Transactions et scripts avancés

2022-03-08

# Transactions et scripts avancés

## Présentation

Dans le chapitre précédent, nous avons présenté les éléments de base des transactions bitcoin et examiné le type de script de transaction le plus courant, le script P2PKH. Dans ce chapitre, nous examinerons des scripts plus avancés et comment nous pouvons les utiliser pour créer des transactions avec des conditions complexes.

Tout d'abord, nous examinerons les scripts *multisignature*. Ensuite, nous examinerons le deuxième script de transaction le plus courant, *Pay-to-Script-Hash*, qui ouvre tout un monde de scripts complexes. Ensuite, nous examinerons de nouveaux opérateurs de script qui ajoutent une dimension temporelle au bitcoin, via les *timelocks* (verrouillage horaire). Enfin, nous examinerons les *Segregated Witness* (témoins séparés), une modification architecturale de la structure des transactions.

## Multisignature

Les scripts multisignatures définissent une condition dans laquelle N clés publiques sont enregistrées dans le script et au moins M d'entre elles doivent fournir des signatures pour débloquer les fonds. Ceci est également connu sous le nom de schéma M-sur-N, où N est le nombre total de clés et M est le seuil de signatures requises pour la validation. Par exemple, une multisignature 2 sur 3 est celle où trois clés publiques sont répertoriées comme signataires potentiels et au moins deux d'entre elles doivent être utilisées pour créer des signatures pour une transaction valide afin de dépenser les fonds.

À l'heure actuelle, les scripts multisignatures *standard* sont limités à un maximum de 3 clés publiques répertoriées, ce qui signifie que vous pouvez faire n'importe quoi d'une multisignature 1 sur 1 jusqu’à 3 sur 3 ou toute combinaison dans cette plage. La limitation à 3 clés répertoriées pourrait être levée au moment de la publication de ce livre, alors vérifiez la fonction IsStandard() pour voir ce qui est actuellement accepté par le réseau. Notez que la limite de 3 clés s'applique uniquement aux scripts multisignatures standard (également appelés "nus (bare en anglais)"), et non aux scripts multisignatures enveloppés dans un script Pay-to-Script-Hash (P2SH). Les scripts multisignatures P2SH sont limités à 15 clés, permettant jusqu'à 15 multisignatures sur 15. Cette limitation est également imposée par la fonction IsStandard(). Nous en apprendrons davantage sur P2SH dans [Pay-to-Script-Hash (P2SH)](#p2sh).

La forme générale d'un script de verrouillage définissant une condition multisignature M-de-N est :

M <Public Key 1> <Public Key 2> ... <Public Key N> N CHECKMULTISIG

où N est le nombre total de clés publiques répertoriées et M est le seuil de signatures requises pour dépenser la sortie.

Un script de verrouillage définissant une condition multisignature 2 sur 3 ressemble à ceci :

2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

Le script de verrouillage précédent peut se contenter d'un script de déverrouillage contenant n'importe quelle combinaison de deux signatures issues des clés privées correspondant aux trois clés publiques listées :

<Signature B> <Signature C>

Les deux scripts ensemble formeraient le script de validation combiné :

<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

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. Dans ce cas, la condition est de savoir si le script de déverrouillage possède une signature valide à partir des deux clés privées qui correspondent à deux des trois clés publiques définies comme une charge.

### Un bogue dans l'exécution de CHECKMULTISIG

Il y a un bug dans l'exécution de CHECKMULTISIG qui nécessite une légère solution de contournement. Lorsque CHECKMULTISIG s'exécute, il doit consommer M+N+2 éléments sur la pile en tant que paramètres. Cependant, en raison du bogue, CHECKMULTISIG affichera une valeur supplémentaire ou une valeur de plus que prévu.

