Opérations

2022-03-07

# Opérations

## Présentation

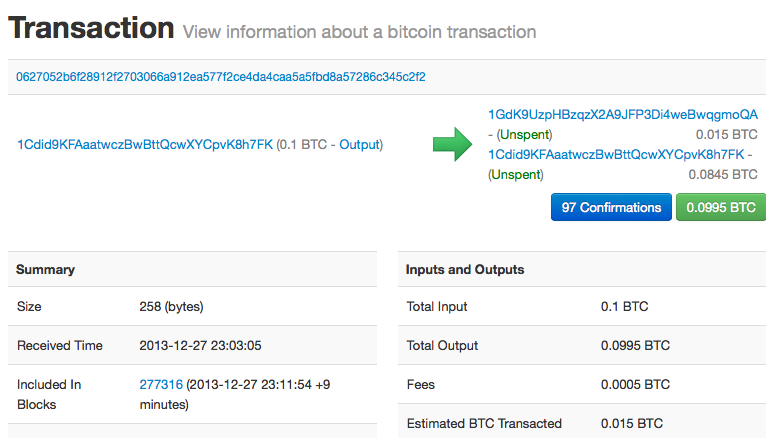
Les transactions sont la partie la plus importante du système Bitcoin. Tout le reste dans le bitcoin est conçu pour garantir que les transactions peuvent être créées, propagées sur le réseau, validées et finalement ajoutées au grand livre mondial des transactions (la chaîne de blocs). Les transactions sont des structures de données qui encodent le transfert de valeur entre les participants au système Bitcoin. Chaque transaction est une entrée publique dans la chaîne de blocs de bitcoin, le grand livre comptable mondial en partie double.

Dans ce chapitre, nous examinerons toutes les différentes formes de transactions, ce qu'elles contiennent, comment les créer, comment elles sont vérifiées et comment elles deviennent partie intégrante de l'enregistrement permanent de toutes les transactions. Lorsque nous utilisons le terme "portefeuille" dans ce chapitre, nous faisons référence au logiciel qui construit les transactions, et pas seulement à la base de données des clés.

## Transactions en détail

Dans [???](#ch02_bitcoin_overview), nous avons examiné la transaction utilisée par Alice pour payer un café au comptoir à café de Bob à l'aide d'un explorateur de blocs ([figure\_title](#alices_transactions_to_bobs_cafe)).

L'application de l'explorateur de blocs montre une transaction de "l'adresse" d'Alice à "l'adresse" de Bob. Il s'agit d'une vue très simplifiée de ce qui est contenu dans une transaction. En fait, comme nous le verrons dans ce chapitre, une grande partie des informations affichées sont construites par l'explorateur de blocs et ne se trouvent pas réellement dans la transaction.



Transaction d'Alice au Bob's Café

### Transactions: Dans les coulisses

Dans les coulisses, une transaction réelle est très différente d'une transaction fournie par un explorateur de blocs typique. En fait, la plupart des constructions de haut niveau que nous voyons dans les différentes interfaces utilisateur des applications bitcoin *n'existent pas réellement* dans le système Bitcoin.

Nous pouvons utiliser l'interface de ligne de commande de Bitcoin Core (getrawtransaction et decoderawtransaction) pour récupérer la transaction "brute" d'Alice, la décoder et voir ce qu'elle contient. Le résultat ressemble à ceci :

**La transaction d’Alice décodée**

{  
 "version": 1,  
 "locktime": 0,  
 "vin": [  
 {  
 "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
 "vout": 0,  
 "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",  
 "sequence": 4294967295  
 }  
 ],  
 "vout": [  
 {  
 "value": 0.01500000,  
 "scriptPubKey": "OP\_DUP OP\_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP\_EQUALVERIFY OP\_CHECKSIG"  
 },  
 {  
 "value": 0.08450000,  
 "scriptPubKey": "OP\_DUP OP\_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP\_EQUALVERIFY OP\_CHECKSIG",  
 }  
 ]  
}

Vous remarquerez peut-être certaines choses à propos de cette transaction, principalement les choses qui manquent ! Où est l'adresse d'Alice ? Où est l'adresse de Bob ? Où est l'entrée 0.1 "envoyée" par Alice ? En bitcoin, il n'y a pas de pièces, pas d'expéditeurs, pas de destinataires, pas de soldes, pas de comptes et pas d'adresses. Toutes ces choses sont construites à un niveau supérieur pour le bénéfice de l'utilisateur, pour rendre les choses plus faciles à comprendre.

Vous pouvez également remarquer de nombreux champs étranges et indéchiffrables et des chaînes hexadécimales. Ne vous inquiétez pas, nous expliquerons en détail chaque champ affiché ici dans ce chapitre.

## Sorties et entrées de transaction

("sorties de transaction non dépensées (UTXO)" )La base fondamentale d'une transaction bitcoin est une *sortie de transaction*. Les sorties de transaction sont des morceaux indivisibles de monnaie bitcoin, enregistrés sur la chaîne de blocs et reconnus comme valides par l'ensemble du réseau. Les nœuds complets Bitcoin suivent toutes les sorties disponibles et utilisables, appelées *sorties de transaction non dépensées*, ou *UTXO*. La collection de tous les UTXO est connue sous le nom d'*ensemble UTXO* et compte actuellement des millions d'UTXO. L'ensemble UTXO augmente à mesure que de nouveaux UTXO sont créés et diminue lorsque UTXO est consommé. Chaque transaction représente un changement (transition d'état) dans l'ensemble UTXO.

Lorsque nous disons que le portefeuille d'un utilisateur a "reçu" du bitcoin, nous voulons dire que le portefeuille a détecté sur la chaîne de blocs un UTXO pouvant être dépensé avec l'une des clés contrôlées par ce portefeuille. Ainsi, le "solde" de bitcoins d'un utilisateur est la somme de tous les UTXO que le portefeuille de l'utilisateur peut dépenser et qui peuvent être dispersés parmi des centaines de transactions et des centaines de blocs. Le concept de solde est créé par l'application de portefeuille. Le portefeuille calcule le solde de l'utilisateur en scannant la chaîne de blocs et en agrégeant la valeur de tout UTXO que le portefeuille peut dépenser avec les clés qu'il contrôle. La plupart des portefeuilles maintiennent une base de données ou utilisent un service de base de données pour stocker un ensemble de référence rapide de tous les UTXO qu'ils peuvent dépenser avec les clés qu'ils contrôlent.

[figure\_title](#utxo-stxo) affiche la chaîne de blocs à trois moments différents, alors que la chaîne de transaction de Joe à Gopesh est en cours de construction. Remarquez comment chaque transaction dépense un UTXO qui a été créé dans une transaction précédente, le transformant en une *sortie de transaction dépensée*, ou *STXO*. Puisque la transaction #1 (de Joe à Alice) dépense un seul UTXO (de Joe) et crée un seul UTXO (à Alice), elle ne modifie pas la taille de l'ensemble UTXO. D'autre part, les transactions #2 et #3 créent toutes deux des sorties de change à l'expéditeur, dépensant un seul UTXO et créant deux UTXO (le paiement et la sortie de change). Par conséquent, chacun d'eux augmente la taille de l'ensemble UTXO de 1.

Chaîne de transaction de Joe à Gopesh en cours de construction sur la chaîne de blocs

Une sortie de transaction peut avoir une valeur arbitraire (entière) libellée comme un multiple de satoshis. Tout comme les dollars peuvent être divisés jusqu'à deux décimales sous forme de cents, le bitcoin peut être divisé jusqu'à huit décimales sous forme de satoshis. Bien qu'une sortie puisse avoir n'importe quelle valeur arbitraire, une fois créée, elle est indivisible. Il s'agit d'une caractéristique importante des sorties qui doit être soulignée : les sorties sont des unités de valeur *discrètes* et *indivisibles*, libellées en satoshis entiers. Une production non dépensée ne peut être consommée dans son intégralité que par une transaction.

Si un UTXO est supérieur à la valeur souhaitée d'une transaction, il doit toujours être consommé dans son intégralité et le changement doit être généré dans la transaction. En d'autres termes, si vous avez un UTXO d'une valeur de 20 bitcoins et que vous souhaitez payer seulement 1 bitcoin, votre transaction doit consommer l'intégralité de l'UTXO de 20 bitcoins et produire deux sorties : une payant 1 bitcoin au destinataire souhaité et une autre payant 19 bitcoin en échange. retour dans votre portefeuille. En raison de la nature indivisible des sorties de transaction, la plupart des transactions bitcoin devront générer du change.

Imaginez une cliente qui achète une boisson à 1,50 $, puise dans son portefeuille et essaie de trouver une combinaison de pièces et de billets de banque pour couvrir le coût de 1,50 $. L'acheteur choisira la monnaie exacte si disponible, par exemple un billet d'un dollar et deux quarts (un quart vaut 0,25 $), ou une combinaison de coupures plus petites (six quarts), ou si nécessaire, une unité plus grande comme un billet de 5 $. Si elle remet trop d'argent, disons 5 $, au propriétaire du magasin, elle s'attendra à 3,50 $ de monnaie, qu'elle remettra dans son portefeuille et qu'elle disposera pour de futures transactions.

De même, une transaction bitcoin doit être créée à partir de l'UTXO d'un utilisateur dans toutes les dénominations disponibles pour cet utilisateur. Les utilisateurs ne peuvent pas couper un UTXO en deux, pas plus qu'ils ne peuvent couper un billet d'un dollar en deux et l'utiliser comme monnaie. L'application de portefeuille de l'utilisateur sélectionne généralement à partir de l'UTXO disponible de l'utilisateur pour composer un montant supérieur ou égal au montant de transaction souhaité.

Comme dans la vraie vie, l'application bitcoin peut utiliser plusieurs stratégies pour satisfaire le montant de l'achat : combiner plusieurs unités plus petites, trouver la monnaie exacte ou utiliser une seule unité supérieure à la valeur de la transaction et rendre la monnaie. Tout cet assemblage complexe d'UTXO dépensables est effectué automatiquement par le portefeuille de l'utilisateur et est invisible pour les utilisateurs. Cela n'est pertinent que si vous construisez par programme des transactions brutes à partir d'UTXO.

Une transaction consomme des sorties de transaction non dépensées précédemment enregistrées et crée de nouvelles sorties de transaction qui peuvent être consommées par une transaction future. De cette façon, des morceaux de valeur bitcoin passent d'un propriétaire à l'autre dans une chaîne de transactions consommant et créant des UTXO.

L'exception à la chaîne de sortie et d'entrée est une type spécial de transaction appelé transaction *coinbase*, qui est la première transaction de chaque bloc. Cette transaction y est placée par le mineur "gagnant" et crée un tout nouveau bitcoin payable à ce mineur en récompense de l'exploitation minière. Cette transaction coinbase spéciale ne consomme pas d'UTXO ; à la place, il a un type spécial d'entrée appelé "coinbase". C'est ainsi que la masse monétaire du bitcoin est créée pendant le processus de minage, comme nous le verrons dans [???](#mining).

Qu'est-ce qui vient en premier ? Entrées ou sorties, la poule ou l'œuf ? Strictement parlant, les sorties viennent en premier parce que les transactions coinbase, qui génèrent de nouveaux bitcoins, n'ont pas d'entrées et créent des sorties à partir de rien.

