ETHEREUM : UN REGISTRE DE TRANSACTIONS GÉNÉRALISTE, SÉCURISÉ ET DÉCENTRALISÉ

RÉVISION EIP-150

DR. GAVIN WOOD
FONDATEUR, ETHEREUM & ETHCORE
GAVIN@ETHCORE.IO

Abstract. Le paradigme de la blockchain, lorsqu'il est couplé à un mécanisme de transaction sécurisé par la cryptographie, a prouvé son utilité non seulement avec Bitcoin mais aussi dans de nombreux projets. Chacun de ces projets peut être vu comme une simple application basée sur une ressource de calcul décentralisée mais unitaire. Nous pouvons appeler ce paradigme une machine transactionnelle unique à état partagé.

Ethereum implémente ce paradigme de façon généraliste. De plus, Ethereum permet une multiplicité de telles ressources, chacune possédant un état distinct et un code fonctionnel, mais néanmoins susceptibles d'interagir via un framework d'échange de messages. Nous discuterons de sa conception, des problèmes d'implémentation, des opportunités que cela ouvre et des obstacles que nous anticipons.

1. Introduction

Internet étant présent dans les plupart des endroits de la planète, la transmission de l'information est globalement devenue incroyablement bon marché. Les innovations technologiques telles que Bitcoin ont démontré, par le pouvoir d'un mécanisme de consensus et de respect volontaire d'un contrat social, qu'il était possible d'utiliser Internet pour créer un système décentralisé de transfert de valeur, partagé à travers le monde et quasiment gratuit à utiliser. Ce système peut être décrit comme une version très spécifique d'une machine a états basée sur des transactions et sécurisée par la cryptographie. Des systèmes de suivi tels que Namecoin ont adapté ce système original d' « application monétaire » à d'autres applications quoiqu'elles aient été plutôt simples.

Ethereum est un projet qui cherche à créer une version généraliste de cette technologie ; technologie sur laquelle tous les concepts de machine transactionnelle à états pourraient être construits. De plus Ethereum a pour but de fournir au développeur final un système totalement intégré permettant de construire des logiciels fondés sur un paradigme de calcul jusque-là inexploré : un framework de confiance pour la messagerie, le calcul et les objets.

1.1. Principes directeurs.

Ce projet poursuit de nombreux objectifs ; l'un des principaux consiste à faciliter les transactions entre deux individus consentants qui n'auraient sans cela aucun moyen de se faire confiance l'un à l'autre. Cela peut être dû à une séparation géographique, une difficulté d'interaction, ou peut-être à l'incompatibilité, l'incompétence, la réticence, le coût, l'incertitude, la gêne ou encore la corruption des systèmes légaux existants. En définissant un système de changement d'état à travers un langage riche et non ambigu, et de plus en architecturant un système tel que nous pouvons raisonnablement en attendre qu'un accord soit assuré de manière autonome, nous pouvons fournir un moyen à cette fin.

Les opérations dans le système proposé pourraient avoir beaucoup d'attributs qui souvent n'existent pas dans le monde réel. L'incorruptibilité de l'appréciation, souvent difficile à trouver, est naturelle lorsqu'elle vient d'un interpréteur algorithmique désintéressé. La transparence, ou capacité à voir exactement comment un état ou un jugement s'est produit au travers des journaux et des règles d'une transaction ou des codes d'instructions, n'est jamais parfaitement atteinte dans les systèmes basés sur l'humain, parce que le langage naturel est nécessairement vague, que de l'information manque souvent, et que des préjudices antérieurs sont difficiles à démêler.

De manière générale, je souhaite fournir un système dans lequel les utilisateurs peuvent être sûrs que peu importe avec quels autres individus, systèmes ou organisations ils interagissent, ils peuvent le faire en ayant une confiance absolue dans les résultats possibles et les règles guidant l'obtention de ces résultats.

1.2. Travaux antérieurs.

Buterin [2013a] a proposé initialement le noyau de ce travail fin novembre 2013. Bien qu'elle ait maintenant évolué sur de nombreux points, la fonctionnalité clé d'une blockchain possédant un langage Turing-complet et une capacité illimitée et fonctionnelle de stockage inter-transactions demeure inchangée.

Dwork and Naor [1992] a fourni le premier travail d'usage de la preuve cryptographique d'effort de calcul (« proof-of-work, ou preuve de travail ») ayant pour but de transmettre un signal de valeur à travers Internet. Le signal de valeur était utilisé ici comme un mécanisme de dissuasion de spam plutôt que comme une forme de monnaie, mais il a démontré de manière décisive le potentiel d'un canal de données basique à porter un *signal économique fort*, autorisant un récepteur à faire une vérification physique sans avoir à s'en remettre à une forme de *confiance*. Back [2002] a plus tard produit un système dans la même veine.

Le premier exemple d'utilisation de la preuve de travail en tant que signal économique fort pour sécuriser une monnaie a été utilisé par Vishnumurthy et al. [2003]. Dans cet exemple, le jeton était utilisé pour contrôler l'échange de fichiers en pair à pair et permettait aux « consommateurs » de procéder à des micro-paiements aux « fournisseurs » pour leurs services. Le modèle de sécurité apporté par la preuve de travail était augmenté par des

signatures numériques et un registre dans le but de protéger l'historique des enregistrements contre la corruption, et contre des actes malicieux tels que la falsification des paiements ou les plaintes abusives. Cinq ans plus tard, Nakamoto [2008] introduisit un autre jeton de valeur basé sur une preuve de travail sécurisée avec une portée plus large. Le fruit de ce projet, Bitcoin, est devenu le premier registre global décentralisé de transactions à être largement adopté.

D'autres projets construits à partir de Bitcoin ont réussi ; les altcoins, ou monnaies alternatives, ont introduit de nombreuses autres monnaies en modifiant le protocole. Quelques-unes des plus connues sont Litecoin et Primecoin, citées par Sprankel [2013]. D'autres projets ont su prendre la valeur essentielle contenue dans le mécanisme du protocole et la réappliquer ; Aron [2012] décrit, par exemple, le projet Namecoin qui a pour but de fournir un système de résolution de nom décentralisé.

D'autres projets visent toujours à construire par-dessus le réseau Bitcoin lui-même, tirant profit de l'importante valeur placée dans le système et de la très grande puissance de calcul destinée au mécanisme de consensus. Le projet Mastercoin, proposé initialement par Willett [2013], vise à construire un protocole plus riche, impliquant beaucoup de fonctionnalités de haut niveau par-dessus le protocole Bitcoin à travers l'utilisation d'un grand nombre de parties auxiliaires au protocole de base. Le projet Colored Coins, proposé par Rosenfeld [2012], a une stratégie similaire mais plus simple, agrémentant les règles d'une transaction afin de casser la fongibilité de la monnaie de base Bitcoin et autoriser la création et le suivi des jetons à travers un « chroma-wallet » une surcouche logicielle fonctionnant avec le protocole.

Des travaux supplémentaires ont été effectués avec pour idée d'écarter les fondements de la décentralisation ; Ripple, décrit par Boutellier and Heinzen [2014], a cherché à créer un système fédéré pour l'échange de devises constituant effectivement un nouveau système de compensation financière. Cela a démontré que de gros gains d'efficience peuvent être faits si le principe de la décentralisation est rejeté.

Antérieurement, un travail sur les smart contracts a été effectué par Szabo [1997] et Miller [1997]. Autour des années 90 il est devenu clair que la résolution algorithmique des accords entre humains pouvait devenir une force significative dans la coopération. Bien qu'aucun système spécifique n'ait été proposé pour implémenter un tel système, il a été présumé que le futur du droit serait profondément affecté par ce genre de résolution. Dans cette optique, Ethereum peut être vu comme une implémentation généraliste d'un tel système de *droit cryptographique*.

2. Le paradigme de la blockchain

Ethereum, pris dans son ensemble, peut être vu comme une machine à état basée sur des transactions : nous commençons avec un état originel et nous exécutons des transactions de manière incrémentale pour le transformer en un état final. C'est cet état final que nous acceptons comme la « version » canonique du monde d'Ethereum. L'état peut inclure des informations telles que les soldes de comptes, la réputation, des accords de confiance, des données portant sur l'information du monde physique ; en résumé, tout ce qui peut actuellement être représenté par un ordinateur est admissible. Les transactions représentent ainsi une passerelle valide entre deux états ; l'aspect « valide » est important—il existe beaucoup plus de changements d'état invalides que de changements d'état valides. Les changements d'état invalides pourraient par exemple être la réduction du solde d'un compte sans compensation par ailleurs d'un montant équivalent. Un changement d'état valide est un changement produit par le truchement d'une transaction. Formellement :

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{1}$$

où Υ est la fonction de transition d'état d'Ethereum. Dans Ethereum, Υ combiné avec $\boldsymbol{\sigma}$ sont considérablement plus puissants que tous les systèmes comparables existants ; Υ autorise les composants à effectuer des calculs quelconques alors que $\boldsymbol{\sigma}$ autorise les composants à stocker des états quelconques entre les transactions.

Les transactions sont regroupées dans des blocs ; les blocs sont chaînés les uns aux autres en utilisant une empreinte (ou *hash* cryptographique) comme technique de référencement. Les blocs fonctionnent comme un journal, enregistrant une série de transactions ainsi que le bloc précédant et un identifiant pour l'état final (l'état final n'est cependant pas stocké lui-même—cela serait beaucoup trop volumineux). Ils ponctuent également la série de transactions avec une incitation pour les nœuds à *miner*.

Miner est le processus consistant à dédier de l'effort (du travail) pour soutenir une série de transactions (un bloc) contre n'importe quel autre bloc concurrent. Cela est mis en œuvre grâce à une preuve cryptographique sécurisée. Ce schéma est connu comme preuve de travail (proof-of-work), et est étudié en détail dans la section 11.5.

Formellement, on développe ainsi :

$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \tag{2}$$

$$B \equiv (..., (T_0, T_1, ...)) \tag{3}$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) \tag{4}$$

Où Ω est la fonction de changement d'état finalisant le bloc (une fonction qui récompense un ensemble désigné) ; B est ce bloc, qui inclut une série de transactions parmi d'autres éléments ; et Π est la fonction de changement d'état au niveau du bloc.

Ceci est la base du paradigme de la blockchain, un modèle qui forme la colonne vertébrale non seulement d'Ethereum, mais aussi de tous les systèmes de transactions décentralisés fondés sur le consensus à l'heure actuelle.

2.1. Valeur.

Dans le but d'inciter à l'affectation de puissance de calcul à l'intérieur du réseau, il y a besoin de convenir d'une méthode pour transmettre la valeur. Pour régler ce problème, Ethereum a une monnaie intrinsèque, l'Ether, aussi connue comme ETH, et parfois en référence à l'Anglais Ancien Ð (prononcé Eth). La plus petite sous-unité de l'Ether, et celle dans laquelle toutes les valeurs entières de la monnaie sont comptées est le Wei. Un Ether représente 10^{18} Wei. Il existe d'autres unités dérivées de l'Ether :

Ordre de grandeur	Nom
10^{0}	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

Tout au long du présent document, toute référence à une valeur ou à l'Ether, comme monnaie, balance ou paiement, sera exprimée en Wei.

2.2. **Quel historique ?** Puisque le système est décentralisé et que toutes les parties ont une opportunité de créer un nouveau bloc sur un bloc préexistant, la structure résultante est nécessairement un arbre de blocs. Il est nécessaire de disposer d'un schéma de passage consensuel dans cette structure, c'est-à-dire un chemin depuis la racine (le bloc originel) jusqu'à la feuille (le bloc contenant les transactions les plus récentes) qui fasse consensus à travers cette structure d'arbre connue sous le nom de blockchain. Si jamais il y a un désaccord entre les nœuds sur quel des chemins de l'arbre des blocs constitue la « meilleure » blockchain, alors il y a fourche ou scission (*fork*).

Cela voudrait dire que passé un point donné dans le temps (bloc), plusieurs états du système peuvent coexister : certains nœuds croient qu'un bloc contient les transactions canoniques alors que d'autres nœuds croient canonique un autre bloc, celui-ci contenant potentiellement des transactions radicalement différentes ou incompatibles. Ceci est à éviter à tout prix, car l'incertitude qui s'ensuivrait pourrait tuer la confiance dans le système entier.

Le schéma que nous utilisons pour générer un consensus est une version simplifiée du protocole GHOST, introduit par Sompolinsky and Zohar [2013]. Ce processus est décrit en détail dans la section 10.

3. Conventions

J'utilise un certain nombre de conventions typographiques pour les notations formelles, quelques-unes étant un peu particulières au document présent :

Les deux ensembles de haut niveau, hautement structurés, des valeurs d'état, sont notés en minuscules grecques en gras. Ces valeurs relèvent de la catégorie de l'état du monde noté $\boldsymbol{\sigma}$ (ou une variante de celui-ci) ou de celle de l'état de la machine, $\boldsymbol{\mu}$.

Les fonctions opérant sur ces valeurs hautement structurées sont notés en lettre grecque majuscule, par exemple Υ, la fonction de changement d'état d'Ethereum.

Pour la plupart des fonctions, une lettre majuscule est utilisée, par exemple C, la fonction de coût général. Elles peuvent être indicées pour dénoter des variantes spécifiées, par exemple C_{SSTORE}, la fonction de coût pour l'opération SSTORE. Pour certaines fonctions spécialisées et éventuellement définies de manière externe, je peux utiliser une typographie particulière, par exemple la fonction de hachage Keccak-256 (la gagnante du concours SHA-3) est dénotée `KEC` (et généralement référencée comme simple Keccak). `KEC512` fait référence à une fonction de hachage Keccak 512.

Les n-uplets sont typiquement notés en lettres majuscules, par exemple T est utilisé pour noter une transaction Ethereum. Ce symbole peut, s'il est défini en conséquence, être indicé pour faire référence à un composant individuel, e.g. T_n, dénote le nonce de ladite transaction. la forme de l'indice est utilisée pour dénoter son type ; par exemple des indices en majuscule font référence à des n-uplets avec des composants indiçables.

Les scalaires et les séquences d'octets de taille fixe (analogue, les tableaux), sont notés avec une lettre normale minuscule, par exemple n est utilisée pour représenter le nonce de telle transaction. Ceux qui ont une signification particulière peuvent être grecs, par exemple δ, le nombre d'éléments requis dans la pile pour une opération donnée.

Des séquences de longueur arbitraire sont typiquement notées en lettre minuscule et en gras, par exemple $\mathbf{o}$ est utilisé pour représenter la séquence d'octets en sortie d'un appel de message. Pour les valeurs particulièrement importantes, une lettre en majuscule et en gras peut être utilisée.

Tout au long du document nous supposons que les valeurs scalaires sont des entiers positifs, et qu'elles appartiennent à l'ensemble $\mathbb{P}$. L'ensemble de toutes les séquences d'octets est $\mathbb{B}$, formellement défini dans l'annexe B. Si un tel ensemble de séquences est restreint à celles d'une longueur particulière, cela est dénoté avec un indice. C'est ainsi que l'ensemble de toutes les séquences d'octets de longueur 32 est nommé $\mathbb{B}_{32}$ et l'ensemble de tous les entiers positifs plus petits que 2^{256} est nommé $\mathbb{P}_{256}$. Cela formalisé dans la section 4.3.

Les crochets sont utilisés pour indexer et référencer les composants individuels ou les sous-séquences de séquences, par exemple $\boldsymbol{\mu}_{\mathbf{s}}[0]$ représente le premier objet dans la pile de la machine. Pour les sous-séquences, les ellipses sont utilisées pour spécifier la tranche attendue en incluant les éléments au deux limites, par exemple $\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ représente les 32 premiers objets de la mémoire de la machine.

Dans le cas de l'état global $\boldsymbol{\sigma}$, qui est une séquence de comptes, eux-mêmes des n-uplets, les crochets sont utilisés pour référencer un compte individuel.

En considérant les variantes des valeurs existantes, je suis la règle selon laquelle, au sein d'un périmètre donné comme définition, si nous supposons que la valeur d'« input » non modifiée est désignée par le signe $\square$, alors la valeur modifiée et utilisable est représentée comme $\square'$, et les valeurs intermédiaires sont alors $\square^*$, $\square^{**}$, etc. Dans des cas vraiment très particuliers, dans le but de maximiser la lisibilité et seulement si le sens est non ambigu, je peux utiliser des indices alphanumériques pour représenter des valeurs intermédiaires.

En considérant l'utilisation de fonctions existantes, étant donnée une fonction f, la fonction f^* représente une version similaire élément par élément de la mise en correspondance (mapping) des éléments de la fonction plutôt qu'entre les séquences d'élements. Cela est précisément décrit dans la section 4.3.

Je définis un nombre de fonctions utiles au long de ce document. L'une des plus communes est ℓ, qui évalue le dernier objet de la séquence donnée :

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1] \tag{5}$$

4. Blocs, état et transactions

Ayant introduit les concepts de base d'Ethereum, nous allons discuter de la signification d'une transaction, d'un bloc et de l'état de manière plus détaillée.

4.1. **L'état du monde.** L'état du monde (*l'état*), est une mise en correspondance (*mapping*) entre les adresses (identifiants de 160-bit) et l'état des comptes (une structure de données sérialisée comme RLP (Recursive Length Prefix - Préfixe de Longueur Récursive), voir annexe B). Bien que non stocké sur la blockchain, on suppose ce mapping maintenu par l'implémentation dans un arbre radix de Merkle modifié (*trie*, voir annexe D). Le trie nécessite une base de données en arrière-plan qui maintienne un mapping de tableau d'octets à tableau d'octets; nous nommons cette base de données sous-jacente la base de données d'état. Elle a un grand nombre d'avantages; premièrement le nœud racine de cette structure est cryptographiquement dépendant de toutes les données internes et son empreinte peut être utilisée comme une identité sécurisée pour l'état du système entier. Deuxièmement, étant une structure de données immutable, elle permet à n'importe quel état précédent (dont l'empreinte racine est connue) d'être appelé à nouveau en altérant simplement l'empreinte racine en fonction. Comme nous stockons toutes ces empreintes racines dans la blockchain, nous sommes capables de revenir aux anciens états de manière triviale.

L'état d'un compte comprend les quatre champs suivants:

nonce: Une valeur scalaire égale au nombre de transactions envoyées depuis cette adresse, ou, dans le cas d'un compte contenant du code, le nombre de contrats créés par ce compte. Pour les comptes ou les adresses a dans l'état $\boldsymbol{\sigma}$, cela sera représenté par $\boldsymbol{\sigma}[a]_n$.

balance: Une valeur scalaire égale au nombre de Wei possédés par cette adresse. Représentée par $\boldsymbol{\sigma}[a]_b$.

storageRoot: Une empreinte de 256-bit du nœud racine d'un arbre radix de Merkle qui encode le stockage des contenus du compte (un mapping entre des valeurs entières de 256 bits), encodés dans le trie comme une correspondance entre l'empreinte Keccak 256 bits des clés entières de 256 bits et les valeurs entières encodées par RLP de 256 bits. L'empreinte est notée $\boldsymbol{\sigma}[a]_s$.

codeHash: L'empreinte du code EVM du compte—c'est le code qui est exécuté lorsque cette adresse reçoit un appel de message; il est immuable et, contrairement à tous les autres champs, ne peut donc pas être changé après construction. Tous ces fragments de code sont contenus dans la base de données d'état sous leur empreinte correspondante pour les récupérer plus tard. Cette empreinte est notée $\boldsymbol{\sigma}[a]_c$, ce qui fait que le code peut être représenté par $\mathbf{b}$, sachant que $\texttt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_c$.

Comme je souhaite typiquement faire référence, non pas à l'empreinte racine du trie mais à l'ensemble des paires clé/valeur stockées à l'intérieur, je définis une équivalence pratique:

$$\texttt{TRIE}\big(L_I^*(\boldsymbol{\sigma}[a]_\mathbf{s})\big) \equiv \boldsymbol{\sigma}[a]_s \tag{6}$$

La fonction d'effondrement ordinale pour l'ensemble de paires de clé/valeur dans le trie, L_I^*, est définie comme la transformation élément par élément de la fonction de base L_I, définie par:

$$L_I\big((k, v)\big) \equiv \big(\texttt{KEC}(k), \texttt{RLP}(v)\big) \tag{7}$$

où:

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{P} \tag{8}$$

Il faut comprendre ici que $\boldsymbol{\sigma}[a]_\mathbf{s}$ n'est pas un membre «physique» du compte et ne contribue pas a sa sérialisation ultérieure.