Examinons cela plus en détail à l'aide de l'exemple de validation précédent :

<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

Tout d'abord, CHECKMULTISIG affiche l'élément supérieur, qui est N (dans cet exemple "3"). Ensuite, il affiche N éléments, qui sont les clés publiques pouvant signer. Dans cet exemple, les clés publiques A, B et C. Ensuite, un élément apparaît, qui est M, le quorum (combien de signatures sont nécessaires). Ici M = 2. À ce stade, CHECKMULTISIG devrait faire apparaître les éléments finaux M, qui sont les signatures, et voir s'ils sont valides. Cependant, malheureusement, un bogue dans l'implémentation fait que CHECKMULTISIG affiche un élément de plus (M+1 total) qu'il ne le devrait. L'élément supplémentaire n'est pas pris en compte lors de la vérification des signatures, il n'a donc aucun effet direct sur CHECKMULTISIG lui-même. Cependant, une valeur supplémentaire doit être présente car si elle n'est pas présente, lorsque CHECKMULTISIG tente d'apparaître sur une pile vide, cela provoquera une erreur de pile et un échec du script (marquant la transaction comme invalide). Comme l'élément supplémentaire n'est pas pris en compte, il peut s'agir de n'importe quoi, mais habituellement 0 est utilisé.

Parce que ce bogue est devenu une partie des règles de consensus, il doit maintenant être répliqué pour toujours. Par conséquent, la validation correcte du script ressemblerait à ceci :

0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

Ainsi le script de déverrouillage réellement utilisé en multisig n'est pas :

<Signature B> <Signature C>

mais à la place c'est :

0 <Signature B> <Signature C>

À partir de maintenant, si vous voyez un script de déverrouillage multisig, vous devriez vous attendre à voir un 0 supplémentaire au début, dont le seul but est de contourner un bogue qui est accidentellement devenu une règle de consensus.

## Pay-to-Script-Hash (P2SH)

Pay- to-Script-Hash (P2SH) a été introduit en 2012 en tant que nouveau type de transaction puissant qui simplifie grandement l'utilisation de scripts de transaction complexes. Pour expliquer le besoin de P2SH, regardons un exemple pratique.

("Pay-to-Script-Hash (P2SH)", "exemple d'import/export")Dans [???](#ch01_intro_what_is_bitcoin) nous avons présenté Mohammed, un importateur d'électronique basé à Dubaï. La société de Mohammed utilise largement la fonction multisignature de Bitcoin pour ses comptes d'entreprise. Les scripts multisignatures sont l'une des utilisations les plus courantes des capacités de script avancées de Bitcoin et constituent une fonctionnalité très puissante. La société de Mohammed utilise un script multisignature pour tous les paiements des clients, connu en termes comptables sous le nom de "comptes débiteurs ou recevables". Avec le schéma multisignature, tous les paiements effectués par les clients sont verrouillés de telle manière qu'ils nécessitent au moins deux signatures pour être libérés, de Mohammed et de l'un de ses partenaires ou de son avocat qui dispose d'une clé de secours. Un système multisignature comme celui-ci offre des contrôles de gouvernance d'entreprise et protège contre le vol, le détournement de fonds ou la perte.

Le script résultant est assez long et ressemble à ceci :

2 <Clé publique de Mohammed> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 CHECKMULTISIG

Bien que les scripts multisignatures soient une fonctionnalité puissante, ils sont lourds à utiliser. Compte tenu du script précédent, Mohammed devrait communiquer ce script à chaque client avant le paiement. Chaque client devrait utiliser un logiciel spécial de portefeuille bitcoin avec la possibilité de créer des scripts de transaction personnalisés, et chaque client devrait comprendre comment créer une transaction à l'aide de scripts personnalisés. De plus, la transaction résultante serait environ cinq fois plus importante qu'une simple transaction de paiement, car ce script contient des clés publiques très longues. Le fardeau de cette transaction extra-large serait supporté par le client sous la forme de frais. Enfin, un gros script de transaction comme celui-ci serait transporté dans l'UTXO défini dans la RAM de chaque nœud complet, jusqu'à ce qu'il soit dépensé. Tous ces problèmes rendent difficile l'utilisation de scripts de verrouillage complexes dans la pratique.