### Sorties des transactions

Chaque transaction bitcoin crée des sorties, qui sont enregistrées sur le grand livre bitcoin. Presque toutes ces sorties, à une exception près (voir [???](#op_return)) créent des morceaux de bitcoin utilisables appelés UTXO, qui sont ensuite reconnus par l'ensemble du réseau et disponibles pour que le propriétaire les dépense lors d'une future transaction.

Les UTXO sont suivis par chaque client Bitcoin à nœud complet dans l'ensemble UTXO. Les nouvelles transactions consomment (dépensent) une ou plusieurs de ces sorties de l'ensemble UTXO.

Les sorties de transaction se composent de deux parties :

* Un montant de bitcoin, libellé en *satoshis*, la plus petite unité de bitcoin
* Un puzzle cryptographique qui détermine les conditions requises pour passer la sortie

Le puzzle cryptographique est également connu sous le nom de *script de verrouillage*, un *script de témoin* ou un scriptPubKey.

Le langage de script de transaction, utilisé dans le script de verrouillage mentionné précédemment, est décrit en détail dans [Scripts de transaction et langage de script](#tx_script).

Maintenant, regardons la transaction d'Alice (montrée précédemment dans [Transactions: Dans les coulisses](#transactions_behind_the_scenes)) et voir si nous pouvons identifier les sorties. Dans l'encodage JSON, les sorties sont dans un tableau (liste) nommé vout :

"vout": [  
 {  
 "value": 0.01500000,  
 "scriptPubKey": "OP\_DUP OP\_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP\_EQUALVERIFY  
 OP\_CHECKSIG"  
 },  
 {  
 "value": 0.08450000,  
 "scriptPubKey": "OP\_DUP OP\_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP\_EQUALVERIFY OP\_CHECKSIG",  
 }  
]

Comme vous pouvez le voir, la transaction contient deux sorties. Chaque sortie est définie par une valeur et un puzzle cryptographique. Dans l'encodage indiqué par Bitcoin Core, la valeur est indiquée en bitcoin, mais dans la transaction elle-même, elle est enregistrée sous la forme d'un entier libellé en satoshis. La deuxième partie de chaque sortie est le puzzle cryptographique qui fixe les conditions de dépenses. Bitcoin Core le montre comme scriptPubKey et nous montre une représentation lisible par l'homme du script.

Le sujet du verrouillage et du déverrouillage de l'UTXO sera abordé plus tard, dans [Construction du script (Verrouiller + Déverrouiller)](#tx_lock_unlock). Le langage de script utilisé pour le script dans scriptPubKey est décrit dans [Scripts de transaction et langage de script](#tx_script). Mais avant d'approfondir ces sujets, nous devons comprendre la structure globale des entrées et sorties des transactions.

#### Sérialisation des transactions: sorties

Lorsque les transactions sont transmises sur le réseau ou échangées entre applications, elles sont *sérialisées*. La sérialisation est le processus de conversion de la représentation interne d'une structure de données dans un format qui peut être transmis un octet à la fois, également appelé flux d'octets. La sérialisation est le plus souvent utilisée pour coder des structures de données à transmettre sur un réseau ou à stocker dans un fichier. Le format de sérialisation d'une sortie de transaction est affiché dans [table\_title](#tx_out_structure).

Sérialisation des sorties de transaction

| Taille | Champ | Description |
| --- | --- | --- |
| 8 octets (petit-boutiste) | Montant | Valeur du bitcoin en satoshis (10-8 bitcoin) |
| 1–9 octets (VarInt) | Taille du script de verrouillage | Longueur du script de verrouillage en octets, à suivre |
| Variable | Script de verrouillage | Un script définissant les conditions nécessaires pour dépenser la sortie |

La plupart des bibliothèques et frameworks bitcoin ne stockent pas les transactions en interne sous forme de flux d'octets, car cela nécessiterait une analyse complexe chaque fois que vous auriez besoin d'accéder à un seul champ. Pour plus de commodité et de lisibilité, les bibliothèques bitcoin stockent les transactions en interne dans des structures de données (généralement des structures orientées objet).

Le processus de conversion de la représentation de flux d'octets d'une transaction en données de représentation interne d'une bibliothèque est appelée *désérialisation* ou *analyse des transactions*. Le processus de reconversion en un flux d'octets pour la transmission sur le réseau, pour le hachage ou pour le stockage sur disque est appelé *sérialisation*. La plupart des bibliothèques bitcoin ont des fonctions intégrées pour la sérialisation et la désérialisation des transactions.

Voyez si vous pouvez décoder manuellement la transaction d'Alice à partir de la forme hexadécimale sérialisée, en trouvant certains des éléments que nous avons vus précédemment. La section contenant les deux sorties est mise en surbrillance dans [example\_title](#example_6_1) pour vous aider :

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73 4d2804fe65fa35779000000008b483045022100884d142d86652a3f47 ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039 ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813 01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84 16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1 7b4a10fa336a8d752adfffffffff02**60e31600000000001976a914ab6** **8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000** **1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac** 00000000

Voici quelques conseils :

* Il y a deux sorties dans la section en surbrillance, chacune sérialisée comme indiqué dans [table\_title](#tx_out_structure).
* La valeur de 0,015 bitcoin est de 1 500 000 satoshis. C'est 16 e3 60 en hexadécimal.
* Dans la transaction sérialisée, la valeur 16 e3 60 est encodée dans l'ordre d’octets petit-boutiste (octet le moins significatif en premier), elle ressemble donc à 60 e3 16.
* La longueur de scriptPubKey est de 25 octets, soit 19 en hexadécimal.

### Entrées de transactions

Les entrées de transaction identifient (par référence) quel UTXO sera consommé et fournissent une preuve de propriété via un script de déverrouillage.

Pour construire une transaction, un portefeuille sélectionne parmi l'UTXO qu'il contrôle, l'UTXO avec une valeur suffisante pour effectuer le paiement demandé. Parfois un UTXO suffit, d'autres fois plus d'un est nécessaire. Pour chaque UTXO qui sera consommé pour effectuer ce paiement, le portefeuille crée une entrée pointant vers l'UTXO et le déverrouille avec un script de déverrouillage.

Examinons plus en détail les composants d'une entrée. La première partie d'une entrée est un pointeur vers un UTXO par référence au hachage de transaction et à un index de sortie, qui identifie l'UTXO spécifique dans cette transaction. La deuxième partie est un script de déverrouillage, que le portefeuille construit afin de satisfaire aux conditions de dépenses définies dans l'UTXO. Le plus souvent, le script de déverrouillage est une signature numérique et une clé publique prouvant la propriété du bitcoin. Cependant, ce ne sont pas tous les scripts de déverrouillage qui contiennent une signature. La troisième partie est un numéro de séquence, qui sera discuté plus tard.

Considérez notre exemple dans [Transactions: Dans les coulisses](#transactions_behind_the_scenes). Les entrées de transaction sont un tableau (liste) appelé vin :

**Les entrées de transaction dans la transaction d'Alice**

"vin": [  
 {  
 "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
 "vout": 0,  
 "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",  
 "sequence": 4294967295  
 }  
]

Comme vous pouvez le voir, il n'y a qu'une seule entrée dans la liste (car un UTXO contenait une valeur suffisante pour effectuer ce paiement). L'entrée contient quatre éléments :

* Un ID de transaction, faisant référence à la transaction qui contient l'UTXO dépensé
* Un index de sortie (vout), identifiant quel UTXO de cette transaction est référencé (le premier est zéro)
* Un scriptSig, qui satisfait aux conditions imposées à l'UTXO, le déverrouillant pour les dépenses
* Un numéro de séquence (à discuter plus tard)

Dans la transaction d'Alice, l'entrée pointe vers l'ID de transaction :

7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18

et l'indice de sortie 0 (c'est-à-dire le premier UTXO créé par cette transaction). Le script de déverrouillage est construit par le portefeuille d'Alice en récupérant d'abord l'UTXO référencé, en examinant son script de verrouillage, puis en l'utilisant pour créer le script de déverrouillage nécessaire pour le satisfaire.

En regardant simplement l'entrée, vous avez peut-être remarqué que nous ne savons rien de cet UTXO, à part une référence à la transaction parent qui le contient. Nous ne connaissons pas sa valeur (montant en satoshi), et nous ne connaissons pas le script de verrouillage qui fixe les conditions pour le dépenser. Pour retrouver ces informations, il faut récupérer l'UTXO référencé en récupérant la transaction mère qui le contient. Notez que parce que la valeur de l'entrée n'est pas explicitement indiquée, nous devons également utiliser l'UTXO référencé afin de calculer les frais qui seront payés dans cette transaction (voir [Frais de transaction](#tx_fees)).

Ce n'est pas seulement le portefeuille d'Alice qui doit récupérer l'UTXO référencé dans les entrées. Une fois cette transaction diffusée sur le réseau, chaque nœud de validation devra également récupérer l'UTXO référencé dans les entrées de transaction afin de valider la transaction.

Les transactions en elles-mêmes semblent incomplètes car elles manquent de contexte. Elles référencent l’UTXO dans leurs entrées mais sans récupérer cet UTXO on ne peut pas connaître la valeur des entrées ni leurs conditions de verrouillage. Lors de l'écriture d'un logiciel bitcoin, chaque fois que vous décodez une transaction dans le but de la valider ou de compter les frais ou de vérifier le script de déverrouillage, votre code devra d'abord récupérer l'UTXO référencé de la blockchain afin de construire le contexte implicite mais non présent dans les références UTXO des entrées. Par exemple, pour calculer le montant payé en frais, vous devez connaître la somme des valeurs des entrées et des sorties. Mais sans récupérer les UTXO référencés dans les entrées, vous ne connaissez pas leur valeur. Ainsi, une opération apparemment simple comme le comptage des frais dans une seule transaction implique en fait plusieurs étapes et des données provenant de plusieurs transactions.

Nous pouvons utiliser la même séquence de commandes avec Bitcoin Core que celle que nous avons utilisée lors de la récupération de la transaction d'Alice (getrawtransaction et decoderawtransaction). Avec cela, nous pouvons obtenir l'UTXO référencé dans l'entrée de la transaction d'Alice et jeter un œil :

**UTXO de la transaction précédente, référencé dans l'entrée de la transaction d'Alice**

"vout": [  
 {  
 "value": 0.10000000,  
 "scriptPubKey": "OP\_DUP OP\_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP\_EQUALVERIFY OP\_CHECKSIG"  
 }  
 ]

On voit que cet UTXO a une valeur de 0.1 BTC et qu'il a un script de verrouillage (scriptPubKey) qui contient "OP\_DUP OP\_HASH160…​".

Pour bien comprendre la transaction d'Alice, nous avons dû récupérer la transaction précédente référencée en entrée. Une fonction qui récupère les transactions précédentes et les sorties de transaction non dépensées est très courante et existe dans presque toutes les bibliothèques et API Bitcoin.

#### Sérialisation des transactions: entrées

Lorsque les transactions sont sérialisées pour être transmises sur le réseau, leurs entrées sont codées dans un flux d'octets comme indiqué dans [table\_title](#tx_in_structure).