Si le champ **codeHash** est l'empreinte Keccak-256 de la chaîne (de caractères) vide, i.e. $\boldsymbol{\sigma}[a]_c = \texttt{KEC}(())$, alors le nœud représente un simple compte, parfois dénommé compte «non-contrat».

Ainsi nous pouvons définir une fonction d'effondrement ordinale de l'état du monde L_S:

$$L_S(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{9}$$

où

$$p(a) \equiv \big(\texttt{KEC}(a), \texttt{RLP}\big((\boldsymbol{\sigma}[a]_n, \boldsymbol{\sigma}[a]_b, \boldsymbol{\sigma}[a]_s, \boldsymbol{\sigma}[a]_c)\big)\big) \tag{10}$$

Cette fonction, L_S, est utilisée tout au long de la fonction de trie pour fournir une identité courte (hash) à l'état du monde. Nous supposons:

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \ \vee \ (a \in \mathbb{B}_{20} \ \wedge \ v(\boldsymbol{\sigma}[a])) \tag{11}$$

où v est la fonction de validité de compte:

$$v(x) \equiv x_n \in \mathbb{P}_{256} \wedge x_b \in \mathbb{P}_{256} \wedge x_s \in \mathbb{B}_{32} \wedge x_c \in \mathbb{B}_{32} \tag{12}$$

4.2. **La Transaction.** Une transaction (T) est une instruction de signature cryptographique unique construite par un acteur externe au périmètre d'Ethereum. Alors que nous supposons que l'ultime acteur externe sera humain par nature, des outils logiciels seront utilisés pour sa construction et sa propagation[1]. Il y a deux types de transactions: celles qui entraînent des appels de messages et celles qui entraînent la création de nouveaux comptes auxquels est associé du code (connues de manière informelle sous le nom de «créations de contrat»). Ces deux types de transaction spécifient un certain nombre de champs communs:

nonce: Une valeur scalaire égale au nombre de transactions émises par l'envoyeur T_n.

gasPrice: Une valeur scalaire égale au nombre de Wei à payer par unité de *gas* pour tous les coûts de calcul engagés comme résultat de l'exécution de cette transaction; T_p.

gasLimit: Une valeur scalaire égale au montant maximum de gaz qui peut être utilisé en exécutant cette transaction. Cela est payé en amont, avant que tout calcul soit fait et ne peut pas être augmenté plus tard; T_g.

to: L'adresse de 160 bits du destinataire de l'appel de message ou, pour une transaction de création de contrat , $\varnothing$, utilisé ici pour noter le seul membre de $\mathbb{B}_0$; formellement T_t.

[1] En particulier, de tels «outils» pourraient en fin de compte devenir tellement disjoints de leur relation causale à une création humaine—ou bien les humains pourraient y devenir si indifférents—qu'ils pourraient se comporter comme des agents autonomes. Par exemple les contrats pourraient offrir des primes aux humains pour avoir envoyé des transactions pour initier leur exécution.

value: Une valeur scalaire égale au nombre de Wei à transférer au destinataire de l'appel de message ou, dans le cas d'une création de contrat, comme une dotation pour le nouveau compte à créer ; T_v.

v, r, s: Les valeurs correspondantes à la signature de la transaction et utilisées pour déterminer l'emetteur de la transaction ; formellement T_w, T_r and T_s. Cela est détaillé dans l'annexe F.

De plus, une transaction de création de contrat contient :

init: Un tableau d'octets de taille illimitée contenant le code EVM pour la procédure d'initialisation du compte, $T_{\mathbf{i}}$.

init est un fragment de code EVM ; il retourne le **body**, un second fragment de code qui s'exécute à chaque fois que le compte reçoit un appel de message (soit au travers d'une transaction, soit par l'exécution interne de code). **init** est exécuté une seule fois à la création du compte et reste écarté immédiatement après.

En revanche, une transaction d'appel de message contient :

data: Un tableau d'octets illimité spécifiant la donnée entrante de l'appel de message, $T_{\mathbf{d}}$.

L'annexe F décrit la fonction, S, qui fait correspondre les transactions au destinataire, et s'exécute à travers l'ECDSA de la courbe SECP-256k1, utilisant l'empreinte de la transaction (excepté les trois derniers champs de signature) en tant que donnée à signer. Pour le moment nous nous assurons seulement que l'émetteur d'une transaction donnée peut être représenté par $S(T)$.

(13)
$$L_T(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{i}}, T_w, T_r, T_s) & \text{si } T_t = \varnothing \\ (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{d}}, T_w, T_r, T_s) & \text{sinon} \end{cases}$$

Ici, nous supposons que tous les composants sont interprétés par le RLP comme des valeurs entières, avec l'exception des tableaux d'octets de longueur arbitraire $T_{\mathbf{i}}$ et $T_{\mathbf{d}}$.

(14)
$$\begin{array}{lllllll} T_n \in \mathbb{P}_{256} & \wedge & T_v \in \mathbb{P}_{256} & \wedge & T_p \in \mathbb{P}_{256} & \wedge \\ T_g \in \mathbb{P}_{256} & \wedge & T_w \in \mathbb{P}_5 & \wedge & T_r \in \mathbb{P}_{256} & \wedge \\ T_s \in \mathbb{P}_{256} & \wedge & T_{\mathbf{d}} \in \mathbb{B} & \wedge & T_{\mathbf{i}} \in \mathbb{B} & \end{array}$$

où

(15)
$$\mathbb{P}_n = \{P : P \in \mathbb{P} \wedge P < 2^n\}$$

L'empreinte de l'adresse $T_{\mathbf{t}}$ est légèrement différente : c'est soit l'empreinte sur 20 octets d'une adresse, soit, dans le cas d'une création de contrat (et ainsi formellement égale à $\varnothing$), c'est la série RLP d'octets vides et ainsi le membre de $\mathbb{B}_0$:

(16)
$$T_t \in \begin{cases} \mathbb{B}_{20} & \text{si} \quad T_t \neq \varnothing \\ \mathbb{B}_0 & \text{sinon} \end{cases}$$

4.3. **Le Bloc.** Le bloc dans Ethereum est l'ensemble des informations pertinentes (connu comme *l'en-tête* du bloc), H, ainsi que l'information correspondant aux transactions incluses, $\mathbf{T}$, et un ensemble d'autres en-têtes $\mathbf{U}$ qui sont connus pour avoir un parent égal au parent du parent du bloc actuel (ce genre de bloc est connu en tant qu'oncle ou *ommer*[2]). L'en-tête du bloc contient plusieurs parties d'information :

parentHash: L'empreinte Keccak de 256 bits de l'en-tête du bloc parent, dans son entièreté ; H_p.

ommersHash: L'empreinte Keccak de 256 bits de la section de liste des oncles de ce bloc ; H_o.

beneficiary: L'addresse de 160 bits à laquelle seront envoyés tous les frais collectés par le minage réussi de ce bloc ; H_c.

stateRoot: L'empreinte Keccak de 256 bits du noeud racine du trie d'état, après que toutes les transactions ont été exécutées et que les finalisations ont été appliquées ; H_r.

transactionsRoot: L'empreinte Keccak de 256 bits du noeud racine de la structure du trie, peuplé avec chaque transaction dans la section de la liste de transactions du bloc ; H_t.

receiptsRoot: L'empreinte Keccak de 256 bits du noeud racine de la structure de trie, peuplé avec les reçus de chaque transaction dans la section de la liste de transactions du bloc ; H_e.

logsBloom: Le filtre de Bloom composé de l'information indexable (l'adresse du logger et les sujets de log) contenu dans chaque entrée de log depuis le reçu de chaque transaction dans la liste de transactions ; H_b.

difficulty: Une valeur scalaire correspondant au niveau de difficulté de ce bloc. Il peut être calculé à partir de la difficulté précédente et du timestamp ; formellement H_d.

number: Une valeur scalaire égale au nombre de blocs ancêtre. Le bloc origine a un number de zéro ; H_i.

gasLimit: Une valeur scalaire égale à la limite courante d'usage de gaz par bloc ; H_l.

gasUsed: Une valeur scalaire égale au gaz total utilisé par les transactions dans ce bloc ; H_g.

timestamp: Une valeur scalaire égale à un timestamp Unix raisonnable (time()) pour la naissance de ce bloc ; H_s.

extraData: Un tableau d'octets de longueur arbitraire contenant des données relatives à ce bloc. Il doit être de 32 octets ou moins ; H_x.

mixHash: Une empreinte de 256 bits qui prouve, combiné avec le nonce, qu'une quantité suffisante de calcul informatique a été consacré à ce bloc ; H_m.

nonce: Une empreinte de 64 bits qui prouve, combiné avec le mixHash, qu'une quantité suffisante de calcul informatique a été consacrée à ce bloc ; H_n.

Les deux derniers composants du bloc sont simplement une liste d'en-têtes de blocs oncles (du même format que ci-dessus) et une liste des transactions. Formellement, on peut noter le bloc B :

(17)
$$B \equiv (B_H, B_{\mathbf{T}}, B_{\mathbf{U}})$$

[2]Dans l'original, *ommer* est le terme, neutre au niveau du genre, le plus répandu pour représenter « la fraternité du parent » ; voir http://nonbinary.org/wiki/Gender_neutral_language#Family_Terms

4.3.1. *Reçu de transaction.* De façon à pouvoir stocker l'information concernant une transaction, pour laquelle il peut être utile de fournir une preuve à divulgation nulle de connaissance, ou l'indexer à des fins de recherche, nous encodons un reçu (*receipt*) de chaque transaction contenant des informations à propos de son exécution. Chaque reçu, noté $B_{\mathbf{R}}[i]$ pour la $i$ème transaction, est placé dans un trie indexé et la racine est encodée dans son en-tête comme H_e.

Le reçu de transaction est un n-uplet de quatre items comprenant l'état post-transaction $R_{\boldsymbol{\sigma}}$, la quantité cumulée du gaz consommé par le bloc contenant le reçu de transaction avec celle usée immédiatement après sa ratification R_u, l'ensemble des journaux créés durant la transaction $R_{\mathbf{l}}$, et le filtre de Bloom composé par l'information dans ces journaux R_b :

$$R \equiv (R_{\boldsymbol{\sigma}}, R_u, R_b, R_{\mathbf{l}}) \tag{18}$$

La fonction L_R prépare simplement le reçu de transaction à être transformé en un tableau d'octets ordonné RLP :

$$L_R(R) \equiv (\texttt{TRIE}(L_S(R_{\boldsymbol{\sigma}})), R_u, R_b, R_{\mathbf{l}}) \tag{19}$$

ainsi l'état post-transaction $R_{\boldsymbol{\sigma}}$ est encodé dans une structure de trie, dont la racine est le premier item.

On pose que R_u, le gaz consommé, est un entier positif et que le Bloom des logs R_b, est une empreinte de 2048 bits (256 octets) :

$$R_u \in \mathbb{P} \quad \wedge \quad R_b \in \mathbb{B}_{256} \tag{20}$$

Les entrées de log, $R_{\mathbf{l}}$, sont une série de traces notées, par exemple, $(O_0, O_1, ...)$. Une entrée de log O est un n-uplet d'une adresse d'enregistreur O_a, d'une série de sujets de log de 32-octets de long $O_{\mathbf{t}}$, et d'un certain nombre d'octets de données, $O_{\mathbf{d}}$:

$$O \equiv (O_a, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}}) \tag{21}$$

$$O_a \in \mathbb{B}_{20} \quad \wedge \quad \forall_{t \in O_{\mathbf{t}}} : t \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B} \tag{22}$$

On définit la fonction filtre de Bloom M comme la réduction d'une entrée de log en une empreinte de 256 octets :

$$M(O) \equiv \bigvee_{t \in \{O_a\} \cup O_{\mathbf{t}}} \left(M_{3:2048}(t)\right) \tag{23}$$

où $M_{3:2048}$ est un filtre de Bloom spécialisé qui extrait trois bits de 2048, étant donnée une série d'octets quelconque. Cela est fait en prenant les 11 bits de poids faible de chacune des trois premières paires d'octets dans une empreinte Keccak-256 de la série d'octets. Formellement :

$$M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{où:} \tag{24}$$

$$\mathbf{y} = (0, 0, ..., 0) \quad \text{excepté:} \tag{25}$$

$$\forall_{i \in \{0,2,4\}} : \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \tag{26}$$

$$m(\mathbf{x}, i) \equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 2048 \tag{27}$$

Où $\mathcal{B}$ est la fonction binaire telle que $\mathcal{B}_j(\mathbf{x})$ égale le bit d'index j (indexé à partir de 0) dans le tableau d'octets $\mathbf{x}$.

4.3.2. *Validation holistique.* On ne peut assurer la validité d'un bloc que s'il satisfait un certain nombre de conditions : il doit être cohérent avec l'oncle et avec les empreintes de transactions du bloc, et les transactions données $B_{\mathbf{T}}$ (comme spécifié en section 11), exécutées dans l'ordre sur l'état de base $\boldsymbol{\sigma}$ (dérivé de l'état final du bloc parent), donnent comme résultat un nouvel état de l'identité H_r :

$$\begin{aligned}
H_r &\equiv \texttt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) &\wedge\\
H_o &\equiv \texttt{KEC}(\texttt{RLP}(L^*_H(B_{\mathbf{U}}))) &\wedge\\
H_t &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{P} : p(i, L_T(B_{\mathbf{T}}[i]))\}) &\wedge\\
H_e &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{P} : p(i, L_R(B_{\mathbf{R}}[i]))\}) &\wedge\\
H_b &\equiv \textstyle\bigvee_{\mathbf{r} \in B_{\mathbf{R}}} (\mathbf{r}_b)
\end{aligned} \tag{28}$$

où $p(k, v)$ est simplement la paire RLP de l'index de la transaction dans le bloc d'une part, et du reçu de la transaction d'autre part :

$$p(k, v) \equiv \left(\texttt{RLP}(k), \texttt{RLP}(v)\right) \tag{29}$$

En outre :

$$\texttt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_r} \tag{30}$$

Ainsi $\texttt{TRIE}(L_S(\boldsymbol{\sigma}))$ est l'empreinte du noeud racine de l'arbre de Merkle radix contenant les paires clé-valeur de l'état $\boldsymbol{\sigma}$ avec des valeurs encodées par RLP, et $P(B_H)$ est le bloc parent de B, défini directement.

Les valeurs provenant du calcul des transactions, spécialement les reçus de transaction $B_{\mathbf{R}}$, et ceux définis par la fonction d'agrégation d'états Π, sont formalisés plus loin dans la section 11.4.

4.3.3. *Sérialisation.* Les fonctions L_B et L_H sont respectivement les fonctions préparatoires pour un bloc et un en-tête de bloc. De façon très semblable à la fonction de préparation du reçu de transaction L_R, nous déclarons les types et l'ordre de la structure pour la transformation RLP :

$$\begin{aligned}
L_H(H) \equiv (\ & H_p, H_o, H_c, H_r, H_t, H_e, H_b, H_d,\\
& H_i, H_l, H_g, H_s, H_x, H_m, H_n\ )
\end{aligned} \tag{31}$$

$$L_B(B) \equiv \left(L_H(B_H), L^*_T(B_{\mathbf{T}}), L^*_H(B_{\mathbf{U}})\right) \tag{32}$$

Avec L^*_T et L^*_H comme transformations élément par élément de la séquence, ainsi :

$$f^*\left((x_0, x_1, ...)\right) \equiv \left(f(x_0), f(x_1), ...\right) \quad \text{pour toute fonction } f \tag{33}$$

Les composants type sont définis ainsi :

$$\begin{array}{llllll}
H_p \in \mathbb{B}_{32} & \wedge & H_o \in \mathbb{B}_{32} & \wedge & H_c \in \mathbb{B}_{20} & \wedge\\
H_r \in \mathbb{B}_{32} & \wedge & H_t \in \mathbb{B}_{32} & \wedge & H_e \in \mathbb{B}_{32} & \wedge\\
H_b \in \mathbb{B}_{256} & \wedge & H_d \in \mathbb{P} & \wedge & H_i \in \mathbb{P} & \wedge\\
H_l \in \mathbb{P} & \wedge & H_g \in \mathbb{P} & \wedge & H_s \in \mathbb{P}_{256} & \wedge\\
H_x \in \mathbb{B} & \wedge & H_m \in \mathbb{B}_{32} & \wedge & H_n \in \mathbb{B}_8 &
\end{array} \tag{34}$$

où

$$\mathbb{B}_n = \{B : B \in \mathbb{B} \wedge \|B\| = n\} \tag{35}$$

Nous avons maintenant une spécification rigoureuse pour la construction de la structure formelle d'un bloc. La fonction RLP `RLP` (voir Annexe B) fournit la méthode de référence pour transformer cette structure en une séquence d'octets prête à la transmission ou au stockage local.

4.3.4. *Validation de l'en-tête de bloc.* On définit $P(B_H)$ comme le bloc parent de B, formellement :

$$P(H) \equiv B' : \texttt{KEC}(\texttt{RLP}(B'_H)) = H_p \tag{36}$$

Le numéro de bloc est le numéro du bloc parent incrémenté de un :

$$H_i \equiv P(H)_{H_i} + 1 \tag{37}$$

La difficulté de référence d'un bloc d'en-tête H est défini comme $D(H)$:

$$D(H) \equiv \begin{cases} D_0 & \text{si} \quad H_i = 0 \\ \max\left(D_0, P(H)_{H_d} + x \times \varsigma_1 + \epsilon\right) & \text{si} \quad H_i < N_H \\ \max\left(D_0, P(H)_{H_d} + x \times \varsigma_2 + \epsilon\right) & \text{sinon} \end{cases} \tag{38}$$

où :

$$D_0 \equiv 131072 \tag{39}$$

$$x \equiv \left\lfloor \frac{P(H)_{H_d}}{2048} \right\rfloor \tag{40}$$

$$\varsigma_1 \equiv \begin{cases} 1 & \text{si} \quad H_s < P(H)_{H_s} + 13 \\ -1 & \text{sinon} \end{cases} \tag{41}$$

$$\varsigma_2 \equiv \max\left(1 - \left\lfloor \frac{H_s - P(H)_{H_s}}{10} \right\rfloor, -99\right) \tag{42}$$

$$\epsilon \equiv \left\lfloor 2^{\lfloor H_i \div 100000 \rfloor - 2} \right\rfloor \tag{43}$$

la limite de reférence de gaz H_l d'un bloc d'en-tête H doit être conforme à la relation :

$$H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{44}$$

$$H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{45}$$

$$H_l \geqslant 125000 \tag{46}$$

H_s est l'horodatage du bloc H et doit être conforme à la relation :

$$H_s > P(H)_{H_s} \tag{47}$$

Ce mécanisme renforce une homéostasie pour les intervalles de temps entre les blocs ; une période de temps plus petite entre deux blocs entraîne une augmentation du niveau de difficulté et ainsi davantage de calcul, rallongeant d'autant la période suivante. Inversement, si la période est trop large, la difficulté, et le temps attendu pour le bloc suivant, sont réduits.

Le nonce H_n, doit satisfaire aux relations :

$$n \leqslant \frac{2^{256}}{H_d} \quad \wedge \quad m = H_m \tag{48}$$

avec $(n, m) = \texttt{PoW}(H_{\not n}, H_n, \mathbf{d})$.

Où $H_{\not n}$ est le nouvel en-tête de bloc H, mais *sans* le nonce et le mixHash, $\mathbf{d}$ étant le DAG courant, un grand ensemble de données utilisé pour calculer le mixHash, et $\texttt{PoW}$ étant la fonction de preuve-de-travail (voir section 11.5) : ceci est évalué en un tableau dont le premier item est le mixHash, pour preuve qu'un DAG correct a été utilisé, et le second item est un nombre pseudo-aléatoire cryptographiquement dépendant de H et $\mathbf{d}$. Étant donnée une distribution approximativement uniforme sur l'intervalle $[0, 2^{64}]$, le temps estimé pour trouver une solution est proportionnel à la difficulté H_d.