P2SH a été développé pour résoudre ces difficultés pratiques et rendre l'utilisation de scripts complexes aussi simple qu'un paiement à une adresse Bitcoin. Avec les paiements P2SH, le script de verrouillage complexe est remplacé par son empreinte numérique, un hachage cryptographique. Lorsqu'une transaction tentant de dépenser l'UTXO est présentée ultérieurement, elle doit contenir le script correspondant au hachage, en plus du script de déverrouillage. En termes simples, P2SH signifie "payer à un script correspondant à ce hachage, un script qui sera présenté plus tard lorsque cette sortie sera dépensée".

Dans les transactions P2SH, le script de verrouillage qui est remplacé par un hachage est appelé *redeem script* ou *script de rachat* car il est présenté au système au moment du rachat plutôt que comme un script de verrouillage. [table\_title](#without_p2sh) affiche le script sans P2SH et [table\_title](#with_p2sh) montre le même script encodé avec P2SH.

Script complexe sans P2SH

|  |  |
| --- | --- |
| Script de verrouillage | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG |
| Script de déverrouillage | 0 Sig1 Sig2 |

Script complexe comme P2SH

|  |  |
| --- | --- |
| Script de rachat | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG |
| Script de verrouillage | HASH160 <20-byte hash of redeem script> EQUAL |
| Script de déverrouillage | 0 Sig1 Sig2 <redeem script> |

Comme vous pouvez le voir dans les tableaux, avec P2SH, le script complexe qui détaille les conditions de dépense de la sortie (script de rachat) n'est pas présenté dans le script de verrouillage. Au lieu de cela, seul un hachage de celui-ci se trouve dans le script de verrouillage et le script de rachat lui-même est présenté plus tard, dans le cadre du script de déverrouillage lorsque la sortie est dépensée. Cela déplace le fardeau des frais et de la complexité de l'expéditeur (qui crée la transaction) vers le destinataire (qui déverrouille et dépense la transaction).

Examinons la société de Mohammed, le script multisignature complexe et les scripts P2SH qui en résultent.

Tout d'abord, le script multisignature que la société de Mohammed utilise pour tous les paiements entrants des clients :

2 <Clé publique de Mohammed> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 CHECKMULTISIG

Si les espaces réservés sont remplacés par des clés publiques réelles (affichées ici sous forme de nombres de 520 bits commençant par 04), vous pouvez voir que ce script devient très long :

2  


Ce script entier peut à la place être représenté par un hachage cryptographique de 20 octets, en appliquant d'abord l'algorithme de hachage SHA256, puis en appliquant l'algorithme RIPEMD160 sur le résultat.

Nous utilisons libbitcoin-explorer (bx) sur la ligne de commande pour produire le hachage du script, comme suit :

echo \  
2 \  
[04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] \  
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] \  
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] \  
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] \  
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] \  
5 CHECKMULTISIG \  
| bx script-encode | bx sha256 | bx ripemd160  
54c557e07dde5bb6cb791c7a540e0a4796f5e97e

La série de commandes ci-dessus encode d'abord le script de rachat multisig de Mohammed en tant que script bitcoin sérialisé encodé en hexadécimal. La commande bx suivante calcule le hachage SHA256 de cela. La prochaine commande bx hache à nouveau avec RIPEMD160, produisant le hachage de script final :

Le hachage de 20 octets du script de rachat de Mohammed est :

54c557e07dde5bb6cb791c7a540e0a4796f5e97e

Une transaction P2SH verrouille la sortie sur ce hachage au lieu du script de rachat plus long, en utilisant le script de verrouillage :

HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL

qui, comme vous pouvez le voir, est beaucoup plus courte. Au lieu de "payer à ce script multisignature à 5 clés", la transaction équivalente à P2SH est "payer à un script avec ce hachage". Un client effectuant un paiement à la société de Mohammed n'a qu'à inclure ce script de verrouillage beaucoup plus court dans son paiement. Lorsque Mohammed et ses partenaires veulent dépenser cet UTXO, ils doivent présenter le script de rachat original (celui dont le hachage a verrouillé l'UTXO) et les signatures nécessaires pour le déverrouiller, comme ceci :

