> Octet 0 | Octet 1 | Octet 2 | Octet 3
> --------+---------+---------+--------
>      01 |      00 |      00 |      00
> ```

La fonction *serializeTransaction* sérialise une transaction représentée avec un objet JSON dans un tampon d'octets.

```
// an arbitrary structure
var transaction = {
    version: 1,
    paymentOrder: "Alice promises to pay the sum of one Bitcoin to Bob"
};

// convert an integer into a buffer of a single byte
var numberToInt8 = function (n) {
    return new Buffer([n]);
}

// convert an integer into a buffer of 4 bytes using
// Little-Endian convention
var numberToInt32LE = function (n) {
    var buffer = new Buffer(4);
    buffer.writeUInt32LE(n,0);
    return buffer;
};

var serializeTransaction = function (transaction) {
    var buffers = [];
    buffers.push (numberToInt32LE(transaction.version));
    buffers.push (numberToInt8(transaction.paymentOrder.length));
    buffers.push (new Buffer(transaction.paymentOrder));
    return Buffer.concat(buffers);
}

> serializeTransaction (transaction).toString('hex');
'0100000033416c6963652070726f6d6973657320746f2070617920746865
2073756d206f66206f6e6520426974636f696e20746f20426f62'
```

Le tampon d'octets résultat est décrit ci-dessous :

```
 |01 00 00 00| version as 32 bits Little-Endian
 |33         | 51 bytes for payment order
 |41 6c 69 63 65 20 70 72 6f 6d 69 73 65 73 20 74|Alice.promises.t|
 |6f 20 70 61 79 20 74 68 65 20 73 75 6d 20 6f 66|o.pay.the.sum.of|
 |20 6f 6e 65 20 42 69 74 63 6f 69 6e 20 74 6f 20|.one.Bitcoin.to.|
 |42 6f 62                                       |Bob|
```

Certains sites Web affichent une notation hexadécimale de messages binaires binaires réels, y compris les transactions : https://blockchain.info/tx/9e9f1efee35b84bf71a4b741c19e1acc6a003f51ef8a7302a3dcd428b99791e4?format=hex

Empreinte numérique d'une transaction

Une fonction de hachage crytographique transforme un message d'entrée en un code numérique, appelé empreinte numérique du message. Cette empreinte reflète le message lui-même. La même fonction de hachage appliquée à un message modifié produira une autre empreinte distincte de la précédente.

SHA-256 (Secured Hash Algorithm) spécifie une méthode de hachage dont le résultat est un grand nombre encodé sur 256 bits. Il est considéré comme sécurisé, car une altération d'un seul bit du message produira une autre empreint numérique distincte, et la tentative de modifier un message pour obtenir une empreinte spécifique nécessiterait trop de tentatives.

Une empreinte d'une transaction est un double hachage du format binaire de la transaction. L'Algorithme SHA-256 est appliqué deux fois, pour des raisons historiques, et pour augmenter la sécurité.

Dans le code ci-dessous, nous définissons la fonction de hachage *sha256* , puis nous définissons la fonction *hachage* pour calculer une empreinte numérique à partir d'une transaction source. Nous devons sérialiser la transaction dans son format binaire, avant d'appliquer la fonction *sha256* deux fois. Nous définissons également la fonction *toReverseHexaNotation* pour afficher l'empreinte numérique, car le résultat est un nombre en utilisant la convention Little-Endian.

```
var crypto = require('crypto');
var sha256 = function(buffer) {
    var f = crypto.createHash("SHA256");
    var h = f.update(buffer);
    return h.digest();
};

var hash = function (encodedTransaction) {
    return sha256 (sha256 (encodedTransaction) );
}

Buffer.prototype.toReverseHexaNotation = function () {
    var hexa = "";
    for (var i = this.length-1; i >= 0; i--) {
        var digits =  this[i].toString(16);
        hexa += ("0" + digits).slice(-2); // Add "0" for single digit
    }
    return hexa;
};

>  hash (serializeTransaction (transaction)).toReverseHexaNotation();
'3e1a51c876a14ea0c09e87d17418910fdc5fb2380af4d040f2b00c2a13906d1c'
```

Clé privée et clé publique

Une signature numérique d'une transaction est le chiffrement de l'empreinte numérique de la transaction calculé avec une clé secrète. Cette clé secrète est appelée la clé privée. La signature de la transaction peut être vérifiée par une clé publique associée. La signature numérique prouve que la transaction n'a pas été modifiée et que la transaction a été émise par le propriétaire de la clé publique associée.