Ceci est le fondement de la sécurité de la chaîne de blocs et est la raison fondamentale pour laquelle un code malicieux ne peut pas propager des blocs nouvellement forgés qui pourraient autrement écraser (« réécrire ») l'historique. Parce que le nonce doit satisfaire cette condition, et parce que cette satisfaction dépend du contenu du bloc et des transactions qui le composent, créer de nouveaux blocs valides est difficile et, avec le temps, demande approximativement la puissance de calcul totale de la population de confiance parmi les pairs mineurs.

Ainsi on est en mesure de définir la fonction de validation de l'entête de bloc $V(H)$:

$$V(H) \equiv n \leqslant \frac{2^{256}}{H_d} \wedge m = H_m \quad \wedge \tag{49}$$

$$H_d = D(H) \quad \wedge \tag{50}$$

$$H_g \leq H_l \quad \wedge \tag{51}$$

$$H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{52}$$

$$H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{53}$$

$$H_l \geqslant 125000 \quad \wedge \tag{54}$$

$$H_s > P(H)_{H_s} \quad \wedge \tag{55}$$

$$H_i = P(H)_{H_i} + 1 \quad \wedge \tag{56}$$

$$\|H_x\| \leq 32 \tag{57}$$

où $(n, m) = \texttt{PoW}(H_{\not n}, H_n, \mathbf{d})$

Notons que **extraData** doit être au plus de 32 octets.

5. Gaz et paiement

Pour éviter d'avoir des problèmes d'abus du réseau et pour contourner les inévitables questions découlant de la complétude au sens de Turing, tout calcul programmable dans Ethereum est sujet à des frais (*fees*). Le barème des frais est spécifié en unités de *gaz* (voir Annexe G pour les frais associés avec divers calculs). Ainsi tout fragment donné de calcul programmable (ceci inclut la création de contrats, le lancement d'appels de messages, l'accès et l'utilisation de données associées aux comptes et l'exécution d'opérations sur la machine virtuelle) a un coût en gaz universellement approuvé.

Toute transaction se voit associer un montant spécifique en gaz : **gasLimit**. C'est ce montant qui est implicitement prélevé sur le compte de l'expéditeur. L'achat est fait à la valeur de **gasPrice**, également spécifié dans la transaction. La transaction est considérée comme invalide si le solde du compte ne peut assurer l'achat. Il est nommé **gasLimit** parce que toute quantité de gaz non utilisée est rendue (au même taux d'achat) au compte de l'expéditeur. Le gaz n'existe pas en dehors de l'exécution d'une transaction. Ainsi, pour des comptes avec du code de confiance associé, une limite relativement haute en gaz peut-elle être ainsi réservée.

En général, l'Ether utilisé pour acheter du gaz qui n'est pas rendu est livré à l'adresse du *bénéficiaire*, adresse typiquement sous le contrôle du mineur. Les contractants sont libres de spécifier un **gasPrice** à leur convenance mais les mineurs sont libres d'ignorer les transactions lors de leur choix. Un prix de gaz plus élevé pour une transaction coûtera davantage d'Ether à son expéditeur et donnera davantage au mineur, et donc elle sera plus aisément sélectionnée pour inclusion par davantage de mineurs. Les

mineurs, en général, choisiront d'annoncer le prix du gaz minimum pour lequel ils exécuteront des transactions et les contractants seront libres de marchander ces prix en déterminant quel prix du gaz offrir. Comme il y aura une distribution (pondérée) des minima des prix du gaz, les contractants devront nécessairement marchander entre baisser le prix du gaz et maximiser les chances que leurs transactions seront minées avec rapidité.

6. Exécution de transaction

L'exécution d'une transaction est la partie la plus complexe du protocole Ethereum: elle définit la fonction de transition d'état Υ. On suppose que toutes les transactions exécutées passent d'abord les tests de validité intrinsèques. Les voici:

(1) La transaction est un RLP bien formé sans octets additionnels à la fin;
(2) La signature de transaction est valide;
(3) Le nonce de transaction est valide (équivalent au nonce courant du compte de l'expéditeur);
(4) La limite de gaz n'est pas plus petite que le gaz intrinsèque, g_0, utilisé par la transaction;
(5) Le solde du compte de l'expéditeur contient au moins le coût, v_0, demandé comme paiement de départ.

Formellement, nous considérons la fonction Υ, T étant une transaction et $\boldsymbol{\sigma}$ l'état:

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{58}$$

$\boldsymbol{\sigma}'$ est donc l'état post-transactionnel. Nous définissons également Υ^g pour évaluer le montant de gaz utilisé dans l'exécution d'une transaction et $\Upsilon^{\mathbf{l}}$ pour évaluer les items de logs ajoutés, les deux étant formellement définis plus loin.

6.1. Sous-état.

Pendant l'exécution d'une transaction, nous voyons s'accroître certaines informations sur lesquelles on agit immédiatement à la suite de la transaction. Nous appelons ceci *sous-état de transaction* et le représentons par A, qui est un n-uplet:

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_r) \tag{59}$$

Le contenu du n-uplet comprend $A_{\mathbf{s}}$, l'ensemble d'autodestruction: un ensemble de comptes qui sera mis au rebut après la fin de la transaction. $A_{\mathbf{l}}$ est la série de logs: il s'agit d'une série de «points de contrôle» dans l'exécution du code de la VM qui permet de facilement tracer les appels de contrats par des observateurs extérieurs au monde Ethereum (comme des frontaux d'applications décentralisées). Enfin, il y a A_r, le solde de remboursement, accru par l'utilisation de l'instruction SSTORE afin de remettre la mémoire de stockage à zéro à partir d'une valeur différente de zéro. Bien qu'il ne soit pas immédiatement remboursé, il lui est permis de compenser partiellement les coûts totaux d'exécution.

Dans un souci de concision, nous définissons le sous-état vide A^0 comme n'ayant pas d'autodestruction, pas de logs et un solde de remboursement à zéro:

$$A^0 \equiv (\varnothing, (), 0) \tag{60}$$

6.2. Exécution.

Nous définissons le gaz intrinsèque g_0 comme le montant de gaz que cette transaction demande pour être payée avant exécution, comme suit:

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{txdatazero} & \text{si} \quad i = 0 \\ G_{txdatanonzero} & \text{sinon} \end{cases} \tag{61}$$

$$+ \begin{cases} G_{\text{txcreate}} & \text{si} \quad T_t = \varnothing \wedge H_i \geq N_H \\ 0 & \text{sinon} \end{cases} \tag{62}$$

$$+ G_{transaction} \tag{63}$$

où $T_{\mathbf{i}}, T_{\mathbf{d}}$ signifie la série d'octets des données et du code d'initialisation de l'EVM associés à la transaction, selon que la transaction sert à une création de contrat ou un a un appel de message. G_{txcreate} est ajouté si la transaction crée un message mais ce n'est pas le cas s'il s'agit d'un résultat de code EVM. G est totalement défini à l'Annexe G.

Le coût de départ v_0 est calculé par:

$$v_0 \equiv T_g T_p + T_v \tag{64}$$

La validité est déterminée par:

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] &\neq \varnothing \quad \wedge \\ T_n &= \boldsymbol{\sigma}[S(T)]_n \quad \wedge \\ g_0 &\leqslant T_g \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_b \quad \wedge \\ T_g &\leqslant B_{Hl} - \ell(B_{\mathbf{R}})_u \end{aligned} \tag{65}$$

On note la condition finale: la somme de la limite en gaz de la transaction, T_g, et le gaz auparavant utilisé dans ce bloc, donné par $\ell(B_{\mathbf{R}})_u$, ne doit pas être plus grand que la **gasLimit**, B_{Hl} du bloc.

L'exécution d'une transaction valide commence par un changement irrévocable de l'état: le nonce du compte de l'expéditeur, $S(T)$, est incrémenté et le solde est réduit par une partie du coût de départ, $T_g T_p$. Le gaz disponible pour le calcul en cours, g, est défini par $T_g - g_0$. Le calcul, qu'il s'agisse d'une création de contrat ou d'un appel de message, donne un état final (qui peut être légalement équivalent à l'état courant), dont le changement est déterministe et jamais invalide: il ne peut y avoir de transaction invalide à partir de ce point.

Nous définissons l'état au point de contrôle $\boldsymbol{\sigma}_0$:

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{66}$$

$$\boldsymbol{\sigma}_0[S(T)]_b \equiv \boldsymbol{\sigma}[S(T)]_b - T_g T_p \tag{67}$$

$$\boldsymbol{\sigma}_0[S(T)]_n \equiv \boldsymbol{\sigma}[S(T)]_n + 1 \tag{68}$$

L'évaluation de $\boldsymbol{\sigma}_P$ à partir de $\boldsymbol{\sigma}_0$ dépend du type de la transaction, selon qu'il s'agit d'un création de contrat ou d'un appel de message; nous définissons le n-uplet de l'état provisoire de post-exécution $\boldsymbol{\sigma}_P$, gaz restant g' et sous-état A:

$$(\boldsymbol{\sigma}_P, g', A) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}_0, S(T), T_o, \\ \quad g, T_p, T_v, T_{\mathbf{i}}, 0) & \text{si} \quad T_t = \varnothing \\ \Theta_3(\boldsymbol{\sigma}_0, S(T), T_o, \\ \quad T_t, T_t, g, T_p, T_v, T_v, T_{\mathbf{d}}, 0) & \text{sinon} \end{cases} \tag{69}$$

où g est le montant de gaz restant après avoir déduit le montant de base demandé à payer pour l'existence de la transaction:

$$g \equiv T_g - g_0 \tag{70}$$

et T_o est l'auteur originel de la transaction, qui peut différer de l'expéditeur dans le cas d'un appel de message ou d'une création de contrat qui n'a pas été directement déclenché par une transaction mais qui vient de l'exécution de code EVM.

On remarque que nous utilisons Θ_3 pour noter le fait que seuls les trois premiers composants de la valeur de la fonction sont pris ; le composant final représente la valeur de sortie d'un appel de message (un tableau d'octets) et est inutilisé dans le contexte de l'évaluation d'une transaction.

Après le traitement de l'appel de message ou de la création de contrat, l'état est finalisé en déterminant le montant à rembourser, g^*, du gaz restant, g', plus un certain montant du compteur de remboursement, à l'expéditeur du taux d'origine.

$$g^* \equiv g' + \min\{\left\lfloor \frac{T_g - g'}{2} \right\rfloor, A_r\} \tag{71}$$

Le montant total à rembourser est le gaz légitimement restant g', ajouté à A_r, ce dernier composant étant limité à un maximum de la moitié (tronquée) du montant total utilisé $T_g - g'$.

L'Ether du gaz est donné au mineur dont l'adresse est spécifiée comme bénéficiaire du bloc présent B. Nous définissons donc l'état pré-final $\boldsymbol{\sigma}^*$ en termes d'état provisoire $\boldsymbol{\sigma}_P$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_P \quad \text{except} \tag{72}$$
$$\boldsymbol{\sigma}^*[S(T)]_b \equiv \boldsymbol{\sigma}_P[S(T)]_b + g^* T_p \tag{73}$$
$$\boldsymbol{\sigma}^*[m]_b \equiv \boldsymbol{\sigma}_P[m]_b + (T_g - g^*) T_p \tag{74}$$
$$m \equiv B_{H_c} \tag{75}$$

L'état final, $\boldsymbol{\sigma}'$, est atteint après la suppression de tous les comptes qui apparaissent dans la liste d'autodestruction :

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{76}$$
$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] \equiv \varnothing \tag{77}$$

Et finalement, nous spécifions Υ^g, le gaz total utilisé dans cette transaction et $\Upsilon^{\mathbf{l}}$, les logs créés par cette transaction :

$$\Upsilon^g(\boldsymbol{\sigma}, T) \equiv T_g - g' \tag{78}$$
$$\Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{79}$$

Ils sont utilisés pour aider à définir le reçu de la transaction qui sera détaillé plus bas.

7. Création de contrat

Un certain nombre de paramètres intrinsèques sont utilisés quand on crée un compte : expéditeur (s), négociateur (*transactor*) originel (o), prix du gaz (p), dotation (v), assemblés avec un tableau d'octets de longueur variable, $\mathbf{i}$, le code EVM d'initialisation et enfin la profondeur actuelle de la pile d'appel de message/création de contrat (e)

Nous définissons formellement la fonction de création comme la fonction Λ qui évalue, à partir de ces valeurs et de l'état $\boldsymbol{\sigma}$, le n-uplet contenant le nouvel état, le gaz restant et le substrat de transaction augmenté $(\boldsymbol{\sigma}', g', A)$, comme en section 6 :

$$(\boldsymbol{\sigma}', g', A) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e) \tag{80}$$

L'adresse du nouveau compte est défini comme étant les 160 bits de droite de l'empreinte Keccak de l'encodage RLP de la structure ne contenant que l'expéditeur et le nonce. Nous définissons donc l'adresse résultante pour le nouveau compte a :

$$a \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\Big(\texttt{RLP}\big(\ (s, \boldsymbol{\sigma}[s]_n - 1)\ \big)\Big)\Big) \tag{81}$$

où KEC est la fonction de hachage Keccak sur 256 bits, RLP est la fonction d'encodage RLP, $\mathcal{B}_{a..b}(X)$ donne une valeur binaire contenant les bits des indices dans l'intervalle $[a, b]$ des données binaires X et $\boldsymbol{\sigma}[x]$ est l'état de l'adresse de x ou $\varnothing$ si aucun n'existe. Notons que nous utilisons un de moins que la valeur du nonce de l'expéditeur ; nous alléguons que nous avons incrémenté le nonce du compte de l'expéditeur avant cet appel et la valeur utilisée est donc celle du début de la transaction ou de l'opération de VM responsable.

Le nonce du compte est initialement défini à zéro, le solde à la valeur passée, la mémoire de stockage à vide et l'empreinte du code à l'empreinte Keccak sur 256 bits de la chaîne vide ; la valeur passée est également défalquée du solde de l'expéditeur. Donc, l'état modifié devient $\boldsymbol{\sigma}^*$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{82}$$
$$\boldsymbol{\sigma}^*[a] \equiv \big(0, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}(())\big) \tag{83}$$
$$\boldsymbol{\sigma}^*[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{84}$$

où v' est la valeur préexistante du compte le cas échéant :

$$v' \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_b & \text{sinon} \end{cases} \tag{85}$$

Enfin, le compte est initialisé par l'exécution du code EVM d'initialisation $\mathbf{i}$ d'après le modèle d'exécution (voir la section 9). L'exécution du code peut engendrer plusieurs événements qui ne sont pas internes à l'état d'exécution : la mémoire de stockage peut être altérée, des comptes supplémentaires peuvent être créés et d'autres appels de message peuvent être lancés. En tant que telle, la fonction d'exécution de code Ξ donne un n-uplet de l'état résultant $\boldsymbol{\sigma}^{**}$, du gaz disponible restant g^{**}, du substrat accru A et du code du corps du compte $\mathbf{o}$.

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I) \tag{86}$$

où I contient les paramètres de l'environnement d'exécution tels que définis en section 9, c'est-à-dire :

$$I_a \equiv a \tag{87}$$
$$I_o \equiv o \tag{88}$$
$$I_p \equiv p \tag{89}$$
$$I_{\mathbf{d}} \equiv () \tag{90}$$
$$I_s \equiv s \tag{91}$$
$$I_v \equiv v \tag{92}$$
$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{93}$$
$$I_e \equiv e \tag{94}$$

$I_{\mathbf{d}}$ donne le n-uplet vide puisqu'il n'y a pas de données en entrée de cet appel. I_H n'a pas de traitement spécial et est déterminé à partir de la blockchain.

L'exécution du code diminue le gaz, lequel ne peut pas descendre en-dessous de zéro, donc l'exécution peut sortir avant que le code ne soit arrivé à un état d'arrêt. Dans

ces cas exceptionnels (et dans plusieurs autres), nous disons qu'une exception *Out-of-Gas* (panne d'essence) s'est produite : l'état évalué est défini comme étant l'ensemble vide, $\varnothing$, et l'opération de création dans son intégralité ne doit avoir aucun effet sur l'état, le laissant dans les faits tel qu'il était immédiatement avant la tentative de création.

Si le code d'initialisation se termine avec succès, un coût final de création de contrat est payé, le coût de dépôt du code, c, étant proportionnel à la taille du code du contrat créé :

$$c \equiv G_{codedeposit} \times |\mathbf{o}| \tag{95}$$

S'il n'y a pas assez de gaz restant pour payer ceci, c'est-à-dire $g^{**} < c$, alors nous déclarons également une exception Out-of-Gas.

Le gaz restant sera à zéro dans ces circonstances exceptionnelles, c'est-à-dire que si la création a été conduite à la réception d'une transaction, alors cela n'affecte pas le paiement du coût intrinsèque de la création du contrat ; celui-ci est payé de toute manière. Cependant, la valeur de la transaction n'est pas transférée à l'adresse du contrat avorté quand nous sommes Out-of-Gas.

Si ce type d'exception ne se produit pas, alors le gaz restant est renvoyé à l'initiateur et l'état maintenant altéré a le droit de persister. Nous pouvons donc formellement spécifier l'état, le gaz et le sous-état résultants par $(\boldsymbol{\sigma}', g', A)$ où :

(96)

$$g' \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ g^{**} & \text{si} \quad g^{**} < c \wedge H_i < N_H \\ g^{**} - c & \text{sinon} \end{cases}$$

(97)

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{si} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{si} \quad g^{**} < c \wedge H_i < N_H \\ \boldsymbol{\sigma}^{**} \quad \text{excepté:} & \\ \quad \boldsymbol{\sigma}'[a]_c = \texttt{KEC}(\mathbf{o}) & \text{sinon} \end{cases}$$

L'exception dans la détermination de $\boldsymbol{\sigma}'$ dicte que $\mathbf{o}$, la séquence d'octets résultant de l'exécution du code d'initialisation, spécifie le code final du corps du compte nouvellement créé.

On remarque que l'intention à partir du bloc N_H (*Homestead*) est que le résultat est soit un nouveau contrat créé avec succès avec sa dotation, soit aucun contrat et aucun transfert de valeur.

7.1. **Subtilités.** Notons qu'alors que le code d'initialisation est en train de s'exécuter, l'adresse nouvellement créée existe mais sans code intrinsèque. Un appel de message qu'il reçoit pendant ce temps n'engendre donc aucune exécution de code. Si l'exécution de l'initialisation se termine par une instruction SELFDESTRUCT, cela importe peu puisque le compte est supprimé avant la fin de la transaction. Pour un code STOP, ou si le code renvoyé est vide, alors l'état reste avec un compte zombie et tout solde résiduel reste à jamais verrouillé dans ce compte.

8. Appel de message

Lors de l'exécution d'un appel de message, plusieurs paramètres sont requis : émetteur (s), créateur de la transaction (o), récepteur (r), le compte dont le code doit être exécuté (c, habituellement le même que le récepteur), le gaz disponible (g), valeur (v) et prix du gaz (p) ensemble dans un tableau d'octets de longueur arbitraire, $\mathbf{d}$, les données d'entrée de l'appel de message et finalement la hauteur actuelle de la pile du appel de message ou de la création de contrat (e).

À part l'évaluation de nouveaux états et de leurs sous-états, un appel de message a aussi un autre composant—les données de sortie désignées par le tableau d'octets $\mathbf{o}$. Il est ignoré durant les exécutions de transactions, mais les appels de message peuvent être lancés par l'exécution de code dans la VM et dans ce cas cette information est utilisée.

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e) \tag{98}$$

Notons que nous devons différencier la valeur qui doit être transférée v, de la valeur apparente dans le contexte d'execution $\tilde{v}$, pour l'instruction DELEGATECALL.

On définit $\boldsymbol{\sigma}_1$, le premier état de transition comme l'état original mais avec la valeur transférée de l'émetteur au récepteur :

$$\boldsymbol{\sigma}_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{99}$$

Au long du présent ouvrage, il est admis que si $\boldsymbol{\sigma}_1[r]$ était indéfini à l'origine, il sera créé comme un compte sans code ou état et avec zéro solde ou nonce. La précédente équation doit donc être interprétée comme suit :

$$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:} \tag{100}$$

$$\boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}'_1[s]_b - v \tag{101}$$

$$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{102}$$

$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (v, 0, \texttt{KEC}(()), \texttt{TRIE}(\varnothing)) & \text{si} \quad \boldsymbol{\sigma}[r] = \varnothing \\ \boldsymbol{\sigma}'_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v & \text{sinon} \end{cases} \tag{103}$$

Le code associé au compte (identifié comme le fragment dont l'empreinte Keccack est $\boldsymbol{\sigma}[c]_c$) est exécuté conformément au modèle d'exécution (voir section 9). Comme avec la création de contrat, si l'exécution cesse pour cause d'anomalie (i.e. à cause de l'épuisement de la fourniture en gaz, épuisement de la pile, saut invalide ou instruction invalide), alors l'appelant n'est pas remboursé et l'état est remis au point immédiatement avant le transfert de solde (i.e. $\boldsymbol{\sigma}$).