Sérialisation des entrées de transaction

| Taille | Champ | Description |
| --- | --- | --- |
| 32 octets | Hachage de transaction | Pointeur vers la transaction contenant l'UTXO à dépenser |
| 4 octets | Indice de sortie | Le numéro d'index de l'UTXO à dépenser ; le premier est 0 |
| 1–9 octets (VarInt) | Taille du script de déverrouillage | Longueur du script de déverrouillage en octets à suivre |
| variables | Déverrouillage-Script | Un script remplissant les conditions du script de verrouillage UTXO |
| 4 octets | Numéro de séquence | Utilisé pour le temps de verrouillage ou désactivé (0xFFFFFFFF) |

Comme pour les sorties, voyons si nous pouvons trouver les entrées de la transaction d'Alice au format sérialisé. Tout d'abord, les entrées décodées :

"vin": [  
 {  
 "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
 "vout": 0,  
 "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",  
 "sequence": 4294967295  
 }  
],

Voyons maintenant si nous pouvons identifier ces champs dans le codage hexadécimal sérialisé dans [example\_title](#example_6_2) :

0100000001**186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73** **4d2804fe65fa35779000000008b483045022100884d142d86652a3f47** **ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039** **ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813** **01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84** **16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1** **7b4a10fa336a8d752adfffffffff**0260e31600000000001976a914ab6 8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000 1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000

Astuces:

* L'ID de transaction est sérialisé dans l'ordre inverse des octets, il commence donc par (hex) 18 et se termine par 79
* L'index de sortie est un groupe de zéros de 4 octets, facile à identifier
* La longueur du scriptSig est de 139 octets, soit 8b en hexadécimal
* Le numéro de séquence est défini sur FFFFFFFF, encore une fois facile à identifier

ScriptSig est un type spécifique de script de déverrouillage qui, lorsqu'il est sérialisé pour être transmis sur le réseau, les entrées sont codées dans un flux d'octets, comme indiqué dans [table\_title](#scriptsig_in_structure). La sérialisation du champ signature est détaillée dans [Sérialisation des signatures (DER)](#seralization_of_signatures_der). Le champ de signature comprend également un type de hachage de signature (SIGHASH), qui est détaillé dans [Types de hachage de signature (SIGHASH)](#sighash_types).

Sérialisation des entrées ScriptSig

| Taille | Champ | Description |
| --- | --- | --- |
| 1–9 octets (VarInt) | Taille signature | Longueur de la signature en octets à suivre |
| variables | signature | Une signature produite par le portefeuille de l'utilisateur à partir de sa clé privée, qui comprend un SIGHASH |
| 1–9 octets (VarInt) | Taille de la clé publique | Longueur de la clé publique en octets à suivre |
| variables | Clé publique | La clé publique, non hachée |

### Frais de transaction

La plupart des transactions incluent des frais de transaction, qui rémunèrent les mineurs de bitcoins pour la sécurisation du réseau. Les frais servent également de mécanisme de sécurité eux-mêmes, en rendant économiquement impossible pour les attaquants d'inonder le réseau de transactions. L'exploitation minière et les frais et récompenses perçus par les mineurs sont discutés plus en détail dans [???](#mining).

Cette section examine comment les frais de transaction sont inclus dans une transaction typique. La plupart des portefeuilles calculent et incluent automatiquement les frais de transaction. Toutefois, si vous créez des transactions par programmation ou à l'aide d'une interface de ligne de commande, vous devez comptabiliser et inclure manuellement ces frais.

Les frais de transaction servent d'incitation à inclure (exploiter) une transaction dans le bloc suivant et également de dissuasion contre les abus du système en imposant un petit coût sur chaque transaction. Les frais de transaction sont perçus par le mineur qui exploite le bloc qui enregistre la transaction sur la chaîne de blocs.

Les frais de transaction sont calculés en fonction de la taille de la transaction en kilo-octets, et non de la valeur de la transaction en bitcoin. Dans l'ensemble, les frais de transaction sont fixés en fonction des forces du marché au sein du réseau Bitcoin. Les mineurs hiérarchisent les transactions en fonction de nombreux critères différents, y compris les frais, et peuvent même traiter les transactions gratuitement dans certaines circonstances. Les frais de transaction affectent la priorité de traitement, ce qui signifie qu'une transaction avec des frais suffisants est susceptible d'être incluse dans le prochain bloc miné, alors qu'une transaction avec des frais insuffisants ou nuls peut être retardée, traitée au mieux après quelques blocs, ou pas traité du tout. Les frais de transaction ne sont pas obligatoires et les transactions sans frais peuvent éventuellement être traitées ; cependant, l'inclusion des frais de transaction encourage le traitement prioritaire.

Au fil du temps, la façon dont les frais de transaction sont calculés et leur effet sur la hiérarchisation des transactions ont évolué. Au départ, les frais de transaction étaient fixes et constants sur l'ensemble du réseau. Progressivement, la structure tarifaire s'est assouplie et peut être influencée par les forces du marché, en fonction de la capacité du réseau et du volume des transactions. Depuis au moins le début de 2016, les limites de capacité en bitcoin ont créé une concurrence entre les transactions, entraînant des frais plus élevés et faisant des transactions gratuites une chose du passé. Les transactions sans frais ou à très faibles frais sont rarement minées et parfois ne seront même pas propagées sur le réseau.

Dans Bitcoin Core, les politiques de relais des frais sont définies par l'option minrelaytxfee. Le minrelaytxfee par défaut actuel est de 0,00001 bitcoin ou un centième de millibitcoin par kilooctet. Par conséquent, par défaut, les transactions dont les frais sont inférieurs à 0,00001 bitcoin sont traitées comme gratuites et ne sont relayées que s'il y a de la place dans le mempool ; sinon, ils sont supprimés. Les nœuds Bitcoin peuvent remplacer la politique de relais de frais par défaut en ajustant la valeur de minrelaytxfee.

Tout service bitcoin qui crée des transactions, y compris les portefeuilles, les échanges, les applications de vente au détail, etc., *doit* implémenter des frais dynamiques. Les frais dynamiques peuvent être mis en œuvre via un service tiers d'estimation des frais ou avec un algorithme d'estimation des frais intégré. Si vous n'êtes pas sûr, commencez par un service tiers et, à mesure que vous acquérez de l'expérience, concevez et implémentez votre propre algorithme si vous souhaitez supprimer la dépendance tierce.

Les algorithmes d'estimation des frais calculent les frais appropriés, en fonction de la capacité et des frais offerts par les transactions « concurrentes ». Ces algorithmes vont du simple (frais moyens ou médians dans le dernier bloc) au sophistiqué (analyse statistique). Ils estiment les frais nécessaires (en satoshis par octet) qui donneront à une transaction une forte probabilité d'être sélectionnée et incluse dans un certain nombre de blocs. La plupart des services offrent aux utilisateurs la possibilité de choisir des frais de priorité élevés, moyens ou faibles. Une priorité élevée signifie que les utilisateurs paient des frais plus élevés, mais la transaction est susceptible d'être incluse dans le bloc suivant. Une priorité moyenne et faible signifie que les utilisateurs paient des frais de transaction moins élevés, mais que la confirmation des transactions peut prendre beaucoup plus de temps.

De nombreuses applications de portefeuille utilisent des services tiers pour le calcul des frais. Un service populaire est [*https://bitcoinfees.earn.com/*](https://bitcoinfees.earn.com/), qui fournit une API et un graphique visuel montrant les frais en satoshi/octet pour différentes priorités.

Les frais statiques ne sont plus viables sur le réseau Bitcoin. Les portefeuilles qui fixent des frais statiques produiront une mauvaise expérience utilisateur car les transactions seront souvent "bloquées" et resteront non confirmées. Les utilisateurs qui ne comprennent pas les transactions et les frais en bitcoins sont consternés par les transactions "bloquées" car ils pensent avoir perdu leur argent.

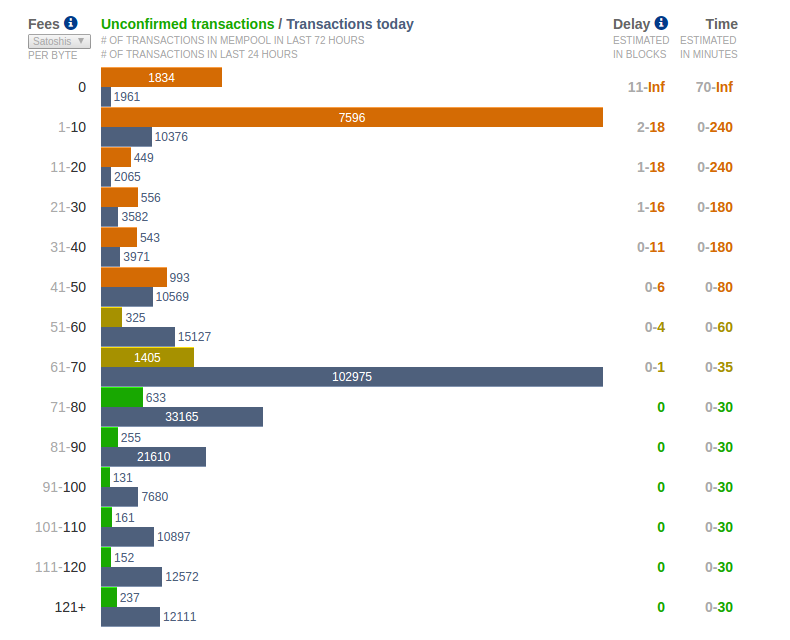
Le graphique en [figure\_title](#bitcoinfeesearncom) affiche l'estimation en temps réel des frais par incréments de 10 satoshi/octet et le temps de confirmation attendu (en minutes et en nombre de blocs) pour les transactions avec des frais dans chaque plage. Pour chaque fourchette de frais (par exemple, 61 à 70 satoshi/octet), deux barres horizontales indiquent le nombre de transactions non confirmées (1 405) et le nombre total de transactions au cours des dernières 24 heures (102 975), avec des frais compris dans cette fourchette. Sur la base du graphique, les frais de haute priorité recommandés à ce moment étaient de 80 satoshi/octet, des frais susceptibles d'entraîner l'inclusion de la transaction dans le bloc suivant (délai de bloc nul). Pour la perspective, la taille médiane des transactions est de 226 octets, donc les frais recommandés pour cette taille de transaction seraient de 18 080 satoshis (0,00018080 BTC).

Les données d'estimation des frais peuvent être récupérées via une simple API HTTP REST, à l'adresse <https://bitcoinfees.earn.com/api/v1/fees/recommended>. Par exemple, sur la ligne de commande en utilisant la commande curl :

**Utilisation de l'API d'estimation des frais**

$ curl https://bitcoinfees.earn.com/api/v1/fees/recommended  
  
{"fastestFee":80,"halfHourFee":80,"hourFee":60}

L'API renvoie un objet JSON avec l'estimation actuelle des frais pour la confirmation la plus rapide (fastestFee), la confirmation dans les trois blocs (halfHourFee) et six blocs (hourFee), en satoshi par octet.



Service d'estimation des frais bitcoinfees.earn.com

### Ajout de frais aux transactions

La structure de données des transactions n'a pas de champ pour les frais. Au lieu de cela, les frais sont implicites comme la différence entre la somme des entrées et la somme des sorties. Tout montant excédentaire qui reste après que toutes les sorties ont été déduites de toutes les entrées est la redevance perçue par les mineurs :

**Les frais de transaction sont implicites, comme l'excédent des intrants moins les extrants :**

Frais = Somme(intrants) - Somme(extrants)

Il s'agit d'un élément quelque peu déroutant des transactions et d'un point important à comprendre, car si vous construisez vos propres transactions, vous devez vous assurer de ne pas inclure par inadvertance des frais très importants en sous-dépensant les intrants. Cela signifie que vous devez comptabiliser toutes les entrées, si nécessaire en créant du change, ou vous finirez par donner un très gros pourboire aux mineurs !