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>

Les deux scripts sont combinés en deux étapes. Tout d'abord, le script de rachat est vérifié par rapport au script de verrouillage pour s'assurer que le hachage correspond :

<2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG> HASH160 <redeem scriptHash> EQUAL

Si le hachage du script de rachat correspond, le script de déverrouillage est exécuté de lui-même pour déverrouiller le script de rachat :

<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG

Presque tous les scripts décrits dans ce chapitre ne peuvent être implémentés qu'en tant que scripts P2SH. Par exemple, un script de verrouillage multisignature standard 2 sur 5 ne peut pas être utilisé directement dans le script de verrouillage d'un UTXO, car IsStandard() invaliderait la transaction. Pour se conformer, un script de verrouillage P2SH peut être utilisé à la place, comme vu ci-dessus. Une transaction qui comprend alors un script de déverrouillage P2SH peut être utilisée pour racheter cet UTXO et sera valide tant qu'elle ne contient pas plus de 15 clés publiques.

N'oubliez pas qu'en raison de la politique définie par la fonction IsStandard() au moment de la rédaction de cet article, les scripts multisignatures standard sont limités à 3 clés publiques répertoriées au maximum, tandis que les scripts P2SH sont limités à 15 clés publiques répertoriées au maximum. Les scripts multisignatures standard peuvent invalider les transactions au moyen de leur script de verrouillage *ou* de déverrouillage, tandis que les scripts P2SH peuvent invalider les transactions au moyen de leur script de déverrouillage *uniquement*. En effet, IsStandard() n'a aucun moyen de savoir si un hachage d'un script de rachat dans un script de verrouillage inclut plus de signatures que la limite de taille actuellement imposée, il ne peut donc observer que les scripts de déverrouillage dans les entrées de transaction.

### Adresses P2SH

Une autre partie importante de la fonctionnalité P2SH est la possibilité d'encoder un hachage de script en tant qu'adresse, comme défini dans BIP-13. Les adresses P2SH sont des encodages Base58Check du hachage de 20 octets d'un script, tout comme les adresses Bitcoin sont des encodages Base58Check du hachage de 20 octets d'une clé publique. Les adresses P2SH utilisent le préfixe de version "5", ce qui donne des adresses encodées en Base58Check qui commencent par un "3".

Par exemple, le script complexe de Mohammed, haché et encodé en Base58Check comme une adresse P2SH, devient 39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw. Nous pouvons le confirmer avec la commande bx :

echo \  
'54c557e07dde5bb6cb791c7a540e0a4796f5e97e'\  
 | bx address-encode -v 5  
39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw

Maintenant, Mohammed peut donner cette "adresse" à ses clients et ils peuvent utiliser presque n'importe quel portefeuille bitcoin pour effectuer un paiement simple, comme s'il s'agissait d'une adresse Bitcoin. Le préfixe 3 leur donne un indice qu'il s'agit d'un type d'adresse spécial, celui correspondant à un script au lieu d'une clé publique, mais sinon cela fonctionne exactement de la même manière qu'un paiement à une adresse Bitcoin.

Les adresses P2SH cachent toute la complexité, de sorte que la personne effectuant un paiement ne voit pas le script.

### Avantages du P2SH

La fonction P2SH offre les avantages suivants par rapport à l'utilisation directe de scripts complexes dans le verrouillage des sorties :

* Les scripts complexes sont remplacés par des empreintes digitales plus courtes dans la sortie de la transaction, ce qui réduit la taille de la transaction.
* Les scripts peuvent être codés comme une adresse, de sorte que l'expéditeur et le portefeuille de l'expéditeur n'ont pas besoin d'ingénierie complexe pour implémenter P2SH.
* P2SH transfère le fardeau de la construction du script au destinataire, pas à l'expéditeur.
* P2SH déplace la charge de stockage des données pour le script long de la sortie (qui en plus d'être stockée sur la chaîne de blocs est dans l'ensemble UTXO) vers l'entrée (uniquement stockée sur la chaîne de blocs).
* P2SH déplace la charge de stockage des données pour le script long du moment présent (paiement) à un moment futur (lorsqu'il est dépensé).
* P2SH transfère les frais de transaction plus élevés d'un long script de l'expéditeur au destinataire, qui doit inclure le long script d'échange pour le dépenser.