L'algorithme *secp256k1* , basé sur des courbes elliptiques (également connu sous le nom de 'ECDSA': Elliptic Curve Digital Signature Algorithm), est approprié pour la signature numérique. Cet algorithme permet de générer une nouvelle paire de clés de chiffrement: une clé privée et une clé publique. La clé privée est un nombre de 256 bits généré aléatoirement. Et la clé publique est calculée à partir de cette clé privée.

```
var secp256k1 = require('secp256k1');

var generatePrivateKey = function () {
    var privateKey;
    do {
        privateKey = crypto.randomBytes(32);
    } while (!secp256k1.privateKeyVerify(privateKey));
    return privateKey;
}

var alicePrivateKey  = generatePrivateKey();
// get the public key in a compressed format
var alicePublicKey = secp256k1.publicKeyCreate(alicePrivateKey);

> alicePrivateKey.toString('hex');
'27d9b1f6d8567054f1542760ff943d0582e95bd8c1ba08355c02536a5aaac4cc'

> alicePublicKey.toString('hex');
'02c90ad3d07fcc5f92194c7c993ff5b373ce6025b23720f028a2ee1c3aaf97346f'
```

Signer et vérifier

Nous créons une signature numérique d'une transaction avec notre clé privée, et la fonction *hash ()* :

```
var hashcode = hash (serializeTransaction(transaction));
var signature = secp256k1.sign(hashcode, alicePrivateKey).signature;
> signature.toString('hex');
'52990ea17ba23c88af7fc762644e3d1c5338a2e432142dae2b09576d259527aa...
>
```

Nous pouvons vérifier que la signature est valide en fonction de la clé publique et de l'empreinte numérique *hashcode* elle-même :

```
> secp256k1.verify(hashcode, signature, alicePublicKey);
true
```

Tenter d'utiliser une autre clé publique entraîne également un échec de vérification :

```
var bobPrivateKey = generatePrivateKey();
var bobPublicKey = secp256k1.publicKeyCreate(bobPrivateKey);
> secp256k1.verify(hashcode, signature, bobPublicKey);
> false
```

Tenter de modifier une transaction entraîne un échec de vérification :

```
transaction.paymentOrder = "Alice promises to pay the sum of 1000 Bitcoins to Bob";
hashcode = hash(serializeTransaction(transaction));
> secp256k1.verify(hashcode, signature, alicePublicKey);
false
```

Distinguished Encoding Rules

Une signature est une séquence de deux entiers appelés R et S. Pour une transaction, cette séquence est sérialisée en utilisant la convention DER (Distinguished Encoding Rules):

- Une séquence commence par l'octet 0x30 suivi du nombre total d'octets de la séquence et suivi des éléments.
- Un nombre commence par l'octet 0x02 suivi de la taille du nombre en octets et suivi du nombre lui-même en utilisant une convention Big-Endian.

Exemple de signature utilisant DER :

```
var signatureDER =
      "3045" // sequence (0x30) of 69 (0x45) bytes
    + "0221" // fist item 'R' is an integer (0x02) of 33 (0x21 ) bytes
    + "009eb819743dc981250daaaab0ad51e37ba47f7fb4ace61f6a69111850d6f29905"
    + "0220" // second item 'S' is an integer (0x02) of 32 (0x20) bytes
    + "6b6e59e1c002a4e35ba2be4d00366ea0f3e0b14c829907920705bce336ab2945"
```

Pour utiliser une signature DER, le package *secp256k1* fournit la fonction *signatureImport* :

```
var signature = secp256k1.signatureImport(signatureDER); // Decode a DER signature
```

Identifiant de transaction

L'empreinte numérique d'une transaction est nommé *txid* . Cet identifiant de transaction est utilisé pour référencer une transaction.