Par exemple, si vous consommez un UTXO de 20 bitcoins pour effectuer un paiement de 1 bitcoin, vous devez inclure une sortie de modification de 19 bitcoins dans votre portefeuille. Sinon, le "reste" de 19 bitcoins sera compté comme des frais de transaction et sera collecté par le mineur qui exploite votre transaction dans un bloc. Bien que vous receviez un traitement prioritaire et rendiez un mineur très heureux, ce n'est probablement pas ce que vous vouliez.

Si vous oubliez d'ajouter une sortie de modification dans une transaction construite manuellement, vous paierez la modification en tant que frais de transaction. Dire "Gardez la monnaie !" au mineur n'est peut-être pas ce que vous vouliez vraiment.

Voyons comment cela fonctionne en pratique, en examinant à nouveau l'achat de café d'Alice. Alice veut dépenser 0,015 bitcoin pour payer son café. Pour s'assurer que cette transaction est traitée rapidement, elle voudra inclure des frais de transaction, disons 0,0005. Cela signifie que le coût total de la transaction sera de 0,0155. Son portefeuille doit donc se procurer un ensemble d'UTXO qui totalise 0,0155 bitcoin ou plus et, si nécessaire, créer de la monnaie. Disons que son portefeuille dispose d'un UTXO de 0,1 bitcoin. Elle devra donc consommer cet UTXO, créer une sortie pour le comptoir à café de Bob pour 0,015, et une seconde sortie avec 0,0845 bitcoin en retour vers son propre portefeuille, laissant 0,0005 bitcoin non alloué, comme frais implicites pour la transaction.

Regardons maintenant un scénario différent. Eugenia, directrice de notre association caritative pour enfants aux Philippines, a organisé une collecte de fonds pour acheter des manuels scolaires pour les enfants. Elle a reçu plusieurs milliers de petits dons de personnes du monde entier, totalisant 50 bitcoins, donc son portefeuille est plein de très petits paiements (UTXO). Maintenant, elle veut acheter des centaines de manuels scolaires auprès d'un éditeur local, en payant en bitcoins.

Alors que l'application de portefeuille d'Eugenia tente de construire une seule transaction de paiement plus importante, elle doit s'approvisionner à partir de l'ensemble UTXO disponible, qui est composé de nombreux montants plus petits. Cela signifie que la transaction résultante proviendra de plus d'une centaine d'UTXO de petite valeur en tant qu'entrées et d'une seule sortie, payant l'éditeur du livre. Une transaction avec autant d'entrées sera supérieure à un kilo-octet, peut-être plusieurs kilo-octets. En conséquence, cela nécessitera des frais beaucoup plus élevés que la transaction de taille médiane.

L'application de portefeuille d'Eugenia calculera les frais appropriés en mesurant la taille de la transaction et en la multipliant par les frais par kilo-octet. De nombreux portefeuilles surpayeront les frais pour les transactions plus importantes afin de s'assurer que la transaction est traitée rapidement. Les frais plus élevés ne sont pas dus au fait qu'Eugenia dépense plus d'argent, mais au fait que sa transaction est plus complexe et plus volumineuse - les frais sont indépendants de la valeur en bitcoins de la transaction.

## Scripts de transaction et langage de script

Le langage de script de transaction bitcoin, appelé *Script* , est un langage d'exécution basé sur une pile de notation polonaise inversée (RPN) de type Forth. Si cela ressemble à du charabia, vous n'avez probablement pas étudié les langages de programmation des années 1960, mais ce n'est pas grave, nous vous expliquerons tout dans ce chapitre. Le script de verrouillage placé sur un UTXO et le script de déverrouillage sont écrits dans ce langage de script. Lorsqu'une transaction est validée, le script de déverrouillage de chaque entrée est exécuté parallèlement au script de verrouillage correspondant pour voir s'il satisfait à la condition de dépense.

Script est un langage très simple qui a été conçu pour être limité dans sa portée et être exécutable sur une gamme de matériel et peut être aussi simple qu'un périphérique intégré. Il nécessite un traitement minimal et ne peut pas faire la plupart des choses fantaisistes que les langages de programmation modernes peuvent faire. Pour son utilisation dans la validation de monnaie programmable, il s'agit d'une fonction de sécurité délibérée.

Aujourd'hui, la plupart des transactions traitées via le réseau Bitcoin ont la forme "Paiement à l'adresse Bitcoin de Bob" et sont basées sur un script appelé Pay-to-Script Public-Key-Hash. Cependant, les transactions bitcoin ne se limitent pas au script "Paiement à l'adresse Bitcoin de Bob". En fait, les scripts de verrouillage peuvent être écrits pour exprimer une grande variété de conditions complexes. Afin de comprendre ces scripts plus complexes, nous devons d'abord comprendre les bases des scripts de transaction et du langage de script.

Dans cette section, nous démontrerons les composants de base du langage de script de transaction bitcoin et montrerons comment il peut être utilisé pour exprimer des conditions simples de dépenses et comment ces conditions peuvent être satisfaites en déverrouillant des scripts.

La validation des transactions Bitcoin n'est pas basée sur un modèle statique, mais est plutôt obtenue grâce à l'exécution d'un langage de script. Ce langage permet d'exprimer une variété presque infinie de conditions. C'est ainsi que le bitcoin obtient le pouvoir de "l'argent programmable".

### Incomplétude de Turing

Le langage de script de transaction bitcoin contient de nombreux opérateurs, mais est délibérément limité d'une manière importante : il n'y a pas de boucles ou de capacités de contrôle de flux complexes autres que le contrôle de flux conditionnel. Cela garantit que le langage n'est pas *Turing complet*, ce qui signifie que les scripts ont une complexité limitée et des temps d'exécution prévisibles. Le script n'est pas un langage universel. Ces limitations garantissent que le langage ne peut pas être utilisé pour créer une boucle infinie ou une autre forme de "bombe logique" qui pourrait être intégrée dans une transaction de manière à provoquer une attaque par déni de service contre le réseau Bitcoin. N'oubliez pas que chaque transaction est validée par chaque nœud complet du réseau Bitcoin. Un langage limité empêche le mécanisme de validation des transactions d'être utilisé comme une vulnérabilité.

### Vérification sans état

Le langage de script de transaction bitcoin est sans état, en ce sens qu'il n'y a pas d'état avant l'exécution du script, ou d'état enregistré après l'exécution du script. Par conséquent, toutes les informations nécessaires à l'exécution d'un script sont contenues dans le script. Un script s'exécutera de manière prévisible de la même manière sur n'importe quel système. Si votre système vérifie un script, vous pouvez être sûr que tous les autres systèmes du réseau Bitcoin vérifieront également le script, ce qui signifie qu'une transaction valide est valide pour tout le monde et tout le monde le sait. Cette prévisibilité des résultats est un avantage essentiel du système Bitcoin.

### Construction du script (Verrouiller + Déverrouiller)

Le moteur de validation des transactions de Bitcoin s'appuie sur deux types de scripts pour valider les transactions : un script de verrouillage et un script de déverrouillage.

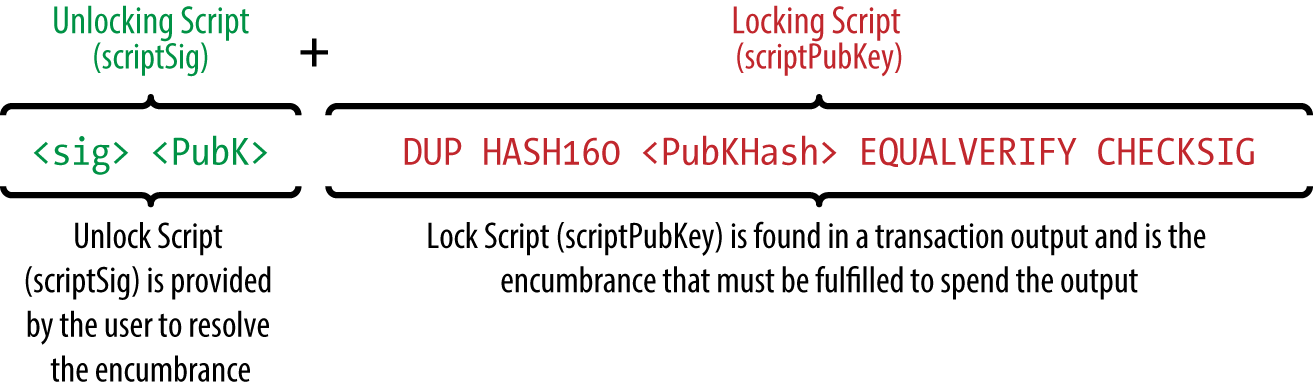
Un script de verrouillage est une condition de dépense placée sur une sortie : il spécifie les conditions qui doivent être remplies pour dépenser la production à l'avenir. Historiquement, le script de verrouillage était appelé *scriptPubKey*, car il contenait généralement une clé publique ou une adresse Bitcoin (hachage de clé publique). Dans ce livre, nous l'appelons un "script de verrouillage" pour reconnaître l'éventail de possibilités beaucoup plus large de cette technologie de script. Dans la plupart des applications bitcoin, ce que nous appelons un script de verrouillage apparaîtra dans le code source sous la forme scriptPubKey. Vous verrez également le script de verrouillage appelé *script de témoin* (voir [???](#segwit)) ou plus généralement comme un *casse-tête cryptographique*. Ces termes signifient tous la même chose, à différents niveaux d'abstraction.

Un script de déverrouillage est un script qui « résout », ou satisfait, les conditions placées sur une sortie par un script de verrouillage et permet à la sortie d'être dépensée. Les scripts de déverrouillage font partie de chaque entrée de transaction. Ils contiennent la plupart du temps une signature numérique produite par le portefeuille de l'utilisateur à partir de sa clé privée. Historiquement, le script de déverrouillage s'appelait *scriptSig*, car il contenait généralement une signature numérique. Dans la plupart des applications bitcoin, le code source fait référence au script de déverrouillage sous la forme scriptSig. Vous verrez également le script de déverrouillage appelé *témoin* (voir [???](#segwit)). Dans ce livre, nous l'appelons un "script de déverrouillage" pour reconnaître la gamme beaucoup plus large d'exigences de script de verrouillage, car ce n’est pas tous les scripts de déverrouillage qui doivent contenir des signatures.

Chaque nœud de validation Bitcoin validera les transactions en exécutant les scripts de verrouillage et de déverrouillage ensemble. Chaque entrée contient un script de déverrouillage et fait référence à un UTXO existant précédemment. Le logiciel de validation copiera le script de déverrouillage, récupérera l'UTXO référencé par l'entrée et copiera le script de verrouillage à partir de cet UTXO. Les scripts de déverrouillage et de verrouillage sont ensuite exécutés en séquence. L'entrée est valide si le script de déverrouillage satisfait les conditions du script de verrouillage (voir [Exécution séparée des scripts de déverrouillage et de verrouillage](#script_exec)). Toutes les entrées sont validées indépendamment, dans le cadre de la validation globale de la transaction.

Notez que l'UTXO est enregistré en permanence dans la blockchain, et est donc invariable et n'est pas affecté par les tentatives infructueuses de le dépenser par référence dans une nouvelle transaction. Seule une transaction valide qui satisfait correctement les conditions de la sortie entraîne que la sortie est considérée comme "dépensée" et retirée de l'ensemble des sorties de transaction non dépensées (ensemble UTXO).