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{si } \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{sinon} \end{cases} \tag{104}$$

$$(\boldsymbol{\sigma}^{**}, g', \mathbf{s}, \mathbf{o}) \equiv \begin{cases} \Xi_{\text{ECREC}}(\boldsymbol{\sigma}_1, g, I) & \text{si } r = 1 \\ \Xi_{\text{SHA256}}(\boldsymbol{\sigma}_1, g, I) & \text{si } r = 2 \\ \Xi_{\text{RIP160}}(\boldsymbol{\sigma}_1, g, I) & \text{si } r = 3 \\ \Xi_{\text{ID}}(\boldsymbol{\sigma}_1, g, I) & \text{si } r = 4 \\ \Xi(\boldsymbol{\sigma}_1, g, I) & \text{sinon} \end{cases} \tag{105}$$

$$I_a \equiv r \tag{106}$$

$$I_o \equiv o \tag{107}$$

$$I_p \equiv p \tag{108}$$

$$I_{\mathbf{d}} \equiv \mathbf{d} \tag{109}$$

$$I_s \equiv s \tag{110}$$

$$I_v \equiv \tilde{v} \tag{111}$$

$$I_e \equiv e \tag{112}$$

$$\text{Let } \texttt{KEC}(I_{\mathbf{b}}) = \boldsymbol{\sigma}[c]_c \tag{113}$$

On suppose que le client aura stocké la paire $(\texttt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$ préalablement pour pouvoir déterminer $I_{\mathbf{b}}$.

Comme on peut le voir, il y a quatre exceptions au framework général d'exécution Ξ lors de l'évaluation d'un appel de message: ce sont les quatre contrats «précompilés», compris comme des éléments d'architecture préliminaires qui pourront ultérieurement devenir des *extensions natives*. Les quatre contrats aux adresses 1, 2, 3 et 4 exécutent la fonction de récupération de la clé publique de courbe elliptique, le schéma d'empreinte SHA2 256-bit, le schéma d'empreinte RIPEMD 160-bit et la fonction d'identité respectivement.

Leur définition formelle complète se trouve en Annexe E.

9. Modèle d'exécution

Le modèle d'exécution spécifie la façon dont l'état du système est altéré étant donnés une série d'instructions de bytecode et un petit n-uplet de données environnementales. On le spécifie par un modèle formel d'une machine à état nommée Ethereum Virtual Machine (EVM). Il s'agit d'une machine de Turing quasi-complète; la qualification de *quasi* vient de ce que le traitement est intrinsèquement borné par un paramètre, *gas* (gaz), qui limite le traitement total effectué.

9.1. **Fondamentaux.** L'EVM est une architecture simple basée sur une pile. La taille des mots de la machine (et donc la taille de l'élément de la pile) est de 256 bits. Ce choix est destiné à faciliter le mécanisme d'empreinte Keccak-256 et les calculs de courbes elliptiques. Le modèle de mémoire est un simple tableau d'octets adressé par mots. La pile a une taille maximale de 1024. La machine comporte également un modèle de stockage indépendant, conceptuellement similaire à celui de la mémoire mais, au lieu d'un tableau d'octets, il s'agit d'un tableau de mots adressé par mots. Au contraire de la mémoire, qui est volatile, le stockage est non volatile et est maintenu comme faisant partie de l'état du système. Tous les emplacements tant en stockage qu'en mémoire sont définis à zéro à l'initialisation.

La machine n'est pas conçue selon l'architecture standard de Von Neumann. Au lieu de stocker le code des programmes dans une mémoire transitoire ou permanente générique, il est mis à part dans une ROM virtuelle avec laquelle seule une instruction spécialisée peut interagir.

La machine peut entrer dans une exécution exceptionnelle pour plusieurs raisons, comme un dépassement de dépilage et des instructions invalides. Comme l'exception *out-of-gas* (OOG), elles ne laissent pas les changements d'état intacts. La machine s'arrête immédiatement et rapporte le problème à l'agent d'exécution (soit le processeur de transactions, soit, récursivement, l'environnement d'exécution du lancement) qui le gèrera de son côté.

9.2. **Aperçu des frais.** Les *fees* (frais), chiffrés en gaz, sont décomptés en trois circonstances distinctes, toutes trois étant des prérequis à l'exécution d'une opération. D'abord et le plus fréquemment, les frais intrinsèques au calcul de l'opération (voir Appendice G). Ensuite le gaz peut être déduit afin de constituer le paiement d'un appel de message ou d'une création de contrat subordonné; cela fait partie du paiement de CREATE, CALL et CALLCODE. Enfin, le gaz peut être payé en raison de l'augmentation de l'utilisation de la mémoire.

Pendant l'exécution d'un compte, les frais totaux dûs pour l'utilisation de la mémoire sont proportionnels au plus petit multiple de 32 octets qui est requis tel que tous les index mémoire (en lecture ou en écriture) sont inclus dans l'intervalle. Le paiement se fait en juste-à-temps; le référencement d'une zone de mémoire plus grande d'au moins 32 octets que la mémoire précédemment référencée résulte avec certitude en des frais d'utilisation mémoire supplémentaires. Ces frais rendent extrêmement improbable un adressage dépassant la limite de 32 bits. Cela dit, les implémentations doivent être capables de gérer cette éventualité.

Les frais de stockage se comportent de manière légèrement différente. Pour inciter à minimiser l'utilisation du stockage (qui correspond directement à une base de données d'état plus grande sur tous les nœuds), les frais d'exécution d'une opération supprimant une entrée dans le stockage ne sont pas seulement abandonnés mais un remboursement explicite est effectué; dans les faits, ce remboursement est payé immédiatement puisque l'utilisation initiale d'un emplacement mémoire coûte bien plus qu'une utilisation courante.

Voir l'appendice H pour une définition rigoureuse du coût en gaz de l'EVM.

9.3. **Environnement d'exécution.** En plus de l'état du système $\boldsymbol{\sigma}$ et du gaz restant pour le traitement g, l'agent d'exécution doit fournir plusieurs informations importantes utilisées dans l'environnement d'exécution, qui sont contenues dans le n-uplet I:

- I_a, l'adresse du compte qui possède le code en cours d'exécution.
- I_o, l'adresse de l'expéditeur de la transaction d'où provient cette exécution.
- I_p, le prix du gaz dans la transaction d'où provient cette exécution.
- $I_{\mathbf{d}}$, le tableau d'octets qui constitue les données d'entrée pour cette exécution; si l'agent d'exécution est une transaction, il s'agit des données de transaction.

- I_s, l'adresse du compte qui a provoqué l'exécution du code; si l'agent d'exécution est une transaction, il s'agit de l'expéditeur de la transaction.
- I_v, la valeur, en Wei, passée à ce compte comme faisant partie de la même procédure que l'exécution; si l'agent d'exécution est une transaction, c'est la valeur de la transaction.
- $I_{\mathbf{b}}$, le tableau d'octets qui est le code machine à exécuter.
- I_H, l'en-tête de bloc du bloc actuel.
- I_e, la profondeur de l'appel de message actuel ou de la création de contrat actuelle (c.à.d. le nombre de CALL ou de CREATE exécutés à ce moment).

Le modèle d'exécution définit la fonction Ξ, qui peut calculer l'état résultant $\boldsymbol{\sigma}'$, le gaz restant g', le sous-état accumulé et la sortie résultante, $\mathbf{o}$, étant données ces définitions. Dans le contexte présent, nous définissons:

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{114}$$

où nous nous rappellerons que A, le sous-état accumulé, est défini comme le n-uplet de l'ensemble d'autodestruction $\mathbf{s}$, la série des logs $\mathbf{l}$ et les remboursements r:

$$A \equiv (\mathbf{s}, \mathbf{l}, r) \tag{115}$$

9.4. **Aperçu de l'exécution.** Nous devons maintenant définir la fonction Ξ. Dans la plupart des implémentations concrètes, elle sera conçue comme une progression itérative de la paire comprenant l'état complet du système, $\boldsymbol{\sigma}$ et de l'état de la machine, $\boldsymbol{\mu}$. Formellement, nous la définissons récursivement avec une fonction X, qui utilise une fonction itératrice O (qui définit le résultat d'un cycle unique de la machine à état) avec les fonctions Z, qui détermine si l'état actuel est un état d'arrêt exceptionnel de la machine, et H, spécifiant les données de sortie de l'instruction si et seulement si l'état actuel est un état d'arrêt normal de la machine.

La séquence vide, notée (), n'est pas égale à l'ensemble vide, noté $\varnothing$; c'est important au moment d'interpréter la sortie de H, qui donne $\varnothing$ quand l'exécution doit continuer mais une série (potentiellement vide) quand l'exécution doit s'arrêter.

$$\Xi(\boldsymbol{\sigma}, g, I) \equiv X_{0,1,2,4}\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \tag{116}$$

$$\boldsymbol{\mu}_g \equiv g \tag{117}$$

$$\boldsymbol{\mu}_{pc} \equiv 0 \tag{118}$$

$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{119}$$

$$\boldsymbol{\mu}_i \equiv 0 \tag{120}$$

$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{121}$$

(122)

$$X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv \begin{cases} \big(\varnothing, \boldsymbol{\mu}, A^0, I, ()\big) & \text{si} \quad Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{si} \quad \mathbf{o} \neq \varnothing \\ X\big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) & \text{sinon} \end{cases}$$

où

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{123}$$

$$(a, b, c) \cdot d \equiv (a, b, c, d) \tag{124}$$

On remarque que nous devons omettre la quatrième valeur du n-uplet renvoyé par X et extraire le gaz restant $\boldsymbol{\mu}'_g$ de l'état machine résultant $\boldsymbol{\mu}'$ pour évaluer correctement Ξ.

On cycle donc dans X (ici, récursivement, mais on s'attend généralement à ce que les implémentations emploient une simple boucle itérative) jusqu'à ce que soit Z devienne vrai, indiquant ainsi que l'état actuel est exceptionnel et que la machine doit être arrêtée en revenant sur toutes les modifications, ou jusqu'à ce que H devienne une série (au lieu de l'ensemble vide) indiquant que la machine a atteint un arrêt contrôlé.

9.4.1. *État de la machine.* L'état de la machine $\boldsymbol{\mu}$ est défini par le n-uplet $(g, pc, \mathbf{m}, i, \mathbf{s})$ qui sont le gaz disponible, le compteur de programme $pc \in \mathbb{P}_{256}$, le contenu de la mémoire, le nombre de mots actifs en mémoire (en comptant en continu depuis la position 0) et le contenu de la pile. Le contenu de la mémoire $\boldsymbol{\mu}_{\mathbf{m}}$ est une série de zéros de taille 2^{256}.

Dans un but de lisibilité, les mnémoniques d'instructions, écrites en petites capitales (comme ADD), doivent être interprétées comme leurs équivalents numériques; la table complète des instructions et de leurs spécificités est donnée à l'Appendice H.

Dans le but de définir Z, H et O, nous définissons x comme l'opération actuelle à exécuter:

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{si} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{sinon} \end{cases} \tag{125}$$

Nous supposons également les montants fixes de δ et α, en spécifiant les éléments de la pile qui ont été supprimés et ajoutés, les deux étant *subscriptables* sur l'instruction et une fonction de coût d'instruction C qui donne le coût total, en gaz, de l'exécution de l'instruction donnée.

9.4.2. *Arrêt exceptionnel.* La fonction d'arrêt exceptionnel Z est définie par:

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv \; & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w \in \{\text{JUMP}, \text{JUMPI}\} \quad \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \end{aligned} \tag{126}$$

Cela établit que l'éxécution se trouve dans un état d'arrêt exceptionnel s'il n'y a pas assez de gaz, si l'instruction est invalide (et donc son *subscript* est indéfini), s'il n'y a pas suffisamment d'éléments de pile, si une destination de JUMP/JUMPI était invalide ou si la nouvelle taille de la pile était supérieure à 1024. Le lecteur attentif aura compris que cela signifie également qu'aucune instruction ne peut, par son exécution, provoquer un arrêt exceptionnel.

9.4.3. *Validité d'une destination de saut.* Nous avons auparavant utilisé D comme fonction pour déterminer l'ensemble des destinations de saut valides étant donné le code en train de tourner. Nous définissons ceci comme toute position dans le code occupée par une instruction JUMPDEST.

Toutes ces positions doivent se trouver sur des limites d'instruction valides au lieu d'occuper la partie de données des opérations PUSH et doivent apparaître dans la portion explicitement définie du code (et non dans les opérations STOP implicitement définies qui les suivent).

Formellement :

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0) \tag{127}$$

où :

$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{si} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{si} \quad \mathbf{c}[i] = \text{JUMPDEST} \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{sinon} \end{cases} \tag{128}$$

où N est la position de l'instruction valide suivante dans le code, en sautant les données d'une instruction PUSH le cas échéant :

$$N(i, w) \equiv \begin{cases} i + w - \text{PUSH1} + 2 & \text{si} \quad w \in [\text{PUSH1}, \text{PUSH32}] \\ i + 1 & \text{sinon} \end{cases} \tag{129}$$

9.4.4. *Arrêt normal.* La fonction d'arrêt normal H est définie par :

$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) & \text{si} \quad w = \text{RETURN} \\ () & \text{si} \quad w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing & \text{sinon} \end{cases} \tag{130}$$

L'opération d'arrêt renvoyant les données, RETURN, a une fonction spéciale H_{RETURN}, définie en Annexe H.

9.5. **Le cycle d'exécution.** Les éléments de la pile sont ajoutés ou retirés de la partie la plus à gauche et dont l'index est le plus bas de la série ; tous les autres éléments restent inchangés :

$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \tag{131}$$

$$\Delta \equiv \alpha_w - \delta_w \tag{132}$$

$$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \tag{133}$$

$$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x + \Delta] \tag{134}$$

Le gaz est réduit du coût en gaz de l'instruction et, pour la plupart des instructions, le compteur de programme s'incrémente à chaque cycle. Pour les trois exceptions, nous posons une fonction J souscrite par une instruction parmi deux, qui donne les valeurs en accord avec :

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{135}$$

$$\boldsymbol{\mu}'_{pc} \equiv \begin{cases} J_{\text{JUMP}}(\boldsymbol{\mu}) & \text{si} \quad w = \text{JUMP} \\ J_{\text{JUMPI}}(\boldsymbol{\mu}) & \text{si} \quad w = \text{JUMPI} \\ N(\boldsymbol{\mu}_{pc}, w) & \text{sinon} \end{cases} \tag{136}$$

En général, nous supposons que la mémoire, l'ensemble d'autodestruction et l'état du système ne changent pas :

$$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}} \tag{137}$$

$$\boldsymbol{\mu}'_i \equiv \boldsymbol{\mu}_i \tag{138}$$

$$A' \equiv A \tag{139}$$

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma} \tag{140}$$

Cependant, les instructions altèrent typiquement une ou plusieurs composantes de ces valeurs. Les composantes altérées listées par instruction sont notées à l'Annexe H, accompagnées des valeurs de α et de δ avec une description formelle des exigences en gaz.

10. De l'arbre des blocs à la chaîne des blocs

La blockchain typique est un chemin de la racine jusqu'à la feuille à travers toute l'arborescence des blocs. Pour atteindre le consensus sur le chemin à suivre, conceptuellement, nous identifions celui où le plus de calculs ont été effectués, c'est-à-dire le chemin dont *le poids est le plus élevé*. L'un des facteurs qui permettent de déterminer le chemin dont poids le plus élevé est clairement le numéro de bloc de la feuille, qui est équivalent au nombre de blocs sans compter le bloc *genesis* non miné, dans le chemin. Plus le chemin est long, plus l'effort nécessaire pour arriver à la feuille a été grand. Ce principe est comparable à d'autres, comme ceux employés dans les protocoles dérivés de Bitcoin.

Comme un en-tête de bloc inclut la difficulté, il suffit à lui seul à valider le calcul effectué. Chaque bloc contribue au calcul total, ou à la *diffulté totale* d'une chaîne.

En conséquence, nous définissons récursivement la difficulté totale du bloc B comme suit :

$$B_t \equiv B'_t + B_d \tag{141}$$

$$B' \equiv P(B_H) \tag{142}$$

Cela implique qu'étant donné un bloc B, B_t est sa difficulté totale, B' est son bloc parent et B_d est sa difficulté.

11. Block Finalisation

Le traitement de finalisation d'un bloc comporte quatre étapes :

(1) Valider (ou, pour du minage, déterminer) les oncles ;
(2) Valider (ou, pour du minage, déterminer) les transactions ;
(3) Appliquer les récompenses ;
(4) Vérifier (ou, pour du minage, calculer) l'état et le nonce.

11.1. **Validation des oncles.** La validation des en-têtes des oncles ne signifie rien de plus que la vérification que chaque en-tête d'oncle est un en-tête valide et qu'il satisfait la relation de l'oncle de Nième génération au bloc actuel où $N \leq 6$. Le maximum d'en-têtes d'oncles est deux. Formellement :

$$\|B_{\mathbf{U}}\| \leqslant 2 \bigwedge_{U \in B_{\mathbf{U}}} V(U) \ \wedge \ k(U, P(B_H)_H, 6) \tag{143}$$

où k note la propriété « is-kin » :

$$k(U, H, n) \equiv \begin{cases} false & \text{si} \quad n = 0 \\ s(U, H) & \\ \quad \vee\, k(U, P(H)_H, n-1) & \text{sinon} \end{cases} \tag{144}$$

et s note la propriété « is-sibling » :

$$s(U, H) \equiv (P(H) = P(U) \ \wedge \ H \neq U \ \wedge \ U \notin B(H)_{\mathbf{U}}) \tag{145}$$

où $B(H)$ est le bloc de l'en-tête correspondant H.

11.2. **Validation de transaction.** Le **gasUsed** donné doit correspondre fidèlement aux transactions listées : B_{Hg}, le gaz total utilisé dans le bloc doit être égal au gaz accumulé utilisé d'après la transaction finale :

$$B_{Hg} = \ell(\mathbf{R})_u \tag{146}$$

11.3. **Application des récompenses.** L'application des récompenses à un bloc implique d'augmenter le solde des comptes de l'adresse du bénéficiaire du bloc et de chaque oncle d'un certain montant. Nous ajoutons R_b au compte du bénéficiaire; pour chaque oncle, nous ajoutons un $\frac{1}{32}$ supplémentaire de la récompense du bloc à celui du bénéficiaire du bloc et le bénéficiaire de l'oncle est récompensé en fonction du numéro de bloc. Formellement, nous définissons la fonction Ω:

$$\Omega(B, \boldsymbol{\sigma}) \equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{excepté:} \tag{147}$$

$$\boldsymbol{\sigma}'[B_{Hc}]_b = \boldsymbol{\sigma}[B_{Hc}]_b + (1 + \frac{\|B_{\mathbf{U}}\|}{32})R_b \tag{148}$$

$$\forall_{U \in B_{\mathbf{U}}} : \tag{149}$$

$$\boldsymbol{\sigma}'[U_c]_b = \boldsymbol{\sigma}[U_c]_b + (1 + \frac{1}{8}(U_i - B_{Hi}))R_b$$

S'il existe des collision des adresses de bénéficiaires entre les oncles et le bloc (c'est-à-dire deux oncles ave la même adresse de bénéficiaire ou un oncle avec la même adresse de bénéficiaire que celle du bloc actuel), les additions sont appliquées de manière cumulative.