Nous introduisons une fonction *getTxid* pour obtenir l'identifiant *txid* pour notre format source. Nous ajoutons également la fonction *hexaNotationToInt256LE* pour obtenir le format binaire de l'identifiant *txid*.

```
// calculate a txid for a source format,
// return an hexa notation string for a source format
var getTxid = function (transaction) {
    var encodedTransaction = serializeTransaction (transaction);
    return hash (encodedTransaction).toReverseHexaNotation();
};

var txid = getTxid(transaction);
> txid
'3e1a51c876a14ea0c09e87d17418910fdc5fb2380af4d040f2b00c2a13906d1c'

var hexaNotationToInt256LE = function (hexa)
{
    var bytes = new Array(32);
    for (var i = 0, j = 31, len = hexa.length; i < len; i+=2, j--) {
        bytes[j] = parseInt(hexa[i]+hexa[i+1],16);
    }
    return new Buffer(bytes);
};

> hexaNotationToInt256LE(txid).toString('hex')
'1c6d90132a0cb0f240d0f40a38b25fdc0f911874d1879ec0a04ea176c8511a3e'
```

Grâce à la fonction *getTxid* , nous pouvons créer une table *map* pour stocker certaines transactions au format source et utiliser *txid* comme clé.

```
var dbtrx = {}; // our transaction table
dbtrx[txid] = transaction;
```

Machine virtuelle à pile

Un script est un ensemble d'instructions pour une machine virtuelle à pile. Une machine virtuelle à pile utilise une pile comme mémoire. Toute instruction lit/écrit des opérandes depuis/vers cette pile.

```
var OP_ADD = 0x93;
var OP_DUP = 0x76;
var script = [numberToInt32LE(16), OP_DUP, OP_ADD];
```

Le script ci-dessus effectuera les opérations suivantes :

Instruction Pointer	Instruction	Operations	Stack after operations
0	`1000000`	• Push 16 on the top of te stack	16 ← top
1	`OP_DUP`	• Duplicate the top of the stack	16 ← top 16
2	`OP_ADD`	• Pop a first operand from the top of the stack, • Pop a second operand from the top of the stack, • Add the two operands, • Push the result on top of the stack.	32 ← top

Ce script commence par une pile vide et s'arrête avec une pile contenant la valeur 0x20. Par défaut, à la fin de l'exécution de ce script, la machine virtuelle est arrêtée avec une valeur de retour 'true'. Cependant, certaines opérations de contrôle peuvent mettre fin à l'exécution du script avec une valeur de résultat 'false' lorsqu'elles échouent.

Compilation de script

Chaque instruction est codée avec un seul octet, cet octet est appelé opcode (code d'opération). Une instruction dont l'opcode est inférieur ou égal au nombre 0x75 spécifie le nombre d'octets suivants à pousser sur la pile en tant que données.

Ce langage de script ne nécessite pas d'analyseur et est donc facile à implémenter par tous les nœuds participants.

Le code ci-dessous compile un code source de script:

```
var compileScript = function(program) {
    var buffers = [];
    var bytes = 0;
    for (var i = 0, len = program.length; i < len; i++) {
        var code = program[i];
        var type = typeof(code);
        switch (type) {
            case 'number':
                buffers.push(numberToInt8(code));
                bytes++;
                break;
            case 'object': // already encoded
                operand = code;
                buffers.push(numberToInt8(operand.length));
                buffers.push(operand);
                bytes += operand.length + 1;
                break;
            case 'string': // not yet encoded
                var operand = new Buffer(code, 'hex');
                buffers.push(numberToInt8(operand.length));
                buffers.push(operand);
                bytes += operand.length + 1;
                break;
        }
    }
    buffers.unshift(numberToInt8(bytes));
    return Buffer.concat(buffers);
};

var script = [numberToInt32LE(16), OP_DUP, OP_ADD];
 > compileScript(script).toString("hex")
'0704100000007693'
```