### Utiliser le script et la validation

("Pay-to-Script-Hash (P2SH)", "scripts de rachat et validation" )Avant la version 0.9.2 du client Bitcoin Core, Pay-to-Script-Hash était limité aux types standard de scripts de transaction bitcoin, par la fonction IsStandard(). Cela signifie que le script d'échange présenté dans la transaction de dépenses ne peut être que l'un des types standard : P2PK, P2PKH ou multisig.

Depuis la version 0.9.2 du client Bitcoin Core, les transactions P2SH peuvent contenir n'importe quel script valide, ce qui rend la norme P2SH beaucoup plus flexible et permet d'expérimenter de nombreux types de transactions nouveaux et complexes.

Vous ne pouvez pas mettre un P2SH dans un script de rachat P2SH, car la spécification P2SH n'est pas récursive. Aussi, bien qu'il soit techniquement possible d'inclure RETURN (voir [Sortie d'enregistrement de données (RETURN)](#op_return)) dans un script de rachat, comme rien dans les règles ne vous empêche de le faire, cela n'a aucune utilité pratique car l'exécution de RETURN lors de la validation entraînera le marquage de la transaction comme invalide.

Notez que, comme le script de rachat n'est pas présenté au réseau tant que vous n'essayez pas de dépenser une sortie P2SH, si vous verrouillez une sortie avec le hachage d'un script de rachat invalide, elle sera traitée malgré tout. L'UTXO sera verrouillé avec succès. Cependant, vous ne pourrez pas le dépenser car la transaction de dépense, qui inclut le script d'échange, ne sera pas acceptée car il s'agit d'un script invalide. Cela crée un risque, car vous pouvez verrouiller des bitcoins dans un P2SH qui ne pourra pas être dépensé plus tard. Le réseau acceptera le script de verrouillage P2SH même s'il correspond à un script de rachat invalide, car le hachage du script ne donne aucune indication sur le script qu'il représente.

Les scripts de verrouillage P2SH contiennent le hachage d'un script de rachat, qui ne donne aucun indice quant au contenu du script de rachat lui-même. La transaction P2SH sera considérée comme valide et acceptée même si le script de rachat est invalide. Vous pourriez accidentellement verrouiller le bitcoin de telle sorte qu'il ne puisse plus être dépensé.

## Sortie d'enregistrement de données (RETURN)

Les bitcoins sont distribués et Le grand livre horodaté, la chaîne de blocs, a des utilisations potentielles bien au-delà des paiements. De nombreux développeurs ont essayé d'utiliser le langage de script de transaction pour tirer parti de la sécurité et de la résilience du système pour des applications telles que les services de notaire numérique, les certificats d'actions et les contrats intelligents. Les premières tentatives d'utilisation du langage de script de bitcoin à ces fins impliquaient la création de sorties de transaction qui enregistraient des données sur la chaîne de blocs; par exemple, pour enregistrer une empreinte digitale d'un fichier de manière à ce que n'importe qui puisse établir la preuve de l'existence de ce fichier à une date précise par référence à cette transaction.

L'utilisation de la chaîne de blocs de Bitcoin pour stocker les données non liées aux paiements en bitcoins sont un sujet controversé. De nombreux développeurs considèrent cette utilisation abusive et veulent la décourager. D'autres y voient une démonstration des puissantes capacités de la technologie chaîne de blocs et souhaitent encourager une telle expérimentation. Ceux qui s'opposent à l'inclusion de données de non-paiement soutiennent que cela provoque un "gonflement de la chaîne de blocs", ce qui impose à ceux qui exécutent des nœuds Bitcoin complets de supporter le coût du stockage sur disque pour les données que la chaîne de blocs n'était pas destinée à transporter. De plus, de telles transactions créent des UTXO qui ne peuvent pas être dépensés, en utilisant l'adresse Bitcoin de destination comme un champ libre de 20 octets. Parce que l'adresse est utilisée pour les données, elle ne correspond pas à une clé privée et l'UTXO résultant ne peut *jamais* être dépensé ; c'est un faux paiement. Ces transactions qui ne peuvent jamais être dépensées ne sont donc jamais supprimées de l'ensemble UTXO et entraînent une augmentation permanente de la taille de la base de données UTXO, ou un "gonflement".