[figure\_title](#scriptSig_and_scriptPubKey) est un exemple des scripts de déverrouillage et de verrouillage pour le type de transaction bitcoin le plus courant (un paiement à un hachage de clé publique), montrant le script combiné résultant de la concaténation des scripts de déverrouillage et de verrouillage avant la validation du script.



Combiner scriptSig et scriptPubKey pour évaluer un script de transaction

#### La pile d'exécution de script

Le langage de script de Bitcoin est appelé un langage basé sur la pile car il utilise une structure de données appelée *pile*. Une pile est une structure de données très simple qui peut être visualisée comme une pile de cartes. Une pile permet deux opérations : push et pop. Push ajoute un élément au sommet de la pile. Pop supprime l'élément du haut de la pile. Les opérations sur une pile ne peuvent agir que sur l'élément le plus haut de la pile. Une structure de données de pile est également appelée file d'attente Last-In-First-Out (dernier entré, premier sorti), ou "LIFO".

Le langage de script exécute le script en traitant chaque élément de gauche à droite. Les nombres (constantes de données) sont poussés sur la pile. Les opérateurs poussent ou sortent un ou plusieurs paramètres de la pile, agissent sur eux et peuvent pousser un résultat sur la pile. Par exemple, OP\_ADD fera sortir deux éléments de la pile, les ajoutera et poussera la somme résultante sur la pile.

Les opérateurs conditionnels évaluent une condition, produisant un résultat booléen VRAI (TRUE) ou FAUX (FALSE). Par exemple, OP\_EQUAL dépile deux éléments de la pile et pousse TRUE (TRUE est représenté par le nombre 1) s'ils sont égaux ou FALSE (représenté par zéro) s'ils ne sont pas égaux. Les scripts de transaction Bitcoin contiennent généralement un opérateur conditionnel, afin qu'ils puissent produire le résultat VRAI qui signifie une transaction valide.

#### Un script simple

Appliquons maintenant ce que nous avons appris sur les scripts et les piles à quelques exemples simples.

Dans [figure\_title](#simplemath_script), le script 2 3 OP\_ADD 5 OP\_EQUAL démontre l'opérateur d'addition arithmétique OP\_ADD, additionnant deux nombres et mettant le résultat sur la pile, suivi de l'opérateur conditionnel OP\_EQUAL, qui vérifie que la somme résultante est égale à 5. Par souci de brièveté, le préfixe OP\_ est omis dans l'exemple étape par étape. Pour plus de détails sur les opérateurs de script et les fonctions disponibles, voir [???](#tx_script_ops).

Bien que la plupart des scripts de verrouillage se réfèrent à un hachage de clé publique (essentiellement, une adresse Bitcoin), nécessitant ainsi une preuve de propriété pour dépenser les fonds, le script n'a pas besoin d'être aussi complexe. Toute combinaison de scripts de verrouillage et de déverrouillage donnant une valeur TRUE est valide. L'arithmétique simple que nous avons utilisée comme exemple de langage de script est également un script de verrouillage valide qui peut être utilisé pour verrouiller une sortie de transaction.

Utilisez une partie de l'exemple de script arithmétique comme script de verrouillage :

3 OP\_ADD 5 OP\_EQUAL

qui peut être satisfaite par une transaction contenant une entrée avec le script de déverrouillage :

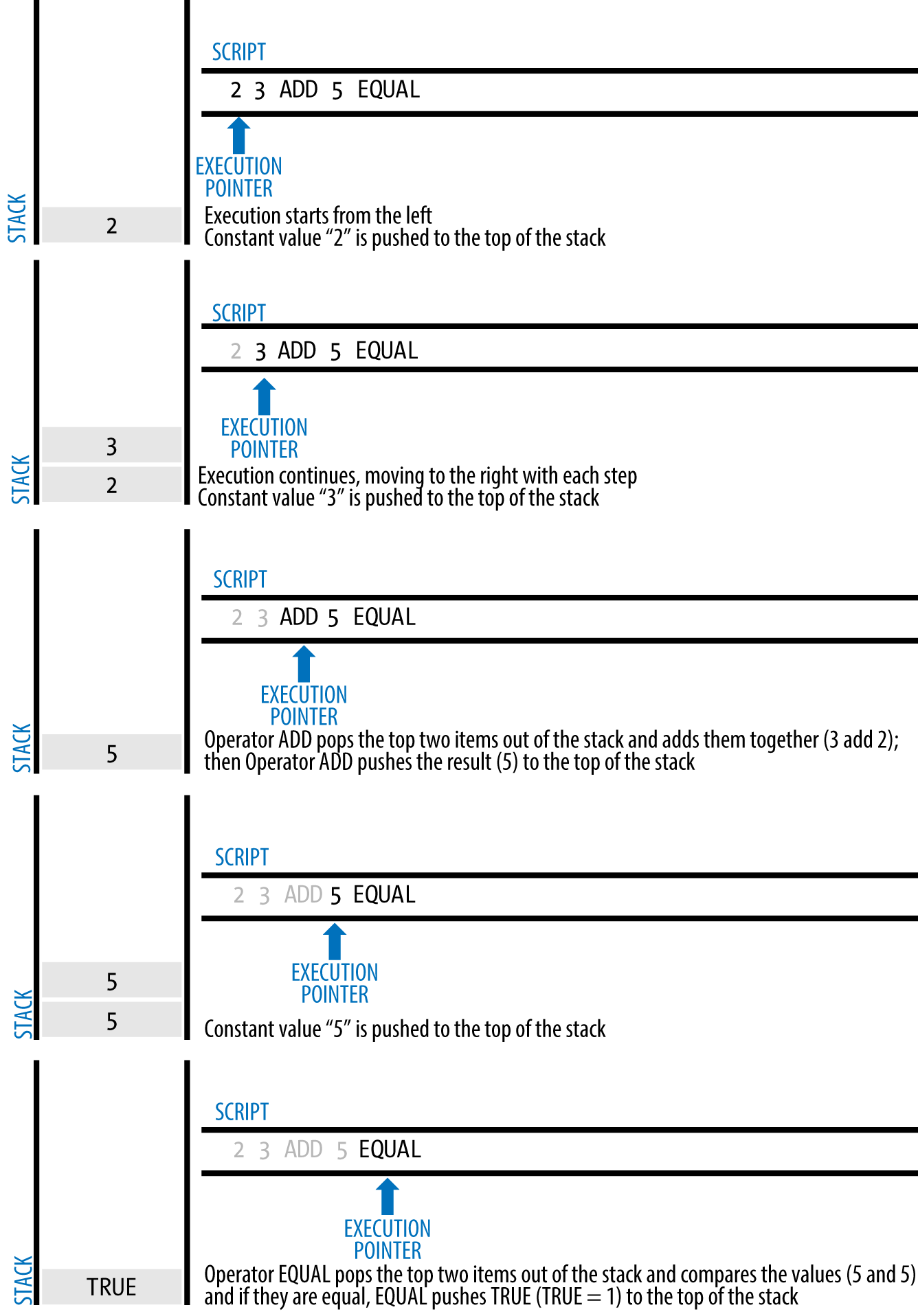
2

Le logiciel de validation combine les scripts de verrouillage et de déverrouillage et le script résultant est :

2 3 OP\_ADD 5 OP\_EQUAL

Comme nous l'avons vu dans l'exemple étape par étape dans [figure\_title](#simplemath_script), lorsque ce script est exécuté, le résultat est OP\_TRUE, rendant la transaction valide. Non seulement il s'agit d'un script de verrouillage de sortie de transaction valide, mais l'UTXO résultant pourrait être dépensé par toute personne ayant les compétences en arithmétique pour savoir que le nombre 2 satisfait le script.

Les transactions sont valides si le premier résultat de la pile est TRUE (noté {0x01}), toute autre valeur différente de zéro, autre que OP\_0, ou si la pile est vide après l'exécution du script. Les transactions ne sont pas valides si la valeur supérieure de la pile est FALSE (une valeur vide de longueur nulle, notée {}) ou si l'exécution du script est arrêtée explicitement par un opérateur, tel que OP\_VERIFY, OP\_RETURN ou un terminateur conditionnel tel que OP\_ENDIF. Voir [???](#tx_script_ops) pour plus de détails.



La validation du script Bitcoin en calculs simples

Ce qui suit est un script légèrement plus complexe, qui calcule 2 + 7 - 3 + 1. Notez que lorsque le script contient plusieurs opérateurs à la suite, la pile permet aux résultats d'un opérateur d'être appliqués par l'opérateur suivant :

2 7 OP\_ADD 3 OP\_SUB 1 OP\_ADD 7 OP\_EQUAL

Essayez de valider vous-même le script précédent à l'aide d'un crayon et de papier. Lorsque l'exécution du script se termine, vous devriez vous retrouver avec la valeur TRUE sur la pile.

#### Exécution séparée des scripts de déverrouillage et de verrouillage

Dans le client Bitcoin d'origine, les scripts de déverrouillage et de verrouillage étaient concaténés et exécutés en séquence. Pour des raisons de sécurité, cela a été modifié en 2010, en raison d'une vulnérabilité qui permettait à un script de déverrouillage malformé de pousser des données sur la pile et de corrompre le script de verrouillage. Dans l'implémentation actuelle, les scripts sont exécutés séparément avec la pile transférée entre les deux exécutions, comme décrit ci-après.

Tout d'abord, le script de déverrouillage est exécuté à l'aide du moteur d'exécution de la pile. Si le script de déverrouillage est exécuté sans erreur (par exemple, s'il ne reste plus de pointeurs " pendants"), la pile principale est copiée et le script de verrouillage est exécuté. Si le résultat de l'exécution du script de verrouillage avec les données de la pile copiées à partir du script de déverrouillage est "TRUE", le script de déverrouillage a réussi à résoudre les conditions imposées par le script de verrouillage et, par conséquent, l'entrée est une autorisation valide de dépenser l'UTXO. S'il reste un résultat autre que "TRUE" après l'exécution du script combiné, l'entrée est invalide car elle n'a pas satisfait aux conditions de dépenses placées sur l'UTXO.

### Pay-to-Public-Key-Hash (P2PKH)

La grande majorité des transactions traitées sur le réseau Bitcoin dépensent des sorties verrouillées avec un script Pay-to-Public-Key-Hash ou "P2PKH". Ces sorties contiennent un script de verrouillage qui verrouille la sortie sur un hachage de clé publique, plus communément appelé adresse Bitcoin. Une sortie verrouillée par un script P2PKH peut être déverrouillée (passée) en présentant une clé publique et une signature numérique créée par la clé privée correspondante (voir [Signatures numériques (ECDSA)](#digital_sigs)).

Par exemple, regardons à nouveau le paiement d'Alice au comptoir à café de Bob. Alice a effectué un paiement de 0,015 bitcoin à l'adresse Bitcoin du comptoir. Cette sortie de transaction aurait un script de verrouillage de la forme :

OP\_DUP OP\_HASH160 <Cafe Public Key Hash> OP\_EQUALVERIFY OP\_CHECKSIG

Le Cafe Public Key Hash équivaut à l'adresse Bitcoin du café, sans l'encodage Base58Check. La plupart des applications afficheraient le *hachage de la clé publique* en codage hexadécimal et non le format familier de l'adresse Bitcoin Base58Check qui commence par un "1".