Exécution de script

Nous créons une fonction *runScript* pour interpréter le code compilé précédent. Cet interpréteur sera limité à 2 opcodes: OP_DUP et OP_ADD. Les opérandes des opérations arithmétiques sont des entiers signés sur 32 bits, codés avec la convention Little-Endian.

```
var runScript = function (program, stack)
{
    var operand;
    var operand1;
    var operand2;
    var ip = 0; // instruction pointer

    var last = program[ip++];
    while (ip <= last) {
        var instruction = program[ip++];
        switch (instruction) {
            case OP_DUP:
                operand = stack.pop();
                stack.push(operand);
                stack.push(operand);
                break;
            case OP_ADD:
                operand1 = stack.pop().readInt32LE();
                operand2 = stack.pop().readInt32LE();
                stack.push(numberToInt32LE(operand1 + operand2));
                break;
            default:
                var size = instruction;
                var data  = new Buffer(size);
                program.copy(data, 0, ip, ip+size);
                stack.push(data);
                ip += size;
                break;
        }
    }
    return true;
};

var stack = [];
var script = compileScript ([numberToInt32LE(16), OP_DUP, OP_ADD]);
var result = runScript (script, stack);
> result
true
> tack[0].readInt32LE()
> 32
```

Code pour vérifier un paiement dans le bloc #266632

Le code entier vérifie le premier paiement de la transaction Bitcoin réelle suivante : 9e9f1efee35b84bf71a4b741c19e1acc6a003f51ef8a7302a3dcd428b99791e4.