Nous définissons la récompense de bloc qui est de 5 Ether:

$$\text{Let} \quad R_b = 5 \times 10^{18} \tag{150}$$

11.4. **Validation de l'état et du nonce.** Nous pouvons maintenant définir la fonction, Γ, qui fait correspondre un bloc B à son état d'initialisation:

$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 & \text{si} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_i : \texttt{TRIE}(L_S(\boldsymbol{\sigma}_i)) = P(B_H)_{H_r} & \text{sinon} \end{cases} \tag{151}$$

Ici, $\texttt{TRIE}(L_S(\boldsymbol{\sigma}_i))$ signifie l'empreinte du nœud racine du trie (arbre préfixe) d'état $\boldsymbol{\sigma}_i$; on suppose que les implémentations le stockeront dans la base d'état, ce qui est trivial et efficient puisque le trie est par nature une structure de données immuable.

Et nous définissons finalement Φ, la fonction de transition de bloc, qui fait correspondre un bloc incomplet B à un bloc complet B':

$$\Phi(B) \equiv B' : \quad B' = B^* \quad \text{excepté:} \tag{152}$$

$$B'_n = n : \quad x \leqslant \frac{2^{256}}{H_d} \tag{153}$$

$$B'_m = m \quad \text{avec } (x, m) = \texttt{PoW}(B^*_{\mathbf{H}}, n, \mathbf{d}) \tag{154}$$

$$B^* \equiv B \quad \text{excepté:} \quad B^*_r = r(\Pi(\Gamma(B), B)) \tag{155}$$

$\mathbf{d}$ étant un ensemble de données tel que spécifié à l'appendice J.

Comme on l'a spécifié au début du présent travail, Π est la fonction de transition d'état, qui est définie en termes de Ω, la fonction de finalisation de bloc, et de Υ, la fonction d'évaluation d'état, toutes deux maintenant bien définies.

Comme on l'a détaillé précédemment, $\mathbf{R}[n]_{\boldsymbol{\sigma}}$, $\mathbf{R}[n]_{\mathbf{l}}$ et $\mathbf{R}[n]_u$ sont les $n$ièmes états, logs et gaz cumulé correspondants utilisés après chaque transaction ($\mathbf{R}[n]_b$, le quatrième composant du n-uplet, a déjà été défini en termes de logs). Le précédent est simplement défini comme l'état résultant de l'application de la transaction correspondante à l'état résultant de la transaction précédente (ou l'état initial du bloc dans le cas d'une première transaction de ce type):

$$\mathbf{R}[n]_{\boldsymbol{\sigma}} = \begin{cases} \Gamma(B) & \text{si} \quad n < 0 \\ \Upsilon(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \text{sinon} \end{cases} \tag{156}$$

Dans le cas de $B_{\mathbf{R}}[n]_u$, nous prenons une approche similaire en définissant chaque élément comme le gaz utilisé en évaluant la transaction correspondante ajouté à l'élément précédent (ou zéro s'il s'agit du premier), nous donnant un total en cours;

$$\mathbf{R}[n]_u = \begin{cases} 0 & \text{si} \quad n < 0 \\ \Upsilon^g(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \\ \quad +\mathbf{R}[n-1]_u & \text{sinon} \end{cases} \tag{157}$$

Pour $\mathbf{R}[n]_{\mathbf{l}}$, nous utilisons la fonction $\Upsilon^{\mathbf{l}}$ que nous avons commodément définie dans la fonction d'exécution de transaction.

$$\mathbf{R}[n]_{\mathbf{l}} = \Upsilon^{\mathbf{l}}(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) \tag{158}$$

Enfin, nous définissons Π comme le nouvel état étant donnée la fonction de récompense de bloc Ω appliquée à l'état résultant de la transaction finale, $\ell(B_{\mathbf{R}})_{\boldsymbol{\sigma}}$:

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\mathbf{R})_{\boldsymbol{\sigma}}) \tag{159}$$

Donc le mécanisme de transition de bloc, moins `PoW`, la fonction de preuve de travail, est défini.

11.5. **Preuve de travail de minage.** La preuve de travail de minage (PoW) existe comme nonce cryptographique sécurisé qui prouve au-delà du doute raisonnable qu'un montant particulier de calcul a été dépensé dans la détermination de telle valeur n d'un token. Elle est utilisée pour renforcer la sécurité de la blockchain en donnant signification et crédibilité à la notion de difficulté (et, par extension, à la difficulté totale). Cependant, comme miner de nouveaux blocs apporte une récompense, la preuve de travail ne fonctionne pas seulement comme méthode de de sécurisation de la confiance dans le statut normatif de la blockchain pour le futur, mais aussi comme un moyen de distribution de la richesse.

C'est pour ces deux raisons que deux buts importants sont assignés à la preuve de travail; premièrement, elle doit être aussi accessible que possible au plus grand nombre de personnes possibles. Le besoin de, ou rétribution par, des matériels spécialisés et spécifiques doit être minimisé. Ceci rend le modèle de distribution aussi ouvert que possible et, dans l'idéal, fait de l'acte de miner une simple transformation de l'électricité en Ether à un taux à peu près identique partout dans le monde.

Deuxièmement, il ne doit pas être possible de réaliser des profits super-linéaires, surtout avec une barrière à l'entrée importante. Un tel mécanisme accorde à un adversaire fortuné un gain fâcheux de pouvoir de minage sur le réseau total et ainsi lui donne une récompense super-linéaire (et détourne de ce fait la distribution en sa faveur), réduisant en outre la sécurité du réseau.

L'un fléau du monde Bitcoin provient des équipements ASIC. Ce sont des équipements informatiques spécialisés qui sont conçus pour effectuer une tâche unique. Dans le cas de Bitcoin cette tâche est la fonction SHA256. Comme les ASICs existent pour une fonction de preuve de travail, les deux objectifs ci-dessus sont menacés. C'est pourquoi une fonction de preuve de travail ASIC-résistante (c'est-à-dire difficile ou économiquement inefficiente dans ces équipements informatiques spécialisés) serait une panacée.

Il y a deux moyens d'obtenir l'ASIC-résistance; le premier est de le rendre séquentiel par exigence en mémoire, i.e. composer la fonction de façon à ce que la détermination du nonce requière une telle quantité de mémoire et de bande passante que la mémoire ne puisse être utilisée en parallèle pour découvrir simultanément de multiples nonces. Le second est de rendre le type de calcul demandé d'ordre général; la signification de «matériel spécialisé» pour un calcul d'ordre général est, naturellement, matériel généraliste, et on peut escompter que les ordinateurs de bureau entrent dans une catégorie proche. Pour la version 1.0 d'Ethereum nous avons choisi le premier moyen.

Plus formellement, la fonction de preuve de travail prend la forme de PoW telle que:

(160)
$$m = H_m \quad \wedge \quad n \leqslant \frac{2^{256}}{H_d} \quad \text{with} \quad (m, n) = \mathtt{PoW}(H_{\mathbb{H}}, H_n, \mathbf{d})$$

Où $H_{\mathbb{H}}$ est le nouvel en-tête de bloc mais *sans* le nonce et les composants mix-hash; H_n est le nonce du header; $\mathbf{d}$ est un grand ensemble de données nécessaire pour calculer le mixHash et H_d est la difficulté du nouveau block (i.e. la difficulté de la section 10). PoW est la fonction preuve de travail qui produit un tableau avec le premier élément comme mixhash et le deuxième élément un nombre pseudo-aléatoire cryptographiquement dépendant de H et $\mathbf{d}$. L'algorithme sous-jacent se nomme Ethash et est décrit ci-dessous.

11.5.1. *Ethash.* Ethash est l'algorithme de preuve de travail pour la version 1.0 d'Ethereum. C'est la dernière version du Dagger-Hashimoto, présenté par Buterin [2013b] et Dryja [2014], bien qu'il ne puisse plus être appelé ainsi puisque nombre des fonctionnalités de ces deux algorithmes ont été grandement changées ces derniers mois de recherche et développement. Le comportement général de l'algorithme est le suivant:

Il existe une graine cryptographique, ou *seed*, qui peut être calculée pour chaque bloc en examinant les en-têtes jusqu'à ce point. A partir de la graine, on peut calculer un cache pseudo-aléatoire, de $J_{cacheinit}$ octets de taille initiale. Les clients légers stockent le cache. Depuis le cache on peut générer un ensemble de données, de $J_{datasetinit}$ octets de taille initiale, avec la propriété que chaque élément de l'ensemble de données dépend de seulement un petit nombre d'éléments du cache. Les clients complets et les mineurs stockent l'ensemble des données. L'ensemble des données croît linéairement avec le temps.

Le minage consiste à sélectionner des tranches de l'ensemble des données et à en faire des empreintes. La vérification peut être faite avec peu de mémoire en utilisant le cache pour régénérer les éléments spécifiques de l'ensemble des données dont vous avez besoin, et ainsi vous n'avez besoin que de stocker le cache. L'ensemble des données est mis à jour tous les J_{epoch} blocs, donc la grande majorité de l'effort de minage sera de lire les données, et non de les changer. Les paramètres cités comme l'algorithme lui-même sont décrits en annexe J.

12. Implémentation des contrats

IL existe plusieurs modèles d'ingénierie des contrats qui autorisent des fonctionnalités particulièrement utiles; je vais brièvement traiter de deux d'entre eux, les *data feeds* (sources de données) et les nombres aléatoires.

12.1. **Sources de données.** Un contrat de source de données fournit un unique service: il donne accès depuis Ethereum à des informations du monde extérieur. L'exactitude en qualité et en temps de ces informations ne sont pas garanties et c'est la tâche d'un auteur de contrat secondaire – le contrat qui utilise la source de données – de déterminer la confiance qui peut être placée en celle-ci.

Le modèle général implique un unique contrat dans Ethereum qui, à réception d'un appel de message, répond par une information en temps voulu concernant un phénomène extérieur, par exemple la température à New York. Cela peut être implémenté par un contrat qui renvoie cette valeur depuis un point connu dans la mémoire de stockage. Ce dernier doit être bien entendu maintenu avec la température correcte et la seconde partie du modèle implique donc un serveur externe faisant tourner un nœud Ethereum qui, au moment précis de la découverte d'un nouveau bloc, crée une nouvelle transaction valide, envoyée au contrat, mettant la valeur en question dans la mémoire de stockage. Le code du contrat n'accepterait de mises à jour que depuis l'identité contenue dans le serveur.

12.2. **Nombres aléatoires.** La fourniture de nombres aléatoires au sein d'un système déterministe relève naturellement de l'impossible. Nous pouvons cependant aboutir à une approximation en utilisant des données qui sont généralement inconnues au moment de la transaction. Ces données comprennent l'empreinte du bloc, son horodatage et l'adresse de son bénéficiaire. Afin de compliquer le contrôle éventuel de ces valeurs par un mineur malveillant, il faudrait employer l'opération BLOCKHASH pour utiliser des empreintes des 256 blocs précédents comme nombres pseudo-aléatoires. Pour une série de ces nombres, une solution triviale consiste à y ajouter un montant constant et à prendre l'empreinte du résultat.

13. Directions futures

À l'avenir, la base de données des états ne sera pas forcée de maintenir tous les états passés des structures du trie. Elle devra maintenir un âge pour chaque noeud et eventuellement rejeter les noeuds qui ne sont ni assez récents, ni des checkpoints; les checkpoints, ou un ensemble de noeuds dans la base de donnée qui autorisent à traverser un trie d'état de bloc, peuvent être utilisés pour placer une limite maximum sur le volume de calcul nécessaire pour récupérer n'importe quel état à travers la blockchain.

La consolidation de la blockchain peut être utilisée dans le but de réduire le nombre de blocs qu'un client a besoin de télécharger pour agir comme un mineur, un noeud complet. Une archive compressée de la structure du trie à des points donnés dans le temps (par exemple une tous les 10 000 blocs) peut être maintenue par le réseau de pairs, en renouvelant efficacement le bloc genesis. Cela pourrait réduire le volume à télécharger à une simple archive plus une limite maximum de blocs en dur.

Enfin, la compression de la blockchain pourrait peut-être être menée: les noeuds dans un trie d'état qui n'ont pas envoyé/reçu de transaction dans un certain nombre de blocs donné pourraient être rejetés, réduisant les fuites d'ethers ainsi que l'accroissement de la base de donnée d'état.

13.1. **Extensibilité.** L'extensibilité ou passage à l'échelle (*scalability*) demeure un éternel problème. Avec une fonction de changement d'état généraliste, il devient difficile de partitionner et parallèliser les transactions pour appliquer la stratégie consistant à diviser pour régner. Négligée, la dynamique valeur-rang du système devient figée et, quand la valeur moyenne des transactions augmente, les moindres d'entre elles se font ignorer, perdant leur intérêt économique à être intégrées dans la chaîne principale. Cependant, plusieurs stratégies existent ayant le potentiel d'étendre considérablement les capacités de passage à l'échelle du protocole.

Certaines formes de structure hiérarchique, construites soit en combinant plusieurs petites chaînes de poids inférieur dans les blocs principaux, soit en fabriquant le bloc principal par agrégation incrémentale (*via* preuve de travail) d'ensembles de transactions plus petits, peuvent permettre la parallélisation des combinaisons de transactions et des constructions de blocs. La parallélisation peut aussi être mise en œuvre par un ensemble priorisé de blockchains traitées en parallèle, consolidant ainsi chaque bloc et rejetant les transactions surnuméraires ou invalides.

Pour finir, le calcul vérifiable, s'il est rendu suffisamment abordable et efficient, pourrait ouvrir une voie à l'utilisation de la preuve de travail comme vérification de l'état final.

14. Conclusion

J'ai ici présenté, discuté et défini formellement le protocole d'Ethereum. Par ce protocole le lecteur pourra implémenter un noeud sur le réseau Ethereum et joindre les autres dans un système d'exploitation social sûr et décentralisé. Des contrats pourront être créés pour spécifier de manière algorithmique et imposer en autonomie des règles d'échanges.

15. Remerciements

De nombreux remerciements vont à Aeron Buchanan pour l'écriture des révisions Homestead, à Christoph Jentzsch pour la création de l'algorithme Ethash et à Yoichi Hirai pour la plupart des changements à EIP-150. D'importantes mises à jour, des corrections et suggestions utiles ont été faites par bien d'autres membres de l'Ethereum DEV organisation et de la communauté Ethereum au sens large, entre autres Gustav Simonsson, Paweł Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari et, bien sûr, Vitalik Buterin.

16. Disponibilité

Le source de la version originale de cet article est maintenu sur `https://github.com/ethereum/yellowpaper/` et un PDF auto-généré se trouve sur `https://ethereum.github.io/yellowpaper/paper.pdf`.

Le source de la version traduite est maintenu sur `https://github.com/asseth/yellowpaper/blob/french/Paper.tex` et un PDF auto-généré se trouve sur `https://github.com/asseth/yellowpaper/blob/french/Paper.pdf`

References

Jacob Aron. BitCoin software finds new life. *New Scientist*, 213(2847):20, 2012.

Adam Back. Hashcash - Amortizable Publicly Auditable Cost-Functions. 2002. URL {http://www.hashcash.org/papers/amortizable.pdf}.

Roman Boutellier and Mareike Heinzen. Pirates, Pioneers, Innovators and Imitators. In *Growth Through Innovation*, pages 85–96. Springer, 2014.

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2013a. URL {http://ethereum.org/ethereum.html}.

Vitalik Buterin. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative. 2013b. URL {http://vitalik.ca/ethereum/dagger.html}.

Thaddeus Dryja. Hashimoto: I/O bound proof of work. 2014. URL {https://mirrorx.com/files/hashimoto.pdf}.

Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *In 12th Annual International Cryptology Conference*, pages 139–147, 1992.

Phong Vo Glenn Fowler, Landon Curt Noll. Fowler–Noll–Vo hash function. 1991. URL {https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function#cite_note-2}.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004.

Sergio Demian Lerner. Strict Memory Hard Hashing Functions. 2014. URL {http://www.hashcash.org/papers/memohash.pdf}.

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1:2012, 2008.

Meni Rosenfeld. Overview of Colored Coins. 2012. URL {https://bitcoil.co.il/BitcoinX.pdf}.

Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains, 2013. URL {CryptologyePrintArchive,Report2013/881}. http://eprint.iacr.org/.

Simon Sprankel. Technical Basis of Digital Currencies, 2013.

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. Karma: A secure economic framework for peer-to-peer resource sharing, 2003.

J. R. Willett. MasterCoin Complete Specification. 2013. URL {https://github.com/mastercoin-MSC/spec}.

Annexe A. Terminologie

Acteur externe (*external actor*) : Une personne ou une entité susceptible de s'interfacer à un nœud Ethereum, mais extérieur au monde d'Ethereum. Il peut interagir avec Ethereum en déposant des Transactions signées et en inspectant la blockchain et son état associé. Possède un Compte intrinsèque ou plus.

Adresse (*address*) : Un code sur 160 bits utilisé pour identifier les Comptes.

Compte (*account*) : Les Comptes ont une balance intrinsèque et un compteur de transactions maintenu dans l'état d'Ethereum. Ils ont également du Code EVM (éventuellement vide) et un État de stockage qui leur est associé. Bien qu'homogènes, il est admis de distinguer deux types de comptes : ceux avec un Code EVM vide (et dont le solde est contrôlé le cas échéant par une entité externe) et ceux dont le Code EVM associé n'est pas vide (le compte représente donc un Objet Autonome. Chaque compte a une adresse unique qui l'identifie.

Transaction : Des données, signées par un Acteur externe. Elle représente soit un Message soit un nouvel Objet Autonome. Les transactions sont enregistrées dans chaque bloc de la blockchain.

Objet Autonome (*autonomous object*) : Un objet théorique qui n'existe que dans l'état hypothétique d'Ethereum. Possède une adresse intrinsèque et donc un compte associé ; le compte aura un Code EVM associé non vide. N'est incorporé que comme État de Stockage de ce compte.

État de stockage (*storage state*) : L'information spécifique à un Compte donné qui est conservée entre les occurrences de lancement du Code EVM associé au Compte.

Message : Données (en tant qu'ensemble d'octets) et Valeur (spécifiée en Ether) qui sont passées entre deux Comptes, soit par l'opération déterministe d'un Objet Autonome, soit par la signature cryptographiquement sécurisée de la Transaction.

Appel de message (*message call*) : Le fait de passer un message d'un Compte vers un autre. Si le compte de destination est associé à du Code EVM non vide, alors la VM sera démarrée avec l'état de l'Objet en question et le Message donnera lieu à une action. Si l'expéditeur du message est un Objet Autonome, l'Appel passe toute donnée renvoyée de l'opération de la VM.

Gaz (*gas*) : L'unité de coût fondamentale du réseau. Payée exclusivement en Ether (au moment du PoC-4), qui est librement converti en Gaz et inversement selon les besoins. Le Gaz n'existe pas hors du moteur de calcul interne d'Ethereum ; son prix est fixé par la Transaction et les mineurs sont libres d'ignorer les Transactions dont le prix en Gaz est trop bas.

Contrat (*contract*) : Terme informel employé pour signifier à la fois un bout de Code EV qui peut être associé à un compte et un Objet autonome.

Objet (*object*) : Synonyme d'Objet autonome.

App : Une application visible de l'utilisateur hébergée dans le Navigateur Ethereum.

Navigateur Ethereum (*Ethereum Browser*) : (alias *Ethereum Reference Client*) Une interface utilisateur graphique similaire à un navigateur simplifié (à la Chrome) capable d'héberger des applications en mode bac à sable dont le backend repose purement sur le protocole Ethereum.

Ethereum Virtual Machine : (alias EVM) La machine virtuelle qui forme la partie essentielle du modèle d'exécution du Code EVM associé à un Compte.

Ethereum Runtime Environment : (alias ERE) L'environnement qui est fourni à un Objet Autonome en train de s'exécuter dans l'EM. Comprend l'EVM mais aussi la structure de l'état du monde sur laquelle l'EVM se base pour certaines instructions d'E/S, y compris CALL et CREATE.

Code EVM : Le bytecode que l'EVM peut exécuter nativement. Utilisé pour spécifier formellement la signification et les ramifications d'un message vers un Compte.

Assembleur EVM (*EVM assembly*) : La forme humainement lisible du Code EVM.

LLL : textitLisp-like Low-level Language, un langage humainement utilisable pour créer des contrats simples et une boîte à outils de langage généraliste de bas niveau vers laquelle on transcompile.

Annexe B. *Recursive Length Prefix*

Il s'agit d'une méthode de sérialisation pour encoder des données binaires quelle que soit leur structure (tableaux d'octets).