Dans la version 0.9 du client Bitcoin Core, un compromis a été trouvé avec l'introduction de l'opérateur RETURN. RETURN permet aux développeurs d'ajouter 80 octets de données de non-paiement à une sortie de transaction. Cependant, contrairement à l'utilisation de "faux" UTXO, l'opérateur RETURN crée une sortie explicitement et *manifestement indépensable*, qui n'a pas besoin d'être stockée dans l'ensemble UTXO. Les sorties RETURN sont enregistrées sur la chaîne de blocs, elles consomment donc de l'espace disque et contribuent à l'augmentation de la taille de la chaîne de blocs, mais elles ne sont pas stockées dans l'ensemble UTXO et ne gonflent donc pas le bassin de mémoire UTXO et ne surchargent pas les nœuds complets avec plus de dépenses en RAM.

Les scripts RETURN ressemblent à ceci :

RETURN <data>

La portion de données est limitée à 80 octets et représente le plus souvent un hachage, comme la sortie de l'algorithme SHA256 (32 octets). De nombreuses applications mettent un préfixe devant les données pour aider à identifier l'application. Par exemple, le service de notarisation numérique [Preuve d'existence](https://proofofexistence.com) utilise le préfixe de 8 octets DOCPROOF, qui est encodé en ASCII sous la forme 44 4f 43 50 52 4f 4f 46 en hexadécimal.

Gardez à l'esprit qu'il n'y a pas de "script de déverrouillage" correspondant à RETURN qui pourrait éventuellement être utilisé pour "passer" une sortie RETURN. L'intérêt de RETURN est que vous ne pouvez pas dépenser l'argent bloqué dans cette sortie, et donc il n'a pas besoin d'être conservé dans l'ensemble UTXO comme potentiellement dépensable - RETURN est *manifestement indépensable*. RETURN est généralement une sortie avec un montant de zéro bitcoin, car tout bitcoin attribué à une telle sortie est effectivement perdu à jamais. Si un RETURN est référencé comme entrée dans une transaction, le moteur de validation de script arrêtera l'exécution du script de validation et marquera la transaction comme invalide. L'exécution de RETURN provoque essentiellement le script "RETURN" avec un FALSE et s'arrête. Ainsi, si vous référencez accidentellement une sortie RETURN comme entrée dans une transaction, cette transaction n'est pas valide.

Une transaction standard (c'est-à-dire conforme aux vérifications IsStandard()) ne peut avoir qu'une seule sortie RETURN. Cependant, une seule sortie RETURN peut être combinée dans une transaction avec des sorties de tout autre type.

Deux nouvelles options de ligne de commande ont été ajoutées dans Bitcoin Core à partir de la version 0.10. L'option datacarrier contrôle le relais et l'exploration des transactions RETURN, avec la valeur par défaut définie à "1" pour les autoriser. L'option datacarriersize prend un argument numérique spécifiant la taille maximale en octets du script RETURN, 83 octets par défaut, ce qui permet un maximum de 80 octets de données RETURN plus un octet d'opcode RETURN et deux octets de Opcode PUSHDATA.

RETURN a été initialement proposé avec une limite de 80 octets, mais la limite a été réduite à 40 octets lorsque la fonctionnalité a été publiée. En février 2015, dans la version 0.10 de Bitcoin Core, la limite a été relevée à 80 octets. Les nœuds peuvent choisir de ne pas relayer ou exploiter RETURN, ou uniquement de relayer et d'exploiter RETURN contenant moins de 80 octets de données.