Vous trouverez la représentation binaire de cette transaction avec l'URL suivante : https://blockchain.info/tx/9e9f1efee35b84bf71a4b741c19e1acc6a003f51ef8a7302a3dcd428b99791e4?format=hex

```
var crypto = require('crypto');
var secp256k1 = require('secp256k1');

var sha256 = function(buffer) {
    var f = crypto.createHash("SHA256");
    var h = f.update(buffer);
    return h.digest();
};

var ripemd160 = function(buffer) {
    var f = crypto.createHash("RIPEMD160");
    var h = f.update(buffer);
    return h.digest();
};


Buffer.prototype.toReverseHexaNotation = function ()
{
    var hexa = "";
    for (var i = this.length-1; i >= 0; i--) {
       var digits =  this[i].toString(16);
        hexa += ("0" + digits).slice(-2); // Add "0" for single digit
    }
    return hexa;
};

var numberToInt8 = function (n) {
    return new Buffer([n]);
};

var numberToInt32LE = function (n) {
    var buffer = new Buffer(4);
    buffer.writeUInt32LE(n,0);
    return buffer;
};

var numberToInt64LE = function (n) {
   var buffer = new Buffer(8);
   buffer.writeUInt32LE(n % 0xFFFFFFFFFFFFFFFF, 0);
   buffer.writeUInt32LE(Math.floor(n / 0xFFFFFFFFFFFFFFFF), 4);
   return buffer;
};

var serializeAmount = function (amount)
{
    return numberToInt64LE(amount * 100000000);
};

var hexaNotationToInt256LE = function (hexa)
{
    var bytes = new Array(32);
```

```
    for (var i = 0, j = 31, len = hexa.length; i < len; i+=2, j--) {
        bytes[j] = parseInt(hexa[i]+hexa[i+1],16);
    }
    return new Buffer(bytes);
};


var      OP_ADD         = 0x93;
var      OP_DUP         = 0x76;
var      OP_HASH160     = 0xa9;
var      OP_EQUALVERIFY = 0x88;
var      OP_CHECKSIG    = 0xac;


var serializeTransaction = function(tr) {
    var buffers = [];
    buffers.push(numberToInt32LE(tr.version));
    buffers.push(serializeInputs(tr.inputs));
    buffers.push(serializeOutputs(tr.outputs));
    buffers.push(numberToInt32LE(tr.lockTime));
    if (tr.hashType)
        buffers.push(numberToInt32LE(Number(tr.hashType)));
    return Buffer.concat(buffers);
};

var serializeInputs = function (inputs)
{
    var buffers = [];

    var inputsSize = inputs.length;
    buffers.push(numberToInt8(inputsSize));

    for (var i = 0; i < inputsSize; i++) {
        var input = inputs[i];

        buffers.push(hexaNotationToInt256LE(input.txid));
        buffers.push(numberToInt32LE(input.index));
        buffers.push(compileScript(input.script));
        buffers.push(numberToInt32LE(0xffffffff));
    }
    return Buffer.concat (buffers);
};

var serializeOutputs = function (outputs)
{
    var buffers = [];

    var outputsSize = outputs.length;
    buffers.push(numberToInt8(outputsSize));
    for (var i = 0; i < outputsSize; i++) {
        var output = outputs[i];
        buffers.push(serializeAmount(output.amount));
        buffers.push(compileScript(output.script));
```

```
    }
    return Buffer.concat (buffers);
};

var compileScript = function(program)
{
    var buffers = [];
    var bytes = 0;
    for (var i = 0, len = program.length; i < len; i++) {
        var code = program[i];
        var type = typeof(code);
        switch (type) {
            case 'number':
                buffers.push(numberToInt8(code));
                bytes++;
                break;
            case 'string':
                var operand = new Buffer(code, 'hex');
                buffers.push(numberToInt8(operand.length));
                buffers.push(operand);
                bytes += operand.length + 1
                break;
        }
    }
    buffers.unshift(numberToInt8(bytes));
    return Buffer.concat(buffers);
};

// A simple virtual machine to run a decoded P2SH (Pay to Script Hash) scripts



var runScript = function (program, stack, currentTransaction, currentInputIndex)
{
    var operand;
    var operand1;
    var operand2;
    var ip = 0; // instruction pointer
    var last = program[ip++];
    while (ip <= last) {
        var instruction = program[ip++];

        switch (instruction) {
            case OP_DUP:
                operand = stack.pop();
                stack.push(operand);
                stack.push(operand);
                break;
            case OP_ADD:
                operand1 = stack.pop().readInt32LE();
                operand2 = stack.pop().readInt32LE();
                stack.push(numberToInt32LE(operand1 + operand2));
                break;
```

```
            case  OP_HASH160:
                operand = stack.pop();
                stack.push(ripemd160(sha256(operand)));
                break;

            case  OP_EQUALVERIFY:
                operand1 = stack.pop();
                operand2 = stack.pop();
                if (! operand1.compare(operand2) == 0) return false;
                break;

            case  OP_CHECKSIG:
                operand1 = stack.pop();
                operand2 = stack.pop();

                // operand 1 is Public Key
                var publicKey = operand1;