Nous définissons l'ensemble des structures possibles $\mathbb{T}$:

$$\mathbb{T} \equiv \mathbb{L} \cup \mathbb{B} \tag{161}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \wedge \forall_{n<\|\mathbf{t}\|}\ \mathbf{t}[n] \in \mathbb{T}\} \tag{162}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \wedge \forall_{n<\|\mathbf{b}\|}\ \mathbf{b}[n] \in \mathbb{O}\} \tag{163}$$

où $\mathbb{O}$ est l'ensemble d'octets. $\mathbb{B}$ est donc l'ensemble de toutes les séquences d'octets (connues par ailleurs sous le nom de tableaux d'octets, et une feuille si on l'imagine comme un arbre), $\mathbb{L}$ est l'ensemble de toutes les (sous-)structures arborescentes qui ne sont pas une feuille unique (un nœud de branches si on l'imagine comme un arbre) et $\mathbb{T}$ est l'ensemble de tous les tableaux d'octets et les séquences structurales similaires.

Nous définissons la fonction RLP par $\texttt{RLP}$ grâce à deux sous-fonctions, la première gérant l'instance quand la valeur est un tableau d'octets, la seconde quand elle est une séquence de valeurs successives :

$$\texttt{RLP}(\mathbf{x}) \equiv \begin{cases} R_b(\mathbf{x}) & \text{si} \quad \mathbf{x} \in \mathbb{B} \\ R_l(\mathbf{x}) & \text{sinon} \end{cases} \tag{164}$$

Si la valeur à sérialiser est un tableau d'octets, la sérialisation RLP prend l'une de ces trois formes :

- Si le tableau d'octets ne contient qu'un octet unique et que cet octet est inférieur à 128, l'entrée est exactement égale à la sortie.

- Si le tableau d'octets contient moins de 56 octets, la sortie est égale à l'entrée préfixée par l'octet égal à la longueur du tableau d'octets plus 128.
- Sinon, la sortie est égale à l'entrée préfixée par le tableau d'octets de longueur minimale qui, quand interprété comme un entier gros-boutien, est égal à la longueur du tableau d'octets en entrée, lui-même préfixé par le nombre d'octets requis pour encoder fidèlement la valeur de cette longueur plus 183.

Formellement, nous définissons R_b :

$$R_b(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{si} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{sinon et si} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\mathtt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \mathtt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{sinon} \end{cases} \tag{165}$$

$$\mathtt{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{n<\|\mathbf{b}\|} b_n \cdot 256^{\|\mathbf{b}\|-1-n} \tag{166}$$

$$(a) \cdot (b, c) \cdot (d, e) = (a, b, c, d, e) \tag{167}$$

Donc BE est la fonction d'expansion d'une valeur entière positive vers un tableau d'octets gros-boutien de longueur minimale et l'opérateur point effectue la concaténation de la séquence.

Si en revanche la valeur à sérialiser est une séquence d'éléments autres, la sérialisation RLP prend l'une de ces deux formes :

- Si les sérialisations concaténées de chaque élément contenu fait moins de 56 octets de longueur, la sortie est égale à cette concaténation préfixée par l'octet égal à la longueur de ce tableau d'octets plus 192.
- Sinon, la sortie est égale aux sérialisations concaténées préfixées par le tableau d'octets de longueur minimale qui, quand on l'interprète comme un entier gros-boutien, est égal à la longueur du tableau d'octets des sérialisations concaténées, lui-même préfixé par le nombre d'octets requis pour encoder fidèlement la valeur de cette longueur plus 247.

Nous terminons donc en définissant formellement R_l :

$$R_l(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{si} \quad \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\mathtt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \mathtt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{sinon} \end{cases} \tag{168}$$

$$s(\mathbf{x}) \equiv \mathtt{RLP}(\mathbf{x}_0) \cdot \mathtt{RLP}(\mathbf{x}_1)... \tag{169}$$

Si RLP est utilisé pour encoder un scalaire, défini comme un entier positif ($\mathbb{P}$ ou tout x pour $\mathbb{P}_x$), il doit être spécifié comme le plus petit tableau d'octets tel que son interprétation gros-boutienne est égale. Donc, le RLP d'un entier positif i est défini par :

$$\mathtt{RLP}(i : i \in \mathbb{P}) \equiv \mathtt{RLP}(\mathtt{BE}(i)) \tag{170}$$

Quand on interprète les données RLP, si un fragment inattendu est décodé comme un scalaire et que des zéros préfixent une séquence d'octets, les clients doivent le considérer comme non-canonique et le traiter de la même manière que les autres données RLP invalides en le rejetant complètement.

Il n'existe pas de format d'encodage canonique pour les valeurs signées ou en virgule flottante.

Annexe C. Encodage *hex-prefix*

L'encodage *hex-prefix* est une méthode efficace d'encodage d'un nombre quelconque de *nibbles* ou quartets dans un tableau d'octets. Il est capable de stocker un indicateur supplémentaire qui, quand on l'emploie dans le contexte d'un trie ou arbre préfixe (le seul contexte dans lequel il est utilisé), permet de discriminer les types de nœuds.

Il est défini par la fonction HP qui fait correspondre une séquence de quartets (représentés par l'ensemble $\mathbb{Y}$) avec une valeur booléenne à une séquence d'octets (représentés par l'ensemble $\mathbb{B}$) :

$$\mathtt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{si} \quad \|\mathbf{x}\| \text{ est pair} \\ (16(f(t) + 1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{sinon} \end{cases} \tag{171}$$

$$f(t) \equiv \begin{cases} 2 & \text{si} \quad t \neq 0 \\ 0 & \text{sinon} \end{cases} \tag{172}$$

Le quartet de poids fort du premier octet contient deux indicateurs : le bit de poids faible code la parité de la longueur et le deuxième bit de poids faible code l'indicateur t. L'octet de poids faible du premier octet est à zéro dans le cas d'un nombre de quartets pair et au premier quartet dans le cas d'un nombre impair. Tous les quartets restants (maintenant un nombre pair) s'adaptent correctement dans les octets restants.

Annexe D. Arbre de Merkle Patricia modifié

L'arbre de Merkle modifié (trie) fournit une structure de données persistante pour faire correspondre des données binaire de longueur quelconque (tableaux d'octets). On le définit en termes d'une structure de données modifiables pour faire correspondre des fragments binaires de 256 bits et des données binaires de longueur quelconque, en général implémentés dans une base de données. L'essence du trie, et son seul prérequis en terme de spécification du protocole,

consiste à fournir une valeur unique qui identifie un ensemble donné de paires clef-valeur, qui peut être soit une séquence de 32 octets, soit la séquence d'octets vide. On laisse libre le choix de l'implémentation du stockage et de la maintenance de la structure du trie de manière à respecter le protocole de manière exacte et efficiente.

Formellement, nous posons la valeur de sortie $\mathfrak{I}$, un ensemble contenant les paires de séquences d'octets :

$$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{173}$$

Quand on considère une telle séquence, nous utilisons la notation numérique souscrite courante pour nous référer à la clef ou à la valeur d'un n-uplet, comme ici ;

$$\forall_{I \in \mathfrak{I}} I \equiv (I_0, I_1) \tag{174}$$

Toute série d'octets peut également être vue de façon triviale comme une série de quartets, selon la notation petit ou grand-boutienne choisie ; nous prenons ici grand-boutienne. Donc :

$$y(\mathfrak{I}) = \{(\mathbf{k}'_0 \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}'_1 \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{175}$$

$$\forall_n \quad \forall_{i:i<2\|\mathbf{k}_n\|} \quad \mathbf{k}'_n[i] \equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{si } i \text{ est pair} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{sinon} \end{cases} \tag{176}$$

Nous définissons la fonction TRIE, qui donne la racine du trie qui représente cet ensemble quand il est codé dans cette structure :

$$\texttt{TRIE}(\mathfrak{I}) \equiv \texttt{KEC}(c(\mathfrak{I}, 0)) \tag{177}$$

Nous posons également une fonction n, la fonction de ***cap*** de nœuds du trie. Au moment de composer un nœud, nous utilisons RLP pour coder la structure. Afin de réduire la complexité du stockage, pour les nœuds dont le RLP composé fait moins de 32 octets, nous stockons celui-ci directement ; pour ceux les plus grands, nous supposons une prescience du tableau d'octets dont l'évaluation de l'empreinte Keccak donne notre référence. Nous définissons donc en termes de c, la fonction de composition de nœud :

$$n(\mathfrak{I}, i) \equiv \begin{cases} () & \text{si } \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{si } \|c(\mathfrak{I}, i)\| < 32 \\ \texttt{KEC}(c(\mathfrak{I}, i)) & \text{sinon} \end{cases} \tag{178}$$

D'une façon similaire à un arbre radix, quand le trie est traversé de la racine à une feuille, on peut construire une paire clef-valeur unique. La clef est accumulée pendant la traversée en acquérant un quartet par nœud de branches (comme pour un arbre radix). À la différence d'un arbre radix, dans le cas de clefs multiples partageant le même préfixe ou dans le cas d'une clef unique possédant un suffixe unique, deux nœuds d'optimisation sont fournis. Donc, en traversant, on peut acquérir potentiellement plusieurs quartets de chacun des deux autres types de nœuds, extension et feuille. Il existe trois sortes de nœuds dans le trie :

Feuille: Une structure à deux éléments dont le premier correspond aux quartets dans la clef qui n'ont pas encore été pris en compte par l'accumulation des clefs et des branches traversées depuis la racine. La méthode d'encodage hex-prefix est utilisée et le second paramètre de la fonction doit être à *true*.

Extension: Une structure à deux éléments dont le premier correspond à une série d'octets de taille supérieure à un, qui sont partagés par au moins deux clefs distinctes après l'accumulation de clefs et de branches de nibbles traversées depuis la racine. La méthode d'encodage hex-prefix est utilisée et le second paramètre de la fonction doit être à *false*.

Branche: Une structure à 17 éléments dont les seize premiers correspondent à chacune de seize valeurs de quartet possible pour les clefs à ce point de la traversée. Le 17^{e} élément est utilisé dans le cas où il s'agit d'un nœud de terminaison et donc une clef étant finie à ce point de la traversée.

Une branche n'est donc utilisée que quand c'est nécessaire ; aucun nœud de branche ne peut exister qui ne contiendrait qu'une seule entrée non nulle. Nous pouvons définit formellement cette structure avec la fonction de composition structurelle c :

$$c(\mathfrak{I}, i) \equiv \begin{cases} \texttt{RLP}\Big((\texttt{HP}(I_0[i..(\|I_0\|-1)], true), I_1)\Big) & \text{si } \|\mathfrak{I}\| = 1 \quad \text{où } \exists I : I \in \mathfrak{I} \\ \texttt{RLP}\Big((\texttt{HP}(I_0[i..(j-1)], false), n(\mathfrak{I}, j))\Big) & \text{si } i \neq j \quad \text{où } j = \arg\max_x : \exists \mathbf{l} : \|\mathbf{l}\| = x : \forall_{I \in \mathfrak{I}} : I_0[0..(x-1)] = \mathbf{l} \\ \texttt{RLP}\Big((u(0), u(1), ..., u(15), v)\Big) & \text{sinon où } u(j) \equiv n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i+1) \\ & \qquad v = \begin{cases} I_1 & \text{si } \exists I : I \in \mathfrak{I} \wedge \|I_0\| = i \\ () & \text{sinon} \end{cases} \end{cases} \tag{179}$$

D.1. **Base de données trie.** Aucune supposition explicite n'est donc faite concernant les données stockées ou non, puisque cette considération relève de l'implémentation ; nous nous contentons de définir la fonction d'identité faisant correspondre l'ensemble de clefs-valeurs $\mathfrak{I}$ à une empreinte de 32 octets et nous posons qu'une seule telle empreinte existe pour tout $\mathfrak{I}$, ce qui, bien que n'étant pas strictement vrai, est exact avec une précision acceptable étant donnée la résistance à la collision de l'empreinte Keccak. En réalité, une implémentation raisonnable ne recalculera pas complètement l'empreinte racine du trie pour chaque ensemble.

Cette implémentation maintiendra une base de données de nœuds déterminée à partir du calcul de plusieurs tries ou, plus formellement, elle va mémoriser la fonction c. Cette stratégie utilise la nature du trie tant pour rappeler facilement le contenu de tout ensemble de clefs-valeurs que pour stocker plusieurs de ces ensembles de manière très efficiente. En raison de la relation de dépendances, les preuves de Merkle doivent être construites avec un prérequis d'espace $O(\log N)$ qui peut démontrer qu'une feuille donnée doit exister dans un trie d'une empreinte racine de trie donnée.

Annexe E. Contrats précompilés

Pour chaque contrat précompilé, nous nous servons d'une fonction template, Ξ_{PRE}, qui implémente la vérification du *out-of-gas*.

$$\Xi_{\text{PRE}}(\boldsymbol{\sigma}, g, I) \equiv \begin{cases} (\varnothing, 0, A^0, ()) & \text{si} \quad g < g_r \\ (\boldsymbol{\sigma}, g - g_r, A^0, \mathbf{o}) & \text{sinon} \end{cases} \tag{180}$$

Les contrats précompilés utilisent chacun ces définitions et fournissent les spécifications de $\mathbf{o}$ (les données en sortie) et g_r, les exigences en gaz.

Pour la fonction d'exécution de VM du recouvrement de courbe elliptique DSA, nous définissons également $\mathbf{d}$, les données en entrée, bien définies pour une longueur infinie en ajoutant des zéros le cas échéant. Il est important de noter que, dans le cas d'une signature invalide ($\texttt{ECDSARECOVER}(h, v, r, s) = \varnothing$), il n'y a pas de sortie.

$$\Xi_{\text{ECREC}} \equiv \Xi_{\text{PRE}} \quad \text{où:} \tag{181}$$

$$g_r = 3000 \tag{182}$$

$$|\mathbf{o}| = \begin{cases} 0 & \text{si} \quad \texttt{ECDSARECOVER}(h, v, r, s) = \varnothing \\ 32 & \text{sinon} \end{cases} \tag{183}$$

$$\text{si} \quad |\mathbf{o}| = 32 : \tag{184}$$

$$\mathbf{o}[0..11] = 0 \tag{185}$$

$$\mathbf{o}[12..31] = \texttt{KEC}\big(\texttt{ECDSARECOVER}(h, v, r, s)\big)[12..31] \quad \text{où:} \tag{186}$$

$$\mathbf{d}[0..(|I_{\mathbf{d}}| - 1)] = I_{\mathbf{d}} \tag{187}$$

$$\mathbf{d}[|I_{\mathbf{d}}|..] = (0, 0, ...) \tag{188}$$

$$h = \mathbf{d}[0..31] \tag{189}$$

$$v = \mathbf{d}[32..63] \tag{190}$$

$$r = \mathbf{d}[64..95] \tag{191}$$

$$s = \mathbf{d}[96..127] \tag{192}$$

Les deux fonctions de hachage, RIPEMD-160 et SHA2-256, sont plus trivialement définies comme une opération presque traversante. Leur consommation en gaz dépend de la taille des données en entrée, un facteur arrondi au nombre de mots immédiatement supérieur.

$$\Xi_{\text{SHA256}} \equiv \Xi_{\text{PRE}} \quad \text{où:} \tag{193}$$

$$g_r = 60 + 12\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{194}$$

$$\mathbf{o}[0..31] = \texttt{SHA256}(I_{\mathbf{d}}) \tag{195}$$

$$\Xi_{\text{RIP160}} \equiv \Xi_{\text{PRE}} \quad \text{où:} \tag{196}$$

$$g_r = 600 + 120\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{197}$$

$$\mathbf{o}[0..11] = 0 \tag{198}$$

$$\mathbf{o}[12..31] = \texttt{RIPEMD160}(I_{\mathbf{d}}) \tag{199}$$

(200)

Pour les besoins présents, nous supposons que nous avons des fonctions cryptographiques standard bien définies pour RIPEMD-160 et SHA2-256 de la forme :

$$\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{32} \tag{201}$$

$$\texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{20} \tag{202}$$

Enfin, le quatrième contrat, la fonction d'identité Ξ_{ID}, se contente de définir la sortie d'après l'entrée :

$$\Xi_{\text{ID}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{203}$$

$$g_r = 15 + 3\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{204}$$

$$\mathbf{o} = I_{\mathbf{d}} \tag{205}$$

Annexe F. Signature de transactions

La méthode de signature de transactions est similaire aux « signatures de style Electrum » ; elle utilise la courbe SECP-256k1 décrite par Gura et al. [2004].

On suppose que l'expéditeur possède une clef privée valide p_r, qui est un entier positif sélectionné au hasard (représenté par un tableau d'octets de longueur 32 sous forme gros-boutienne) dans l'intervalle $[1, \texttt{secp256k1n} - 1]$.

Nous posons les fonctions `ECDSASIGN`, `ECDSARESTORE` et `ECDSAPUBKEY`. Celles-ci sont formellement définies dans la littérature.

$$\texttt{ECDSAPUBKEY}(p_r \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \tag{206}$$

$$\texttt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_r \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \tag{207}$$

$$\texttt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \tag{208}$$

où p_u est la clef publique, que l'on suppose un tableau d'octets de taille 64 (formé à partir de la concaténation de deux entiers positifs chacun $< 2^{256}$) et p_r est la clef privée, un tableau de taille 32 (ou un simple entier positif dans l'intervalle susmentionné). On suppose que v est l'« identifiant de recouvrement », une valeur sur un octet spécifiant le signe et la finitude du point de la courbe ; cette valeur se trouve dans l'intervalle $[27, 30]$ mais nous déclarons les deux possibilités supérieures, représentant des valeurs infinies, invalides.

Nous déclarons qu'une signature est invalide à moins que toutes les conditions suivantes soient vraies :

$$0 < r < \texttt{secp256k1n} \tag{209}$$

$$0 < s < \begin{cases} \texttt{secp256k1n} & \text{si} \quad H_i < N_H \\ \texttt{secp256k1n} \div 2 & \text{sinon} \end{cases} \tag{210}$$

$$v \in \{27, 28\} \tag{211}$$

où :

$$\texttt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \tag{212}$$

Pour une clef privée donnée, p_r, l'adresse Ethereum $A(p_r)$ (une valeur sur 160 bits) à laquelle elle correspond est définie comme les 160 bits de droite de l'empreinte Keccak de la clef publique ECDSA correspondante :

$$A(p_r) = \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSAPUBKEY}(p_r)\big)\big) \tag{213}$$

L'empreinte du message $h(T)$ à signer est l'empreinte Keccak de la transaction sans les trois derniers composants de la signature, formellement décrits comme T_r, T_s et T_w :

$$L_S(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_\mathbf{i}) & \text{si } T_t = 0 \\ (T_n, T_p, T_g, T_t, T_v, T_\mathbf{d}) & \text{sinon} \end{cases} \tag{214}$$

$$h(T) \equiv \texttt{KEC}(L_S(T)) \tag{215}$$

La transaction signée $G(T, p_r)$ est définie par :

$$G(T, p_r) \equiv T \quad \text{excepté:} \tag{216}$$

$$(T_w, T_r, T_s) = \texttt{ECDSASIGN}(h(T), p_r) \tag{217}$$

Nous pouvons alors définir la fonction de l'expéditeur S de la transaction par :

$$S(T) \equiv \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSARECOVER}(h(T), T_w, T_r, T_s)\big)\big) \tag{218}$$

L'assertion selon laquelle l'expéditeur d'une transaction signée est égale à l'adresse du signataire devrait être évidente :

$$\forall T : \forall p_r : S(G(T, p_r)) \equiv A(p_r) \tag{219}$$

Annexe G. Grille tarifaire

La grille tarifaire G est un n-uplet de 31 valeurs scalaires correspondant aux coûts relatifs, en gaz, d'un certain nombre d'opérations abstraites qu'une transaction peut effectuer.