Le script de verrouillage précédent peut se contenter d'un script de déverrouillage de la forme :

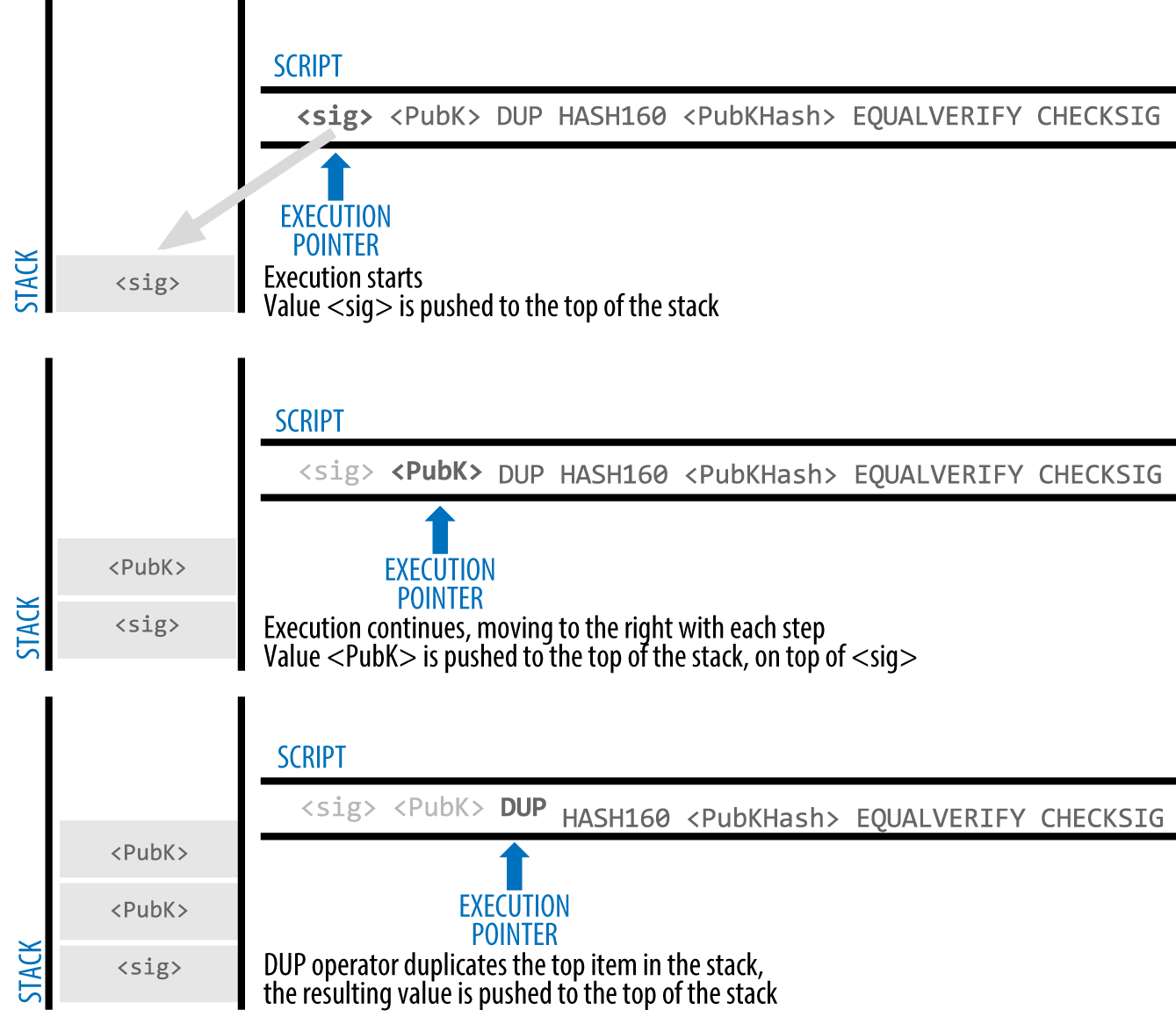
<Cafe Signature> <Cafe Public Key>

Les deux scripts ensemble formeraient le script de validation combiné suivant :

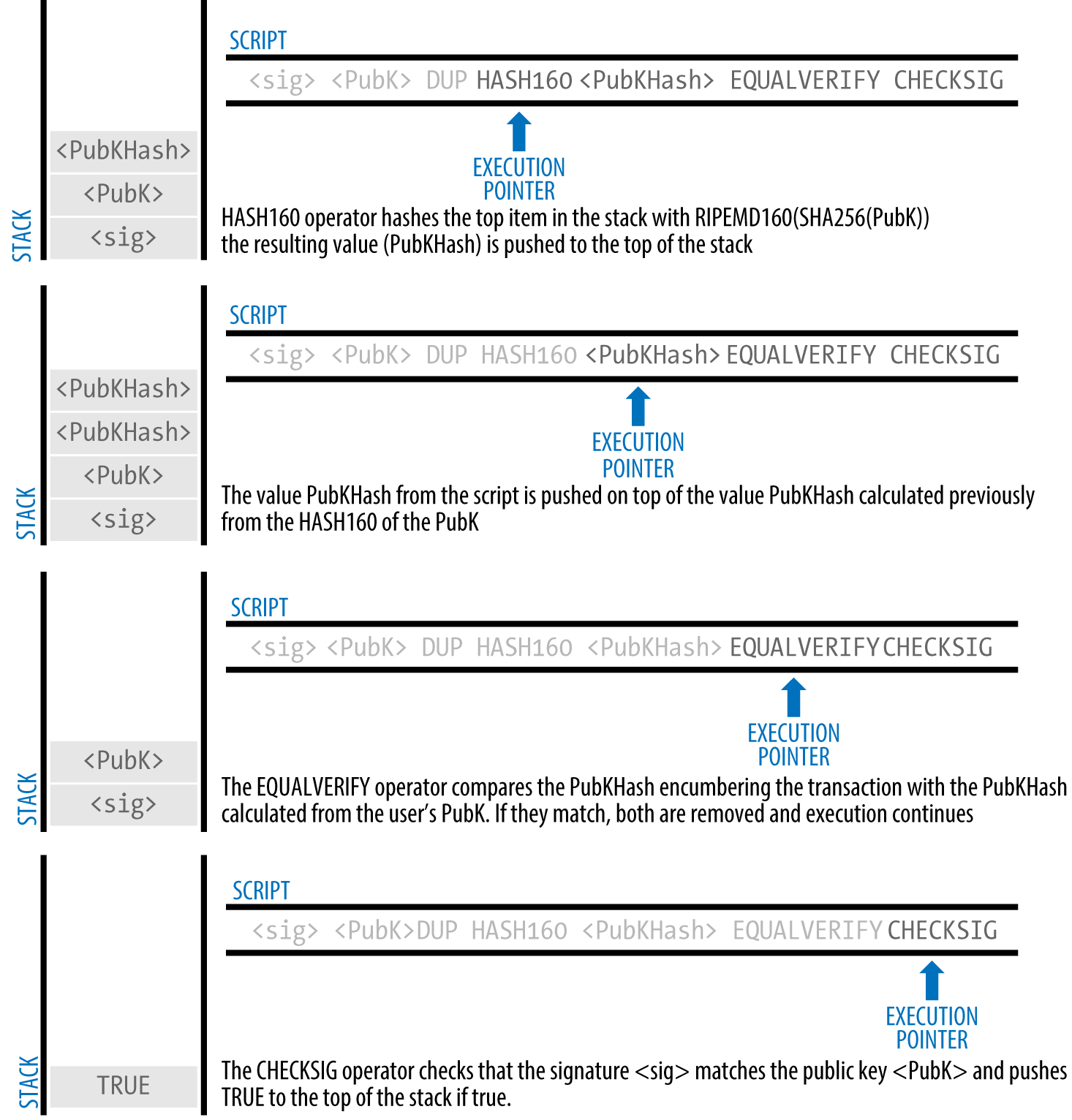
<Cafe Signature> <Cafe Public Key> OP\_DUP OP\_HASH160  
<Cafe Public Key Hash> OP\_EQUALVERIFY OP\_CHECKSIG

Lorsqu'il est exécuté, ce script combiné sera évalué à TRUE si, et seulement si, le script de déverrouillage correspond aux conditions définies par le script de verrouillage. En d'autres termes, le résultat sera VRAI si le script de déverrouillage a une signature valide de la clé privée du comptoir qui correspond au hachage de la clé publique défini comme encombrement.

Les figures #P2PubKHash1 et #P2PubKHash2 montrent (en deux parties) une exécution pas à pas du script combiné, qui prouvera qu'il s'agit d'une transaction valide.



Évaluer un script pour une transaction P2PKH (partie 1 sur 2)



Évaluer un script pour une transaction P2PKH (partie 2 sur 2)

## Signatures numériques (ECDSA)

Jusqu'à présent, nous n'avons approfondi aucun détail sur les "signatures numériques". Dans cette section, nous examinons le fonctionnement des signatures numériques et comment elles peuvent présenter la preuve de la propriété d'une clé privée sans révéler cette clé privée.

L'algorithme de signature numérique utilisé dans le bitcoin est l'\_Elliptic Curve Digital Signature Algorithm\_, ou *ECDSA*. ECDSA est l'algorithme utilisé pour les signatures numériques basées sur des paires de clés privées/publiques à courbe elliptique, comme décrit dans [???](#elliptic_curve). ECDSA est utilisé par les fonctions de script OP\_CHECKSIG, OP\_CHECKSIGVERIFY, OP\_CHECKMULTISIG et OP\_CHECKMULTISIGVERIFY. Chaque fois que vous les voyez dans un script de verrouillage, le script de déverrouillage doit contenir une signature ECDSA.

Une signature numérique a trois objectifs en bitcoin. Premièrement, la signature prouve que le propriétaire de la clé privée, qui est implicitement le propriétaire des fonds, a *autorisé* la dépense de ces fonds. Deuxièmement, la preuve de l'autorisation est *indéniable* (non-répudiable). Troisièmement, la signature prouve que la transaction (ou des parties spécifiques de la transaction) n'a pas et *ne peut pas être modifiée* par qui que ce soit après avoir été signée.

Notez que chaque entrée de transaction est signée indépendamment. Ceci est essentiel, car ni les signatures ni les entrées ne doivent appartenir ou être appliquées par les mêmes "propriétaires". En fait, un schéma de transaction spécifique appelé "CoinJoin" utilise ce fait pour créer des transactions multipartites pour la confidentialité.

Chaque entrée de transaction et toute signature qu'elle peut contenir est *complètement* indépendante de toute autre entrée ou signature. Plusieurs parties peuvent collaborer pour construire des transactions et signer une seule entrée chacune.

Une signature numérique est un schéma mathématique permettant de démontrer l'authenticité d'un message ou de documents numériques. Une signature numérique valide donne au destinataire des raisons de croire que le message a été créé par un expéditeur connu (authentification), que l'expéditeur ne peut pas nier avoir envoyé le message (non-répudiable) et que le message n'a pas été altéré en transit (intégrité).

*Source:* [*https://en.wikipedia.org/wiki/Digital\_signature*](https://en.wikipedia.org/wiki/Digital_signature)

### Comment fonctionnent les signatures numériques

Une signature numérique est un *schéma mathématique* qui se compose de deux parties. La première partie est un algorithme de création d'une signature, à l'aide d'une clé privée (la clé de signature), à partir d'un message (la transaction). La deuxième partie est un algorithme qui permet à quiconque de vérifier la signature, étant donné également le message et une clé publique.