                // operand 2 contains hashType
                var hashType = operand2[operand2.length-1]; //get last byte of signatu
re

                // operand 2 contains DER Signature
                var signatureDER = operand2.slice(0,-1);
                var signature = secp256k1.signatureImport(signatureDER); // Decode a s
ignature in DER format

                // recover signed transaction and hash of this transaction
                var copy = copyForSignature(currentTransaction, currentInputIndex, has
hType);
                var buffer = serializeTransaction(copy);
                var hashcode = sha256 (sha256 (buffer));

                // Check signature
                if (! secp256k1.verify(hashcode, signature, publicKey)) return false;
                break;
            default:
                var size = instruction;
                var data  = new Buffer(size);
                program.copy(data, 0, ip, size+ip);
                stack.push(data);
                ip += size;
                break;
        }
    }
    return true;
};

var SIGHASH_ALL    = "01";
var SIGHASH_NONE = "02";
var SIGHASH_SINGLE = "03";
var SIGHASH_ANYONECANPAY = "80";
```

```
// We create a previous transaction with an output
// We skip other data that are not required for validation

var previousTransaction =
{
    version: 1,
    inputs: {}, // missing actual data here
    outputs: [
        {}, // missing output[0]
        {
            amount: 0.09212969,
            script: [
                OP_DUP,
                OP_HASH160,
                '4586dd621917a93058ee904db1b7a43bfc05910a',
                OP_EQUALVERIFY,
                OP_CHECKSIG
            ]
        }
    ],
    lockTime: 0
};

var transaction = {
   version: 1,
   inputs: [
        {
            txid: "14e5c51d3bc1cf0d29f2457d61fbf8d6567883e0711f9877795783d2105b50c9",
            index: 1,

            script: [
                  "3045"
                + "0221"
                + "009eb819743dc981250daaaab0ad51e37ba47f7fb4ace61f6a69111850d6f29905"
                + "0220"
                + "6b6e59e1c002a4e35ba2be4d00366ea0f3e0b14c829907920705bce336ab2945" /
/ signature
                +  SIGHASH_ALL,    // hashtype

                "0275e9b1369179c24935337d597a06df0e388b53e8ac3f10ee426431d1a90c1b6e" /
/ Public Key
            ]
        },
        {
            txid: "5b7aeedc2e82c9646408ce0588d9f98d2107062e9291af0e9e6fa372b0d7d1fb",
            index: 1,

            script: [
                  "3045"
                + "0220"
                + "35a9e444883acaaae166d2ee1389272424ec7885f4210aaf118fee58b5683445"
                + "0221"
                + "00e40624a0df47943aa5ee63d8997dd36c5da44409ccc4dafcbfabc96a020d971c"
```

```
// signature
                + SIGHASH_ALL,    // hashtype

                "033b18e24fb031dae396297516a54f3e46cc9902adfd1b8edea0d6a01dab0e027d" /
/ Public Key
            ]
        }
   ],
   outputs: [
        {
            amount: 0.05580569,
            script: [
                    OP_DUP,
                    OP_HASH160,
                    '4753945f3b34d6ca3fedcf41bf499c13d20bfec4',
                    OP_EQUALVERIFY,
                    OP_CHECKSIG
            ],
        },
        {
            amount: 0.1,
            script: [
                    OP_DUP,
                    OP_HASH160,
                    '81a9e7d0ab008005d36c61563a178ad20a3a5224',
                    OP_EQUALVERIFY,
                    OP_CHECKSIG
            ],
        }
   ],
   lockTime: 0
};


var dbtx = {};
dbtx["14e5c51d3bc1cf0d29f2457d61fbf8d6567883e0711f9877795783d2105b50c9"] = previousTra
nsaction;
dbtx["9e9f1efee35b84bf71a4b741c19e1acc6a003f51ef8a7302a3dcd428b99791e4"] = transaction
;

var copyForSignature = function(transaction, inputIndex, hashType)
{
    var copy = Object.assign({}, transaction);

    var inputs = copy.inputs;
    for (var i = 0, len = inputs.length; i < len; i++) {
        inputs[i].script = []; // reset script to nothing
    }

    var currentInput = inputs[inputIndex];

    var previousTransaction =  dbtx[currentInput.txid];
    var previousOutput =previousTransaction.outputs[currentInput.index];
```

```
    currentInput.script = previousOutput.script;

    copy.hashType = hashType;
    return copy;
};

var validateInput = function (transaction, inputIndex)
{
    var stack = [];

    var input = transaction.inputs[inputIndex];
    var previousTransaction =  dbtx[input.txid];
    var previousOutput =previousTransaction.outputs[input.index];

    var program1 = compileScript(input.script);
    var program2 = compileScript(previousOutput.script);

    var result = runScript (program1, stack, transaction, inputIndex);
    if (result) result = runScript (program2, stack, transaction, inputIndex);
    console.log(stack);
    return result;
};

var currentTransaction = transaction;
var currentInputIndex  = 0;
console.log(validateInput(currentTransaction, currentInputIndex));
```