Nom	Valeur	Description*
G_{zero}	0	Rien n'est payé pour les opérations de l'ensemble W_{zero}.
G_{base}	2	Montant en gaz à payer pour les opérations de l'ensemble W_{base}.
$G_{verylow}$	3	Montant en gaz à payer pour les opérations de l'ensemble $W_{verylow}$.
G_{low}	5	Montant en gaz à payer pour les opérations de l'ensemble W_{low}.
G_{mid}	8	Montant en gaz à payer pour les opérations de l'ensemble W_{mid}.
G_{high}	10	Montant en gaz à payer pour les opérations de l'ensemble W_{high}.
$G_{extcode}$	700	Montant en gaz à payer pour les opérations de l'ensemble $W_{extcode}$.
$G_{balance}$	400	Montant en gaz à payer pour une opération BALANCE.
G_{sload}	200	Payé pour une opération SLOAD.
$G_{jumpdest}$	1	Payé pour une opération JUMPDEST.
G_{sset}	20000	Payé pour une opération SSTORE quand la l'état zéro de la valeur du stockage passe de non-zéro à zéro.
G_{sreset}	5000	Payé pour une opération SSTORE quand la valeur du stockage reste ou passe à zéro.
R_{sclear}	15000	Remboursement (ajouté au compteur de remboursement) quand la valeur du stockage passe de zéro à non-zéro.
$R_{selfdestruct}$	24000	Remboursement (ajouté au compteur de remboursement) pour l'autodestruction d'un compte.
$G_{selfdestruct}$	5000	Montant en gaz à payer pour une opération SELFDESTRUCT.
G_{create}	32000	Payé pour une opération CREATE.
$G_{codedeposit}$	200	Payé par octet pour qu'une opération CREATE réussise à placer le code dans l'état.
G_{call}	700	Payé pour une opération CALL.
$G_{callvalue}$	9000	Payé pour un transfert de valeur non nulle faisant partie de l'opération CALL.
$G_{callstipend}$	2300	Une allocation pour le contrat appelé, soustraite de $G_{callvalue}$ pour le transfert d'une valeur non nulle.
$G_{newaccount}$	25000	Payé pour une opération CALL ou SELFDESTRUCT qui crée un compte.
G_{exp}	10	Paiement partiel pour une opération EXP.
$G_{expbyte}$	50	Paiement partiel quand il est multiplié par $\lceil \log_{256}(exposant) \rceil$ pour l'opération EXP.
G_{memory}	3	Payé pour tout mot additionnel en augmentant la mémoire.
G_{txcreate}	32000	Payé par toutes les transactions de création de contrat après la *transition Homestead*.
$G_{txdatazero}$	4	Payé pour tout octet de données ou de code à zéro pour une transaction.
$G_{txdatanonzero}$	68	Payé pour tout octet de données ou de code non nul pour une transaction.
$G_{transaction}$	21000	Payé pour toute transaction.
G_{log}	375	Paiement partiel pour une opération LOG.
$G_{logdata}$	8	Payé pour chaque octet de données d'une opération LOG.
$G_{logtopic}$	375	Payé pour chaque sujet d'une opération LOG.
G_{sha3}	30	Payé pour chaque opération SHA3.
$G_{sha3word}$	6	Payé pour chaque mot (arrondi à la valeur supérieure) pour les données en entrée d'une opération SHA3.
G_{copy}	3	Paiement partiel pour les opérations *COPY, multiplié par le nombre de mots copiés, arrondi à la valeur supérieure.
$G_{blockhash}$	20	Paiement pour une opération BLOCKHASH.

Annexe H. Spécification de la machine virtuelle

Quand on interprète des valeurs binaires sur 256 bits comme entiers, la représentation est gros-boutienne.

Quand une donnée machine sur 256 bits est convertie depuis et vers une adresse ou une empreinte sur 160 bits, les 20 octets de droite (ici de poids faible) sont utilisés et les 12 de gauche sont supprimés ou mis à zéro, ce qui rend les valeurs entières (les octets étant interprétés en gros-boutien) sont équivalents.

H.1. **Coût en gaz.** La fonction générale de coût en gaz, C, est définie par :

(220)
$$C(\boldsymbol{\sigma},\boldsymbol{\mu},I) \equiv C_{mem}(\boldsymbol{\mu}'_i) - C_{mem}(\boldsymbol{\mu}_i) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma},\boldsymbol{\mu}) & \text{si} \quad w = \text{SSTORE} \\ G_{exp} & \text{si} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{exp} + G_{expbyte} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{si} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{verylow} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{si} \quad w = \text{CALLDATACOPY} \vee \text{CODECOPY} \\ G_{extcode} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{si} \quad w = \text{EXTCODECOPY} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{si} \quad w = \text{LOG0} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{logtopic} & \text{si} \quad w = \text{LOG1} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{logtopic} & \text{si} \quad w = \text{LOG2} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{logtopic} & \text{si} \quad w = \text{LOG3} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{logtopic} & \text{si} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma},\boldsymbol{\mu}) & \text{si} \quad w = \text{CALL} \vee \text{CALLCODE} \vee \text{DELEGATECALL} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma},\boldsymbol{\mu}) & \text{si} \quad w = \text{SELFDESTRUCT} \\ G_{create} & \text{si} \quad w = \text{CREATE} \\ G_{sha3} + G_{sha3word} \lceil \mathbf{s}[1] \div 32 \rceil & \text{si} \quad w = \text{SHA3} \\ G_{jumpdest} & \text{si} \quad w = \text{JUMPDEST} \\ G_{sload} & \text{si} \quad w = \text{SLOAD} \\ G_{zero} & \text{si} \quad w \in W_{zero} \\ G_{base} & \text{si} \quad w \in W_{base} \\ G_{verylow} & \text{si} \quad w \in W_{verylow} \\ G_{low} & \text{si} \quad w \in W_{low} \\ G_{mid} & \text{si} \quad w \in W_{mid} \\ G_{high} & \text{si} \quad w \in W_{high} \\ G_{extcode} & \text{si} \quad w \in W_{extcode} \\ G_{balance} & \text{si} \quad w = \text{BALANCE} \\ G_{blockhash} & \text{si} \quad w = \text{BLOCKHASH} \end{cases}$$

(221)
$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{si} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{sinon} \end{cases}$$

où :

(222)
$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

avec C_{CALL}, $C_{\text{SELFDESTRUCT}}$ et C_{SSTORE} tels qu'ils sont spécifiés dans la section correspondante ci-dessous. Nous définissons les sous-ensembles suivants d'instructions :

$W_{zero} = \{\text{STOP, RETURN}\}$
$W_{base} = \{\text{ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, POP, PC, MSIZE, GAS}\}$
$W_{verylow} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}\}$
$W_{low} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}\}$
$W_{mid} = \{\text{ADDMOD, MULMOD, JUMP}\}$
$W_{high} = \{\text{JUMPI}\}$
$W_{extcode} = \{\text{EXTCODESIZE}\}$

Notons le composant de coût mémoire, donné par le produit de G_{memory} et du maximum de 0 & le plafond du nombre de mots dépassant le nombre actuel de mots, $\boldsymbol{\mu}_i$ afin que tous les accès référencent de la mémoire valide que ce soit pour la lecture ou l'écriture. Ces accès doivent concerner un nombre d'octets non nul.

Le référencement d'un intervalle de longueur zéro (c'est-à-dire en essayant de le passer comme intervalle en entrée d'un CALL) ne demande pas à la mémoire d'être étendue au début de l'intervalle. $\boldsymbol{\mu}'_i$ est défini comme ce nouveau nombre maximum de mots de mémoire active ; les cas spéciaux sont donnés quand ces deux-ci ne sont pas égaux.

Notons également que C_{mem} est la fonction de coût mémoire (la fonction d'expansion étant la différence entre les coûts avant et après). C'est un polynôme dont le coefficient de plus haut degré est divisé et tronqué, et il est donc linéaire jusqu'à 724 octets de mémoire utilisée, après quoi il coûte substantiellement plus.

En définissant l'ensemble d'instructions, nous avons défini l'expansion mémoire pour la fonction d'intervalle, M, d'où :

(223)
$$M(s,f,l) \equiv \begin{cases} s & \text{si} \quad l = 0 \\ \max(s, \lceil (f+l) \div 32 \rceil) & \text{sinon} \end{cases}$$

Une autre fonction utile est la fonction L « tout sauf un 64^{e} » définie par :

$$L(n) \equiv n - \lfloor n/64 \rfloor \tag{224}$$

H.2. **Jeu d'instructions.** Comme spécifié auparavant dans la section 9, toutes ces définitions prennent place dans le contexte final présent. En particulier nous assumons que O est la fonction d'itération d'état de l'EVM et définissons les termes s'appliquant à l'état du cycle suivant $(\boldsymbol{\sigma}', \boldsymbol{\mu}')$ tels que :

$$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{avec exceptions, comme mentionné} \tag{225}$$

Ici sont données pour chaque instruction les diverses exceptions aux règles de transition d'état données dans la section 9, avec les définitions additionnelles de J et C spécifiques à cette instruction. Pour chaque instruction est aussi spécifié α, les éléments additionnels placés sur la pile et δ, les éléments ôtés de la pile, comme définie dans la section 9.

0s : Opérations d'arrêt et arithmétiques

Toutes les valeurs arithmétiques sont modulo 2^{256} sauf mention exprès.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Arrête l'exécution.
0x01	ADD	2	1	Addition. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x03	SUB	2	1	Soustraction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] - \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x04	DIV	2	1	Division entière. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{sinon} \end{cases}$
0x05	SDIV	2	1	Division entière signée (tronquée). $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = -2^{255} \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = -1 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1]) \lfloor \lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert \rfloor & \text{sinon} \end{cases}$ Où toutes les valeurs sont traitées comme des entiers de 256 bits signés en complément à deux. Noter la sémantique de dépassement quand -2^{255} est négativé.
0x06	MOD	2	1	Modulo. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{sinon} \end{cases}$
0x07	SMOD	2	1	Modulo signé. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0]) \lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \rvert \bmod \lvert \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert & \text{sinon} \end{cases}$ Où toutes les valeurs sont traitées comme des entiers de 256 bits signés en complément à deux.
0x08	ADDMOD	3	1	Addition modulo. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{sinon} \end{cases}$ Les calculs intermédiaires de cette opération ne sont pas modulo 2^{256}.
0x09	MULMOD	3	1	Multiplication modulo. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{sinon} \end{cases}$ Les calculs intermédiaires de cette opération ne sont pas modulo 2^{256}.
0x0a	EXP	2	1	Exponentielle. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]^{\boldsymbol{\mu}_{\mathbf{s}}[1]}$
0x0b	SIGNEXTEND	2	1	Extension signée en complément à deux. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_t & \text{si} \quad i \leqslant t \quad \text{where } t = 256 - 8(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \\ \boldsymbol{\mu}_{\mathbf{s}}[1]_i & \text{sinon} \end{cases}$

$\boldsymbol{\mu}_{\mathbf{s}}[x]_i$ donne le $i$ème bit (partant de zéro) de $\boldsymbol{\mu}_{\mathbf{s}}[x]$

Value	Mnemonic	δ	α	Description
				10s : Comparaison & opérations bit à bit
0x10	LT	2	1	Inférieur à. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{sinon} \end{cases}$
0x11	GT	2	1	Supérieur à. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{sinon} \end{cases}$
0x12	SLT	2	1	Inférieur à signé. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{sinon} \end{cases}$ Où toutes les valeurs sont traitées comme des entiers de 256 bits signés en complément à deux.
0x13	SGT	2	1	Supérieur à signé. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{sinon} \end{cases}$ Où toutes les valeurs sont traitées comme des entiers de 256 bits signés en complément à deux.
0x14	EQ	2	1	Égalité $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{sinon} \end{cases}$
0x15	ISZERO	1	1	Opérateur de négation simple. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = 0 \\ 0 & \text{sinon} \end{cases}$
0x16	AND	2	1	Opérateur AND bit à bit. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \wedge \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x17	OR	2	1	Opérateur OR bit à bit. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \vee \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x18	XOR	2	1	Opérateur XOR bit à bit. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \oplus \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x19	NOT	1	1	Opérateur NOT bit à bit. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} 1 & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[0]_i = 0 \\ 0 & \text{sinon} \end{cases}$
0x1a	BYTE	2	1	Récupère un octet d'un mot. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i+8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{si} \quad i < 8 \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{sinon} \end{cases}$ Pour le Nème octet, on compte depuis la gauche (i.e. N=0 serait le plus significatif en grand-boutien).
				20s : SHA3
Value	**Mnemonic**	δ	α	**Description**
0x20	SHA3	2	1	Calcule l'empreinte Keccak-256. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{Keccak}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

30s : Informations d'environnement

Value	Mnemonic	δ	α	Description
0x30	ADDRESS	0	1	Obtient l'adresse du compte en cours d'exécution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_a$
0x31	BALANCE	1	1	Obtient le solde du compte en argument. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_b & \text{si} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{sinon} \end{cases}$
0x32	ORIGIN	0	1	Obtient l'adresse originelle d'exécution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_o$ C'est l'émetteur de la transaction de départ ; ce n'est jamais un compte avec du code associé non vide.
0x33	CALLER	0	1	Obtient l'adresse appelante. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_s$ Ceci est l'adresse directement responsable de l'exécution.
0x34	CALLVALUE	0	1	Obtient la valeur déposée par l'instruction/transaction responsable de cette exécution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_v$
0x35	CALLDATALOAD	1	1	Obtient les données en entrée de l'environnement courant. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0]+31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{si} \quad x \geqslant \|I_{\mathbf{d}}\|$ Ceci se rapporte aux données passées en entrée du message d'appel de l'instruction ou transaction.
0x36	CALLDATASIZE	0	1	Obtient la taille des données en entrée de l'environnement courant. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{d}}\|$ Ceci se rapporte aux données passées en entrée du message d'appel de l'instruction ou transaction.
0x37	CALLDATACOPY	3	0	Copie les données en entrée de l'environnement courant. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]+i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1]+i] & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1]+i < \|I_{\mathbf{d}}\| \\ 0 & \text{sinon} \end{cases}$ Les additions en $\boldsymbol{\mu}_{\mathbf{s}}[1]+i$ ne sont pas sujettes au modulo 2^{256}. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ Ceci se rapporte aux données passées en entrée du message d'appel de l'instruction ou transaction.
0x38	CODESIZE	0	1	Obtient la taille du code dans l'environnement courant. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{b}}\|$
0x39	CODECOPY	3	0	Copie le code tournant en entrée de l'environnement courant vers la mémoire. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]+i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1]+i] & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1]+i < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{sinon} \end{cases}$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
0x3a	GASPRICE	0	1	Obtient le prix du gaz dans l'environnement courant. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_p$ C'est le prix de gaz spécifié par la transaction d'origine.
0x3b	EXTCODESIZE	1	1	Obtient la taille du code d'un compte. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|\boldsymbol{\sigma}[\boldsymbol{\mu}_s[0] \mod 2^{160}]_c\|$
0x3c	EXTCODECOPY	4	0	Copie le code d'un compte en mémoire. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1]+i] \equiv \begin{cases} \mathbf{c}[\boldsymbol{\mu}_{\mathbf{s}}[2]+i] & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[2]+i < \|\mathbf{c}\| \\ \text{STOP} & \text{sinon} \end{cases}$ où $\mathbf{c} \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_s[0] \mod 2^{160}]_c$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ Les additions en $\boldsymbol{\mu}_{\mathbf{s}}[2]+i$ ne sont pas sujettes au modulo 2^{256}.

40s : Informations de bloc

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Obtient l'empreinte de l'un des 256 derniers blocs complets. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{H_p}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ Où P est l'empreinte d'un bloc d'ordre particulier, jusqu'à un âge maximum. 0 est mis sur la pile si le numéro de bloc demandé est plus grand que le numéro du bloc actuel ou plus éloigné que de 256 blocs du bloc actuel. $P(h,n,a) \equiv \begin{cases} 0 & \text{si} \quad n > H_i \vee a = 256 \vee h = 0 \\ h & \text{si} \quad n = H_i \\ P(H_p, n, a+1) & \text{sinon} \end{cases}$ et on affirme que l'entête H pouvant être déterminé comme son empreinte est l'empreinte parente du bloc qui le suit.
0x41	COINBASE	0	1	Obtient l'adresse bénéficiaire du bloc. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hc}$
0x42	TIMESTAMP	0	1	Obtient le timestamp du bloc. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hs}$
0x43	NUMBER	0	1	Obtient le numéro d'ordre du bloc. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hi}$
0x44	DIFFICULTY	0	1	Obtient la difficulté du bloc. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hd}$
0x45	GASLIMIT	0	1	Obtient la limite de gaz du bloc. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hl}$

				50s : Opérations sur pile, mémoire, stockage et flux
Value	**Mnemonic**	δ	α	**Description**
0x50	POP	1	0	Supprime un élément de la pile.
0x51	MLOAD	1	1	Charge un mot depuis la mémoire. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ L'addition dans le calcul de $\boldsymbol{\mu}'_i$ n'est pas sujette au modulo 2^{256}.
0x52	MSTORE	2	0	Sauve un mot en mémoire. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ L'addition dans le calcul de $\boldsymbol{\mu}'_i$ n'est pas sujette au modulo 2^{256}.
0x53	MSTORE8	2	0	Sauve un octet en mémoire. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32 \rceil)$ L'addition dans le calcul de $\boldsymbol{\mu}'_i$ n'est pas sujette au modulo 2^{256}.
0x54	SLOAD	1	1	Charge un mot depuis le stockage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$
0x55	SSTORE	2	0	Sauve un mot dans le stockage. $\boldsymbol{\sigma}'[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{sset} & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \ \wedge \ \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ G_{sreset} & \text{sinon} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \ \wedge \ \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{sinon} \end{cases}$
0x56	JUMP	1	0	Altération du compteur du programme. $J_{\text{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ Ceci a pour effet l'écriture de la dite value dans $\boldsymbol{\mu}_{pc}$. Voir section 9.
0x57	JUMPI	2	0	Altération conditionnelle du compteur du programme. $J_{\text{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{si} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{pc} + 1 & \text{sinon} \end{cases}$ Ceci a pour effet l'écriture de la dite value dans $\boldsymbol{\mu}_{pc}$. Voir section 9.
0x58	PC	0	1	Obtient la valeur du compteur du programme *avant* l'incrément correspondant à cette instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{pc}$
0x59	MSIZE	0	1	Obtient la taille de la mémoire active en octets. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_i$
0x5a	GAS	0	1	Obtient le montant de gaz disponible, y compris la réduction correspondant au coût de cette instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_g$
0x5b	JUMPDEST	0	0	Marque une destination valide pour les sauts. Cette opération n'a pas d'effet sur l'état de la machine durant l'exécution.

60s & 70s : Opérations d'empilement

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place un élément de 1 octet sur la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{pc}+1)$ où $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{si } x < \|I_{\mathbf{b}}\| \\ 0 & \text{sinon} \end{cases}$ Les octets sont lus séquentiellement depuis le tableau d'octets du code du programme. La fonction c assure que les octets sont mis à zéro si ils dépassent la limite. L'octet est aligné à droite (il prend la place la moins significative en grand-boutien).
0x61	PUSH2	0	1	Place un élément de 2 octets sur la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+2)\big)$ avec $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\|x\|-1}))$ avec c comme défini ci-dessus. Les octets sont alignés à droite (ils prennent la place la moins significative en grand-boutien).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place un élément de 32 octets (un mot entier) sur la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+32)\big)$ où $\boldsymbol{c}$ est défini comme ci-dessus. Les octets sont alignés à droite (ils prennent la place la moins significative en grand-boutien).