#### Création d'une signature numérique

Dans l'implémentation de l'algorithme ECDSA par bitcoin, le "message" signé est la transaction, ou plus précisément un hachage d'un sous-ensemble spécifique des données de la transaction (voir [Types de hachage de signature (SIGHASH)](#sighash_types)). La clé de signature est la clé privée de l'utilisateur. Le résultat est la signature :

$\(Sig = F\_{sig}(F\_{hash}(m), dA)\)$

où:

* *dA* est la clé privée de signature
* *m* est la transaction (ou une partie de celle-ci)
* *Fhash* est la fonction de hachage
* *Fsig* est l'algorithme de signature
* *Sig* est la signature résultante

Plus de détails sur les mathématiques d'ECDSA peuvent être trouvés dans [Mathématiques ECDSA](#ecdsa_math).

La fonction *Fsig* produit une signature Sig composée de deux valeurs, communément appelées R et S :

Sig = (R, S)

Maintenant que les deux valeurs R et S ont été calculées, elles sont sérialisées en un flux d'octets à l'aide d'un schéma de codage standard international appelé *Distinguished Encoding Rules*, ou *DER* (ou règles d’encodage distingués).

#### Sérialisation des signatures (DER)

Regardons la transaction Alice créée à nouveau. Dans l'entrée de transaction, il y a un script de déverrouillage qui contient la signature encodée DER suivante du portefeuille d'Alice :

3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e38133

Cette signature est un flux d'octets sérialisé des valeurs R et S produites par le portefeuille d'Alice pour prouver qu'elle possède la clé privée autorisée à dépenser cette sortie. Le format de sérialisation se compose de neuf éléments comme suit :

* 0x30—indiquant le début d'une séquence DER
* 0x45—la longueur de la séquence (69 octets)
* 0x02—une valeur entière suit
* 0x21—la longueur de l'entier (33 octets)
* R—00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb
* 0x02—un autre entier suit
* 0x20—la longueur de l'entier (32 octets)
* S—4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
* Un suffixe (0x01) indiquant le type de hachage utilisé (SIGHASH\_ALL)

Voyez si vous pouvez décoder la signature sérialisée (encodée en DER) d'Alice en utilisant cette liste. Les nombres importants sont R et S ; le reste des données fait partie du schéma de codage DER.

### Vérification de la signature

Pour vérifier la signature, il faut avoir la signature (R et S), la transaction sérialisée, et la clé publique (qui correspond à la clé privée utilisée pour créer la signature). Essentiellement, la vérification d'une signature signifie "Seul le propriétaire de la clé privée qui a généré cette clé publique pourrait avoir produit cette signature sur cette transaction."

L'algorithme de vérification de signature prend le message (un hachage de la transaction ou de parties de celui-ci), la clé publique du signataire et la signature (+ valeurs R+ et S), et renvoie TRUE si la signature est valide pour ce message et cette clé publique.

### Types de hachage de signature (SIGHASH)

Les signatures numériques sont appliquées aux messages, qui dans le cas du bitcoin, sont les transactions elles-mêmes. La signature implique un *engagement* du signataire sur des données de transaction spécifiques. Dans sa forme la plus simple, la signature s'applique à l'ensemble de la transaction, engageant ainsi toutes les entrées, sorties et autres champs de transaction. Cependant, une signature ne peut s'engager que sur un sous-ensemble des données d'une transaction, ce qui est utile pour un certain nombre de scénarios, comme nous le verrons dans cette section.

Les signatures Bitcoin ont un moyen d'indiquer quelle partie des données d'une transaction est incluse dans le hachage signé par la clé privée à l'aide d'un drapeau SIGHASH. Le drapeau SIGHASH est un octet unique ajouté à la signature. Chaque signature a un drapeau SIGHASH et le drapeau peut être différent d'une entrée à l'autre. Une transaction avec trois entrées signées peut avoir trois signatures avec des drapeaux SIGHASH différents, chaque signature signant (committant) différentes parties de la transaction.

N'oubliez pas que chaque entrée peut contenir une signature dans son script de déverrouillage. Par conséquent, une transaction qui contient plusieurs entrées peut avoir des signatures avec différents indicateurs SIGHASH qui valident différentes parties de la transaction dans chacune des entrées. Notez également que les transactions bitcoin peuvent contenir des entrées de différents "propriétaires", qui ne peuvent signer qu'une seule entrée dans une transaction partiellement construite (et invalide), collaborant avec d'autres pour rassembler toutes les signatures nécessaires pour effectuer une transaction valide. De nombreux types d'indicateurs SIGHASH n'ont de sens que si vous pensez à plusieurs participants collaborant en dehors du réseau Bitcoin et mettant à jour une transaction partiellement signée.

Il existe trois indicateurs SIGHASH : ALL, NONE et SINGLE, comme indiqué dans [table\_title](#sighash_types_and_their).

Types SIGHASH et leurs significations

| Drapeau SIGHASH | Valeur | Description |
| --- | --- | --- |
| TOUS | 0x01 | La signature s'applique à toutes les entrées et sorties |
| AUCUN | 0x02 | La signature s'applique à toutes les entrées, aucune des sorties |
| SIMPLE | 0x03 | La signature s'applique à toutes les entrées mais uniquement à la sortie avec le même numéro d'index que l'entrée signée |

De plus, il existe un drapeau modificateur SIGHASH\_ANYONECANPAY, qui peut être combiné avec chacun des drapeaux précédents. Lorsque ANYONECANPAY est défini, une seule entrée est signée, laissant le reste (et leurs numéros de séquence) ouverts pour modification. Le ANYONECANPAY a la valeur 0x80 et est appliqué par OR au niveau du bit, ce qui donne les drapeaux combinés comme indiqué dans [table\_title](#sighash_types_with_modifiers).

Types SIGHASH avec modificateurs et leurs significations

| Drapeau SIGHASH | Valeur | Description |
| --- | --- | --- |
| ALL|ANYONECANPAY | 0x81 | La signature s'applique à une entrée et à toutes les sorties |
| NONE|ANYONECANPAY | 0x82 | La signature s'applique à une entrée, aucune des sorties |
| SINGLE|ANYONECANPAY | 0x83 | La signature s'applique à une entrée et à la sortie avec le même numéro d'index |

Ces combinaisons de drapeaux sont résumées dans [figure\_title](#sighash_combinations).

Résumé des différentes combinaisons de sighash

La façon dont les drapeaux SIGHASH sont appliqués lors de la signature et de la vérification est qu'une copie de la transaction est faite et que certains champs à l'intérieur sont tronqués (mis à zéro et vidés). La transaction résultante est sérialisée. Le drapeau SIGHASH est ajouté à la fin de la transaction sérialisée et le résultat est haché. Le hachage lui-même est le "message" qui est signé. Selon l'indicateur SIGHASH utilisé, différentes parties de la transaction sont tronquées. Le hachage résultant dépend de différents sous-ensembles de données dans la transaction. En incluant le SIGHASH comme dernière étape avant le hachage, la signature valide également le type SIGHASH, de sorte qu'il ne peut pas être modifié (par exemple, par un mineur).

Tous les types SIGHASH signent le champ nLocktime de la transaction (voir [???](#transaction_locktime_nlocktime)). De plus, le type SIGHASH lui-même est ajouté à la transaction avant sa signature, de sorte qu'il ne peut pas être modifié une fois signé.

Dans l'exemple de la transaction d'Alice (voir la liste dans [Sérialisation des signatures (DER)](#seralization_of_signatures_der)), nous avons vu que la dernière partie de la signature codée en DER était 01, qui est le drapeau SIGHASH\_ALL. Cela verrouille les données de transaction, de sorte que la signature d'Alice valide l'état de toutes les entrées et sorties. C'est la forme de signature la plus courante.

Examinons quelques-uns des autres types de SIGHASH et comment ils peuvent être utilisés dans la pratique :

ALL|ANYONECANPAY

Cette construction peut être utilisée pour effectuer une transaction de type "financement participatif". les fonds peuvent construire une transaction avec une seule sortie. La sortie unique paie le montant "objectif" à la collecte de fonds. Une telle transaction n'est évidemment pas valide, car elle n'a pas d'entrées. Cependant, d'autres peuvent maintenant la modifier en ajoutant une entrée de leur propre, en tant que don. Ils signent leur propre entrée avec ALL|ANYONECANPAY. À moins que suffisamment d'entrées ne soient rassemblées pour atteindre la valeur de la sortie, la transaction n'est pas valide. Chaque don est un "engagement", qui ne peut pas être collecté par le collecteur de fonds jusqu'à ce que le montant total de l'objectif soit atteint.

NONE

Cette construction peut être utilisée pour créer un "chèque au porteur" ou un "chèque en blanc" d'un montant spécifique. Il s'engage sur l'entrée, mais permet de modifier le script de verrouillage de sortie. N'importe qui peut écrire sa propre adresse Bitcoin dans le script de verrouillage de sortie et racheter la transaction. Cependant, la valeur de sortie elle-même est verrouillée par la signature.

NONE|ANYONECANPAY

Cette construction peut être utilisée pour construire un "collecteur de poussière". Les utilisateurs qui ont de minuscules UTXO dans leur portefeuille ne peuvent pas les dépenser car le coût des frais dépasse la valeur de la poussière. Avec ce type de signature, la poussière UTXO peut être donnée à quiconque pour l'agréger et la dépenser quand il le souhaite.

Il y a quelques propositions pour modifier ou étendre le système SIGHASH. L'une de ces propositions est *Bitmask Sighash Modes* de Glenn Willen de Blockstream, dans le cadre du projet Elements. Cela vise à créer un remplacement flexible pour les types SIGHASH qui permet "des masques de bits arbitraires et réinscriptibles par des mineurs d'entrées et de sorties" qui peuvent exprimer "des schémas de pré-engagement contractuels plus complexes, tels que des offres signées avec changement dans un échange d'actifs distribué".

Vous ne verrez pas les drapeaux SIGHASH présentés comme une option dans l'application de portefeuille d'un utilisateur. À quelques exceptions près, les portefeuilles construisent des scripts P2PKH et signent avec les drapeaux SIGHASH\_ALL. Pour utiliser un indicateur SIGHASH différent, vous devrez écrire un logiciel pour construire et signer des transactions. Plus important encore, les drapeaux SIGHASH peuvent être utilisés par des applications bitcoin à usage spécial qui permettent de nouvelles utilisations.

### Mathématiques ECDSA

Comme mentionné précédemment, les signatures sont créées par une fonction mathématique *Fsig* qui produit une signature composée de deux valeurs *R* et *S*. Dans cette section, nous examinons la fonction *Fsig* plus en détail.

L'algorithme de signature génère d'abord une paire de clés publiques privées *éphémères* (temporaires). Cette paire de clés temporaire est utilisée dans le calcul des valeurs *R* et *S*, après une transformation impliquant la clé privée de signature et le hachage de la transaction.

La paire de clés temporaire est basée sur un nombre aléatoire *k*, qui est utilisé comme clé privée temporaire. À partir de *k*, nous générons la clé publique temporaire correspondante *P* (calculée comme *P = k\*G*, de la même manière que les clés publiques bitcoin sont dérivées ; voir [???](#pubkey)). La valeur *R* de la signature numérique est alors la coordonnée x de la clé publique éphémère *P*.

A partir de là, l'algorithme calcule la valeur *S* de la signature, telle que :

*S* = *k*-1 (*Hash*(*m*) + *dA* \* *R*) *mod n*

où:

* *k* est la clé privée éphémère
* *R* est la coordonnée x de la clé publique éphémère
* *dA* est la clé privée de signature
* *m* est les données de transaction
* *n* est l'ordre premier de la courbe elliptique

La vérification est l'inverse de la fonction de génération de signature, utilisant les valeurs *R*, *S* et la clé publique pour calculer une valeur *P*, qui est un point sur la courbe elliptique (la clé publique éphémère utilisée dans la création de signature) :

*P* = *S*-1 \* *Hash*(*m*) \* *G* + *S*-1 \* *R* \* *Qa*

où:

* *R* et *S* sont les valeurs de signature
* *Qa* est la clé publique d'Alice
* *m* est les données de transaction qui ont été signées
* *G* est le point générateur de la courbe elliptique

Si la coordonnée x du point calculé *P* est égale à *R*, alors le vérificateur peut conclure que la signature est valide.

Notez qu'en vérifiant la signature, la clé privée n'est ni connue ni révélée.

ECDSA est nécessairement un calcul assez compliqué; une explication complète est au-delà de la portée de ce livre. Un certain nombre d'excellents guides en ligne vous guident étape par étape : recherchez "ECDSA expliqué" ou essayez celui-ci : <https://bit.ly/2r0HhGB>.

### L'importance du caractère aléatoire dans les signatures

Comme nous l'avons vu dans [Mathématiques ECDSA](#ecdsa_math), l'algorithme de génération de signature utilise une clé aléatoire *k*, comme base d'un couple clé privée/publique éphémère. La valeur de *k* n'est pas importante, *tant qu'elle est aléatoire*. Si la même valeur *k* est utilisée pour produire deux signatures sur différents messages (transactions), alors la signature *clé privée* peut être calculée par n'importe qui. La réutilisation de la même valeur pour *k* dans un algorithme de signature conduit à l'exposition de la clé privée !

Si la même valeur *k* est utilisée dans l'algorithme de signature sur deux transactions différentes, la clé privée peut être calculée et exposée au monde !

Ce n'est pas seulement une possibilité théorique. Nous avons vu ce problème conduire à l'exposition de clés privées dans quelques implémentations différentes d'algorithmes de signature de transaction en bitcoin. Des personnes se sont fait voler des fonds en raison de la réutilisation par inadvertance d'une valeur *k*. La raison la plus courante de réutilisation d'une valeur *k* est un générateur de nombres aléatoires mal initialisé.

Pour éviter cette vulnérabilité, la meilleure pratique de l'industrie est de ne pas générer *k* avec un générateur de nombres aléatoires semé d'entropie, mais plutôt d'utiliser un processus déterministe-aléatoire semé avec les données de transaction elles-mêmes. Cela garantit que chaque transaction produit un *k* différent. L'algorithme standard de l'industrie pour l'initialisation déterministe de *k* est défini dans [RFC 6979](https://tools.ietf.org/html/rfc6979), publié par l'Internet Engineering Task Force.

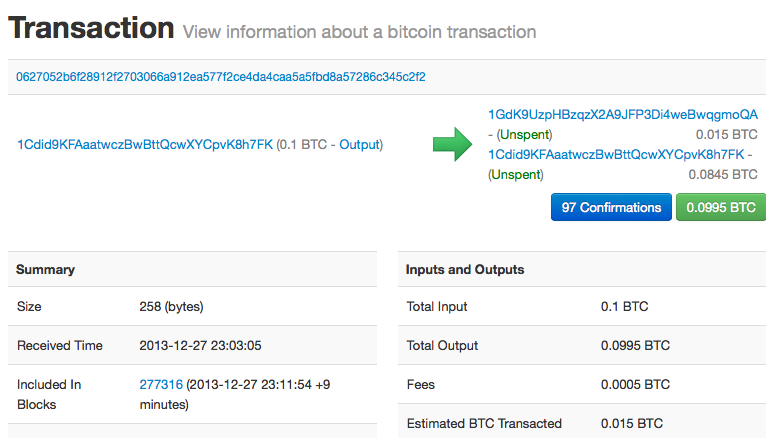
Si vous implémentez un algorithme pour signer des transactions en bitcoin, vous *devez* utiliser la RFC 6979 ou un algorithme déterministe-aléatoire similaire pour vous assurer de générer un *k* différent pour chaque transaction.

## Adresses Bitcoin, soldes et autres abstractions

Nous avons commencé ce chapitre avec la découverte que les transactions sont très différentes "dans les coulisses" de la façon dont elles sont présentées dans les portefeuilles, les explorateurs de chaîne de blocs, et d'autres applications destinées aux utilisateurs. De nombreux concepts simplistes et familiers des chapitres précédents, tels que les adresses et les soldes Bitcoin, semblent être absents de la structure de transaction. Nous avons vu que les transactions ne contiennent pas d'adresses Bitcoin en soi, mais fonctionnent plutôt via des scripts qui verrouillent et déverrouillent des valeurs discrètes de bitcoin. Les soldes ne sont présents nulle part dans ce système et pourtant, chaque application de portefeuille affiche bien en évidence le solde du portefeuille de l'utilisateur.

Maintenant que nous avons exploré ce qui est réellement inclus dans une transaction bitcoin, nous pouvons examiner comment les abstractions de niveau supérieur sont dérivées des composants apparemment primitifs de la transaction.

Regardons à nouveau comment la transaction d'Alice a été présentée sur un explorateur de blocs populaire ([figure\_title](#alice_transaction_to_bobs_cafe)).

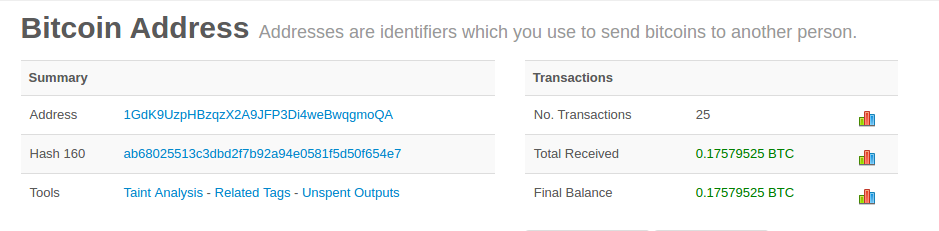


Transaction d'Alice au Bob's Café

Sur le côté gauche de la transaction, l'explorateur de explorateur de chaîne de blocs affiche l'adresse Bitcoin d'Alice comme "expéditeur". En fait, cette information n'est pas dans la transaction elle-même. Lorsque l'explorateur de blockchain fait référence à la transaction, il fait également référence à la transaction précédente associée à l'entrée et extrait la première sortie de cette transaction plus ancienne. Dans cette sortie se trouve un script de verrouillage qui verrouille l'UTXO sur le hachage de la clé publique d'Alice (un script P2PKH). L'explorateur de blockchain a extrait le hachage de la clé publique et l'a encodé à l'aide de l'encodage Base58Check pour produire et afficher l'adresse Bitcoin qui représente cette clé publique.

De même, sur le côté droit, l'explorateur de blockchain affiche les deux sorties ; le premier à l'adresse Bitcoin de Bob et le second à l'adresse Bitcoin d'Alice (comme change). Encore une fois, pour créer ces adresses Bitcoin, l'explorateur de blockchain a extrait le script de verrouillage de chaque sortie, l'a reconnu comme un script P2PKH et a extrait le hachage de clé publique de l'intérieur. Enfin, l'explorateur de blockchain a réencodé chaque hachage de clé publique avec Base58Check pour produire et afficher les adresses Bitcoin.

Si vous deviez cliquer sur l'adresse Bitcoin de Bob, l'explorateur de blockchain vous montrerait la vue dans [figure\_title](#the_balance_of_bobs_bitcoin_address).



Le solde de l'adresse Bitcoin de Bob

L'explorateur de la chaîne de blocs affiche le solde de l'adresse Bitcoin de Bob. Mais nulle part dans le système Bitcoin il n'y a un concept de "solde". Au lieu de cela, les valeurs affichées ici sont construites par l'explorateur de explorateur de chaîne de blocs comme suit.

Pour construire le montant "Total reçu", l'explorateur de explorateur de la chaîne de blocs décodera d'abord l'encodage Base58Check de l'adresse Bitcoin pour récupérer le hachage 160 bits de la clé publique de Bob qui est encodé dans l'adresse. Ensuite, l'explorateur de blockchain recherchera dans la base de données des transactions, à la recherche de sorties avec des scripts de verrouillage P2PKH contenant le hachage de la clé publique de Bob. En additionnant la valeur de toutes les sorties, l'explorateur de la chaîne de blocs peut produire la valeur totale reçue.

Construire le solde actuel (affiché comme "Solde final") nécessite un peu plus de travail. L'explorateur de la chaîne de blocs conserve une base de données distincte des sorties actuellement non dépensées, l'ensemble UTXO. Pour maintenir cette base de données, l'explorateur de la chaîne de blocs doit surveiller le réseau Bitcoin, ajouter les UTXO nouvellement créés et supprimer les UTXO dépensés, en temps réel, tels qu'ils apparaissent dans les transactions non confirmées. Il s'agit d'un processus compliqué qui dépend du suivi des transactions au fur et à mesure qu'elles se propagent, ainsi que du maintien d'un consensus avec le réseau Bitcoin pour s'assurer que la chaîne correcte est suivie. Parfois, l'explorateur de la chaîne de blocs est désynchronisé et sa perspective de l'ensemble UTXO est incomplète ou incorrecte.

À partir de l'ensemble UTXO, l'explorateur de la chaîne de blocs résume la valeur de toutes les sorties non dépensées faisant référence au hachage de la clé publique de Bob et produit le numéro "Solde final" affiché à l'utilisateur.

Afin de produire cette image unique, avec ces deux "soldes", l'explorateur de la chaîne de blocs doit indexer et rechercher parmi des dizaines, des centaines, voire des centaines de milliers de transactions.

En résumé, les informations présentées aux utilisateurs via les applications de portefeuille, les explorateurs de chaînes de blocs et d'autres interfaces utilisateur Bitcoin sont souvent composées d'abstractions de niveau supérieur qui sont dérivées en recherchant de nombreuses transactions différentes, en inspectant leur contenu et en manipulant les données qu'elles contiennent. En présentant cette vision simpliste des transactions bitcoin qui ressemblent à des chèques bancaires d'un expéditeur à un destinataire, ces applications doivent extraire de nombreux détails sous-jacents. Ils se concentrent principalement sur les types de transactions courants : P2PKH avec des signatures SIGHASH\_ALL sur chaque entrée. Ainsi, alors que les applications bitcoin peuvent présenter plus de 80% de toutes les transactions de manière facile à lire, elles sont parfois bloquées par des transactions qui s'écartent de la norme. Les transactions qui contiennent des scripts de verrouillage plus complexes, ou différents indicateurs SIGHASH, ou de nombreuses entrées et sorties, démontrent la simplicité et la faiblesse de ces abstractions.

Chaque jour, des centaines de transactions qui ne contiennent pas de sorties P2PKH sont confirmées sur la chaîne de blocs. Les explorateurs de chaîne de blocs les présentent souvent avec des messages d'avertissement rouges indiquant qu'ils ne peuvent pas décoder une adresse.

Comme nous le verrons au chapitre suivant, il ne s'agit pas nécessairement de transactions étranges. Ce sont des transactions qui contiennent des scripts de verrouillage plus complexes que le P2PKH commun. Nous apprendrons ensuite à décoder et à comprendre des scripts plus complexes et les applications qu'ils prennent en charge.