80s : Opérations de duplication

Value	Mnemonic	δ	α	Description
0x80	DUP1	1	2	Duplique le 1er élément de la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x81	DUP2	2	3	Duplique le 2ème élément de la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$
⋮	⋮	⋮	⋮	⋮
0x8f	DUP16	16	17	Duplique le 16ème élément de la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[15]$

90s : Opérations d'échange

Value	Mnemonic	δ	α	Description
0x90	SWAP1	2	2	Échange les 1er et 2nd éléments de la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathbf{s}}[1] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x91	SWAP2	3	3	Échange les 1er et 3ème éléments de la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[2]$ $\boldsymbol{\mu}'_{\mathbf{s}}[2] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
⋮	⋮	⋮	⋮	⋮
0x9f	SWAP16	17	17	Échange les 1er et 17ème éléments de la pile. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[16]$ $\boldsymbol{\mu}'_{\mathbf{s}}[16] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$

a0s : Opérations de journalisation

Pour toutes les opérations de journalisation, le changement d'état consiste à ajouter une nouvelle entrée de log à la série de logs du sous-état :
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_a, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
et à mettre à jour le compteur de consommation mémoire :
$\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
La série de sujets de l'entrée, $\mathbf{t}$, diffère en conséquence :

Value	Mnemonic	δ	α	Description
0xa0	LOG0	2	0	Ajoute une entrée de log sans sujet. $\mathbf{t} \equiv ()$
0xa1	LOG1	3	0	Ajoute une entrée de log avec un sujet. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2])$
⋮	⋮	⋮	⋮	⋮
0xa4	LOG4	6	0	Ajoute une entrée de log avec quatre sujets. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4], \boldsymbol{\mu}_{\mathbf{s}}[5])$

f0s : Opérations système

Value	Mnemonic	δ	α	Description
0xf0	CREATE	3	1	Crée un nouveau compte avec le code associé. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[1] \dots (\boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A^+) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_a, I_o, L(\boldsymbol{\mu}_g), I_p, \boldsymbol{\mu}_\mathbf{s}[0], \mathbf{i}, I_e + 1) & \text{si} \quad \boldsymbol{\mu}_\mathbf{s}[0] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_g, \varnothing) & \text{sinon} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}$ excepté $\boldsymbol{\sigma}^*[I_a]_n = \boldsymbol{\sigma}[I_a]_n + 1$ $A' \equiv A \Cup A^+$ qui implique : $A'_\mathbf{s} \equiv A_\mathbf{s} \cup A^+_\mathbf{s} \quad \wedge \quad A'_\mathbf{l} \equiv A_\mathbf{l} \cdot A^+_\mathbf{l} \quad \wedge \quad A'_\mathbf{r} \equiv A_\mathbf{r} + A^+_\mathbf{r}$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ où $x = 0$ si l'exécution du code pour cette opération subit un arrêt exceptionnel $Z(\boldsymbol{\sigma}^*, \boldsymbol{\mu}, I) = \top$ ou $I_e = 1024$ (la profondeur maximale d'appels est atteinte) ou $\boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\sigma}[I_a]_b$ (le solde de l'appelant est trop bas pour assurer le transfert) ; et sinon $x = A(I_a, \boldsymbol{\sigma}[I_a]_n)$, l'adresse du compte nouvellement créé. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[1], \boldsymbol{\mu}_\mathbf{s}[2])$ Ainsi l'ordre des opérandes est : valeur, décalage d'entrée, taille d'entrée.
0xf1	CALL	7	1	Appel de message à l'intérieur d'un compte. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, & \text{si} \quad \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{sinon} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], |\mathbf{o}|\})$ $\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$ $\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ $A' \equiv A \Cup A^+$ $t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}$ où $x = 0$ si l'exécution du code pour cette opération subit un arrêt exceptionnel $Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) = \top$ ou si $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_a]_b$ (pas assez de fonds) ou $I_e = 1024$ (limite de la profondeur de niveaux d'appels atteinte) $x = 1$ sinon. $\boldsymbol{\mu}'_i \equiv M(M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$ Ainsi l'ordre des opérandes est : gaz, à, valeur, décalage d'entrée, taille d'entrée, décalage de sortie, taille de sortie. $C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{si} \quad \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{sinon} \end{cases}$ $C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_g - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{si} \quad \boldsymbol{\mu}_g \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{sinon} \end{cases}$ $C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{si} \quad \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{sinon} \end{cases}$ $C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{si} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{sinon} \end{cases}$
0xf2	CALLCODE	7	1	Appel de message à l'intérieur de ce compte avec le code d'un compte alternatif. Exactement équivalent à CALL excepté : $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_a, I_o, I_a, t, & \text{si} \quad \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{sinon} \end{cases}$ On note le changement dans le 4ème paramètre de l'appel Θ depuis la 2ème valeur de pile $\boldsymbol{\mu}_\mathbf{s}[1]$ (comme dans CALL) à l'adresse actuelle I_a. Cela signifie que le récepteur est bien le même compte que l'actuel, mais que le code est simplement remplacé.
0xf3	RETURN	2	0	Arrête l'exécution et renvoie les données en sortie. $H_{\text{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ L'effet est d'arrêter l'exécution à ce point avec la sortie définie. Voir section 9. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$

0xf4	DELEGATECALL	6	1	Appel de message à l'intérieur de ce compte avec le code d'un autre compte, mais en laissant persistantes les valeurs courantes de *sender* et *value*. Par rapport à CALL, DELEGATECALL prend un argument de moins. L'argument omis est $\boldsymbol{\mu}_\mathbf{s}[2]$. En conséquence, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ et $\boldsymbol{\mu}_\mathbf{s}[6]$ dans la définition de CALL doivent être remplacées respectivement par $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. Par ailleurs exactement équivalent à CALL hormis : $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_s, I_o, I_a, t, \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, 0, I_v, \mathbf{i}, I_e + 1) & \text{si} \quad I_v \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{sinon} \end{cases}$ On note les modifications (en plus de celle du quatrième paramètre) aux second et neuvième paramètres de l'appel Θ. Cela signifie que le destinataire est en fait le même compte qu'à présent, simplement que le code est écrasé *et* que le contexte est presque entièrement identique.
0xfe	INVALID	$\varnothing$	$\varnothing$	Instruction désignée invalide.
0xff	SELFDESTRUCT	1	0	Arrête l'exécution et enregistre le compte pour suppression ultérieure. $A'_\mathbf{s} \equiv A_\mathbf{s} \cup \{I_a\}$ $\boldsymbol{\sigma}'[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b + \boldsymbol{\sigma}[I_a]_b$ $\boldsymbol{\sigma}'[I_a]_b \equiv 0$ $A'_r \equiv A_r + \begin{cases} R_{selfdestruct} & \text{si} \quad I_a \notin A_\mathbf{s} \\ 0 & \text{sinon} \end{cases}$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{selfdestruct} + \begin{cases} G_{newaccount} & \text{si} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{sinon} \end{cases}$

ANNEXE I. BLOC *genesis*

Le bloc *genesis* possède 15 éléments et est spécifié ainsi :

$$\big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, \mathit{stateRoot}, 0, 0, 0_{2048}, 2^{17}, 0, 0, 3141592, \mathit{time}, 0, 0_{256}, \texttt{KEC}\big((42)\big)\big), (), ()\big) \tag{226}$$

où 0_{256} se réfère à l'empreinte parente, une empreinte sur 256 bits à zéro ; 0_{160} se réfère à l'adresse du bénéficiaire, une empreinte sur 160 bits à zéro ; 0_{2048} se réfère au bloom de log, 2048 bits à zéro ; 2^{17} se réfère à la difficulté ; la racine du trie de transaction, la racine du trie des reçus, le gaz utilisé, le numéro de bloc et les extra-données sont à 0, ce qui équivaut au tableau d'octets vide. Les séquences des oncles et des transactions sont vides et représentées par (). $\texttt{KEC}\big((42)\big)$ se réfère à l'empreinte Keccak d'une tableau d'octets de longueur 1 dont le premier et seul octet prend la valeur 42, utilisé pour le nonce. La valeur $\texttt{KEC}\big(\texttt{RLP}\big(()\big)\big)$ se réfère à l'empreinte des listes d'oncles en RLP, qui sont vides.

La série de la preuve de concept comprend un pré-minage de développement, donnant à l'empreinte de la racine d'état une valeur *stateRoot*. *time* prendra également la valeur de l'horodatage initial du bloc de genèse. Il faudra consulter la documentation la plus récente pour connaître ces valeurs.

ANNEXE J. ETHASH

J.1. **Définitions.** Nous employons les définitions suivantes :

Nom	Valeur	Description
$J_{wordbytes}$	4	Octets par mot.
$J_{datasetinit}$	2^{30}	Octets dans l'ensemble de données à la genèse.
$J_{datasetgrowth}$	2^{23}	Croissance de l'ensemble de données par *epoch* (période).
$J_{cacheinit}$	2^{24}	Octets en cache à la genèse.
$J_{cachegrowth}$	2^{17}	Croissance du cache par période.
J_{epoch}	30000	Blocs par période.
$J_{mixbytes}$	128	Longueur du mix en octets.
$J_{hashbytes}$	64	Longueur de l'empreinte en octets.
$J_{parents}$	256	Nombre de parents de chaque élément de l'ensemble de données.
$J_{cacherounds}$	3	Nombre d'itérations en production de cache.
$J_{accesses}$	64	Nombre d'accès dans une boucle de hashimoto.

J.2. **Taille de l'ensemble de données et du cache.** La taille du cache de l'Ethash $\mathbf{c} \in \mathbb{B}$ et celle de l'ensemble de données $\mathbf{d} \in \mathbb{B}$ dépend de la période (*epoch*), qui dépend du numéro de bloc.

$$E_{epoch}(H_i) = \left\lfloor \frac{H_i}{J_{epoch}} \right\rfloor \tag{227}$$

La taille de l'ensemble de données croît de $J_{datasetgrowth}$ octets et la taille du cache de $J_{cachegrowth}$ octets à chaque période. Afin d'éviter une régularité menant à un comportement cyclique, la taille doit être un nombre premier. La taille

est donc réduite d'un multiple de $J_{mixbytes}$, pour l'ensemble de données, et $J_{hashbytes}$ pour le cache. Soit $d_{size} = \|\mathbf{d}\|$ la taille de l'ensemble de données, qui est calculée par

$$d_{size} = E_{prime}(J_{datasetinit} + J_{datasetgrowth} \cdot E_{epoch} - J_{mixbytes}, J_{mixbytes}) \tag{228}$$

La taille du cache, c_{size}, est calculée en utilisant

$$c_{size} = E_{prime}(J_{cacheinit} + J_{cachegrowth} \cdot E_{epoch} - J_{hashbytes}, J_{hashbytes}) \tag{229}$$

$$E_{prime}(x, y) = \begin{cases} x & \text{si} \quad x/y \in \mathbb{P} \\ E_{prime}(x - 1 \cdot y, y) & \text{sinon} \end{cases} \tag{230}$$

J.3. **Génération de l'ensemble de données.** Afin de générer l'ensemble de données, nous avons besoin du cache $\mathbf{c}$, qui est un tableau d'octets. Il dépend de la taille du cache c_{size} et de l'empreinte de la graine $\mathbf{s} \in \mathbb{B}_{32}$.

J.3.1. *Empreinte de la graine.* L'empreinte de la graine cryptographique (*seed*) est différente pour chaque période. Pour la première, c'est l'empreinte Keccak-256 d'une série de 32 octets à zéro. Pour chacune des autres, c'est toujours l'empreinte Keccak-256 de l'empreinte de graine précédente :

$$\mathbf{s} = C_{seedhash}(H_i) \tag{231}$$

$$C_{seedhash}(H_i) = \begin{cases} \texttt{KEC}(\mathbf{0}_{32}) & \text{si} \quad E_{epoch}(H_i) = 0 \\ \texttt{KEC}(C_{seedhash}(H_i - J_{epoch})) & \text{sinon} \end{cases} \tag{232}$$

avec $\mathbf{0}_{32}$ étant 32 octets à zéro.

J.3.2. *Cache.* Le processus de production de cache implique d'utiliser l'empreinte de la graine pour remplir séquentiellement c_{size} octets de mémoire, puis d'effectuer $J_{cacherounds}$ passes de l'algorithme RandMemoHash créé par Lerner [2014]. Le cache initial $\mathbf{c}'$, qui est un tableau d'octets, sera construit comme suit.

Nous définissons le tableau $\mathbf{c}_i$, constitué de 64 octets, comme le $i$ème élément du cache initial :

$$\mathbf{c}_i = \begin{cases} \texttt{KEC512}(\mathbf{s}) & \text{si} \quad i = 0 \\ \texttt{KEC512}(\mathbf{c}_{i-1}) & \text{sinon} \end{cases} \tag{233}$$

Donc $\mathbf{c}'$ peut être défini comme

$$\mathbf{c}'[i] = \mathbf{c}_i \quad \forall \quad i < n \tag{234}$$

$$n = \left\lfloor \frac{c_{size}}{J_{hashbytes}} \right\rfloor \tag{235}$$

Le cache est calculé en effectuant $J_{cacherounds}$ itérations de l'algorithme RandMemoHash au cache initial $\mathbf{c}'$:

$$\mathbf{c} = E_{cacherounds}(\mathbf{c}', J_{cacherounds}) \tag{236}$$

$$E_{cacherounds}(\mathbf{x}, y) = \begin{cases} \mathbf{x} & \text{si} \quad y = 0 \\ E_{\text{RMH}}(\mathbf{x}) & \text{si} \quad y = 1 \\ E_{cacherounds}(E_{\text{RMH}}(\mathbf{x}), y - 1) & \text{sinon} \end{cases} \tag{237}$$

où une seule itération modifie chaque sous-ensemble du cache comme suit :

$$E_{\text{RMH}}(\mathbf{x}) = \big(E_{rmh}(\mathbf{x}, 0), E_{rmh}(\mathbf{x}, 1), ..., E_{rmh}(\mathbf{x}, n - 1)\big) \tag{238}$$

$$E_{rmh}(\mathbf{x}, i) = \texttt{KEC512}(\mathbf{x}'[(i - 1 + n) \mod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \mod n]) \tag{239}$$
$$\text{avec} \quad \mathbf{x}' = \mathbf{x} \quad \text{excepté} \quad \mathbf{x}'[j] = E_{rmh}(\mathbf{x}, j) \quad \forall \quad j < i$$

J.3.3. *Calcul de l'ensemble de données complet.* Il s'agit essentiellement de combiner des données de $J_{parents}$ nœuds de cache sélectionnés de façon pseudo-aléatoire, et hacher cela pour calculer l'ensemble de données. L'ensemble de données complet est ensuite généré par un certain nombre d'éléments, chacun faisant $J_{hashbytes}$ octets :

$$\mathbf{d}[i] = E_{datasetitem}(\mathbf{c}, i) \quad \forall \quad i < \left\lfloor \frac{d_{size}}{J_{hashbytes}} \right\rfloor \tag{240}$$

Afin de calculer chaque élément, nous utilisons un algorithme inspiré de l'empreinte FNV (Glenn Fowler [1991]) dans certains cas, comme un substitut non-associatif de XOR.

$$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (0\text{x}01000193 \oplus \mathbf{y})) \mod 2^{32} \tag{241}$$

Chaque élément de l'ensemble de données peut maintenant être calculé comme :

$$E_{datasetitem}(\mathbf{c}, i) = E_{parents}(\mathbf{c}, i, -1, \varnothing) \tag{242}$$

$$E_{parents}(\mathbf{c}, i, p, \mathbf{m}) = \begin{cases} E_{parents}(\mathbf{c}, i, p + 1, E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1)) & \text{si} \quad p < J_{parents} - 2 \\ E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1) & \text{sinon} \end{cases} \tag{243}$$

$$E_{mix}(\mathbf{m},\mathbf{c},i,p) = \begin{cases} \texttt{KEC512}(\mathbf{c}[i \mod c_{size}] \oplus i) & \text{si} \quad p = 0 \\ E_{\text{FNV}}\big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p, \mathbf{m}[p \mod \lfloor J_{hashbytes}/J_{wordbytes} \rfloor]) \mod c_{size}]\big) & \text{sinon} \end{cases} \tag{244}$$

J.4. **Fonction de preuve de travail.** Il s'agit essentiellement de maintenir un « mix » de $J_{mixbytes}$ octets et de récupérer séquentiellement et de manière répétée $J_{mixbytes}$ octets de l'ensemble de données complet, et d'utiliser la fonction E_{FNV} pour la combiner avec le mix. $J_{mixbytes}$ octets en accès séquentiel sont utilisés pour que chaque itération de l'algorithme récupère toujours une pleine page de la RAM, minimisant ainsi les accès infructueux au *translation lookaside buffer* que les ASIC pourraient théoriquement éviter.

Si la sortie de cet algorithme se trouve en dessous de l'objectif désiré, le nonce est valide. On note que l'application supplémentaire de KEC à la fin assure qu'il existe un nonce intermédiaire qui peut être fourni pour prouver qu'au moins une petite quantité de travail a été effectuée ; cette rapide vérification externe de preuve de travail peut être employée contre les DDOS. Elle sert également à fournir une assurance statistique que le résultat est un nombre sur 256 bits non faussé.

La fonction de preuve de travail renvoie un tableau avec le mix compressé comme premier élément et l'empreinte Keccak-256 de la concaténation du mix compressé avec l'empreinte de la graine comme second élément :

$$\texttt{PoW}(H_{\texttt{H}}, H_n, \mathbf{d}) = \{\mathbf{m}_c(\texttt{KEC}(\texttt{RLP}(L_H(H_{\texttt{H}}))), H_n, \mathbf{d}), \texttt{KEC}(\mathbf{s}_h(\texttt{KEC}(\texttt{RLP}(L_H(H_{\texttt{H}}))), H_n) + \mathbf{m}_c(\texttt{KEC}(\texttt{RLP}(L_H(H_{\texttt{H}}))), H_n, \mathbf{d}))\} \tag{245}$$

avec $H_{\texttt{H}}$ étant l'empreinte de l'en-tête sans le nonce. Le mix $\mathbf{m}_c$ est obtenu comme suit :

$$\mathbf{m}_c(\mathbf{h},\mathbf{n},\mathbf{d}) = E_{compress}(E_{accesses}(\mathbf{d}, \sum_{i=0}^{n_{mix}} \mathbf{s}_h(\mathbf{h},\mathbf{n}), \mathbf{s}_h(\mathbf{h},\mathbf{n}), -1), -4) \tag{246}$$

L'empreinte de la graine étant :

$$\mathbf{s}_h(\mathbf{h},\mathbf{n}) = \texttt{KEC512}(\mathbf{h} + E_{revert}(\mathbf{n})) \tag{247}$$

$E_{revert}(\mathbf{n})$ renvoie la séquence inversée d'octets du nonce $\mathbf{n}$:

$$E_{revert}(\mathbf{n})[i] = \mathbf{n}[\|\mathbf{n}\| - i] \tag{248}$$

Nous notons que l'opérateur « + » entre deux séquences d'octets donne la concaténation des deux séquences.

L'ensemble de données $\mathbf{d}$ est obtenu comme le décrit la section J.3.3.

Le nombre de séquences répliquées dans le mix est :

$$n_{mix} = \left\lfloor \frac{J_{mixbytes}}{J_{hashbytes}} \right\rfloor \tag{249}$$

Afin d'ajouter des nœuds de l'ensemble de données dans le mix, la fonction $E_{accesses}$ est utilisée :

$$E_{accesses}(\mathbf{d},\mathbf{m},\mathbf{s},i) = \begin{cases} E_{mixdataset}(\mathbf{d},\mathbf{m},\mathbf{s},i) & \text{si} \quad i = J_{accesses} - 2 \\ E_{accesses}(E_{mixdataset}(\mathbf{d},\mathbf{m},\mathbf{s},i),\mathbf{s},i+1) & \text{sinon} \end{cases} \tag{250}$$

$$E_{mixdataset}(\mathbf{d},\mathbf{m},\mathbf{s},i) = E_{\text{FNV}}(\mathbf{m}, E_{newdata}(\mathbf{d},\mathbf{m},\mathbf{s},i)) \tag{251}$$

$E_{newdata}$ renvoie un tableau de n_{mix} éléments :

$$E_{newdata}(\mathbf{d},\mathbf{m},\mathbf{s},i)[j] = \mathbf{d}[E_{\text{FNV}}(i \oplus \mathbf{s}[0], \mathbf{m}[i \mod \left\lfloor \frac{J_{mixbytes}}{J_{wordbytes}} \right\rfloor]) \mod \left\lfloor \frac{d_{size}/J_{hashbytes}}{n_{mix}} \right\rfloor \cdot n_{mix} + j] \quad \forall \quad j < n_{mix} \tag{252}$$

Le mix est compressé comme suit :

$$E_{compress}(\mathbf{m},i) = \begin{cases} \mathbf{m} & \text{si} \quad i \geqslant \|\mathbf{m}\| - 8 \\ E_{compress}(E_{\text{FNV}}(E_{\text{FNV}}(E_{\text{FNV}}(\mathbf{m}[i+4], \mathbf{m}[i+5]), \mathbf{m}[i+6]), \mathbf{m}[i+7]), i+8) & \text{sinon} \end{cases} \tag{253}$$