## Verrous horaires

Les verrous horaires (ou timelocks) sont des restrictions sur les transactions ou les sorties qui ne permettent de dépenser qu'après un certain temps. Bitcoin a eu une fonction de verrouillage du temps au niveau de la transaction depuis le début. Il est implémenté par le champ nLocktime dans une transaction. Deux nouvelles fonctionnalités de verrou horaire ont été introduites fin 2015 et mi-2016 qui offrent des verrous horaires de niveau UTXO. Ce sont CHECKLOCKTIMEVERIFY et CHECKSEQUENCEVERIFY.

Les verrous horaires sont utiles pour postdater les transactions et verrouiller les fonds à une date future. Plus important encore, les verrous horaires étendent les scripts bitcoin dans la dimension du temps, ouvrant la porte à des contrats intelligents complexes en plusieurs étapes.

### Temps de verrouillage des transactions (nLocktime)

Depuis le début, Bitcoin a eu une fonction de timelock au niveau de la transaction. L'heure de verrouillage de la transaction est un paramètre au niveau de la transaction (un champ dans la structure des données de la transaction) qui définit la première heure à laquelle une transaction est valide et peut être relayée sur le réseau ou ajoutée à la chaîne de blocs. Le verrou horaire est également connu sous le nom de nLocktime à partir du nom de variable utilisé dans la base de code Bitcoin Core. Il est défini sur zéro dans la plupart des transactions pour indiquer une propagation et une exécution immédiates. Si nLocktime est différent de zéro et inférieur à 500 millions, il est interprété comme une hauteur de bloc, ce qui signifie que la transaction n'est pas valide et n'est pas relayée ou incluse dans la blockchain avant la hauteur de bloc spécifiée. S'il est supérieur ou égal à 500 millions, il est interprété comme un horodatage Unix Epoch (secondes depuis le 1er janvier 1970) et la transaction n'est pas valide avant l'heure spécifiée. Les transactions avec nLocktime spécifiant un bloc ou une heure future doivent être conservées par le système d'origine et transmises au réseau Bitcoin uniquement après leur validité. Si une transaction est transmise au réseau avant le nLocktime spécifié, la transaction sera rejetée par le premier nœud comme invalide et ne sera pas relayée vers les autres nœuds. L'utilisation de nLocktime équivaut à postdater un chèque papier.

#### Limitations du temps de verrouillage des transactions

nLocktime a la limitation que s'il permet de dépenser certaines sorties dans le futur, il ne rend pas impossible de les dépenser jusqu'à ce moment-là. Expliquons cela avec l'exemple suivant.

Alice signe une transaction en dépensant l'une de ses sorties à l'adresse de Bob, et fixe la transaction nLocktime à 3 mois dans le futur. Alice envoie cette transaction à Bob pour la conserver. Avec cette transaction, Alice et Bob savent que :

* Bob ne peut pas transmettre la transaction pour racheter les fonds avant que 3 mois ne se soient écoulés.
* Bob peut transmettre la transaction après 3 mois.

Toutefois:

* Alice peut créer une autre transaction, en dépensant deux fois les mêmes entrées sans temps de verrouillage. Ainsi, Alice peut passer le même UTXO avant que les 3 mois ne se soient écoulés.
* Bob n'a aucune garantie qu'Alice ne le fera pas.

Il est important de comprendre les limites de la transaction nLocktime. La seule garantie est que Bob ne pourra pas l'échanger avant que 3 mois ne se soient écoulés. Il n'y a aucune garantie que Bob obtiendra les fonds. Pour obtenir une telle garantie, la restriction de verrouillage temporel doit être placée sur l'UTXO lui-même et faire partie du script de verrouillage, plutôt que sur la transaction. Ceci est réalisé par la forme suivante de timelock, appelée Check Lock Time Verify.

### Vérifier l'heure de verrouillage (CLTV)

En décembre 2015, une nouvelle forme de verrou horaire a été introduite dans le bitcoin en tant qu’embranchement convergent (soft fork) amélioré. Basé sur une spécification dans BIP-65, le nouvel opérateur de script appelé *CHECKLOCKTIMEVERIFY* (*CLTV*) a été ajouté au langage de script. CLTV et est un verrou horaire par sortie, plutôt qu'un verrou horaire par transaction comme c'est le cas avec nLocktime. Cela permet une plus grande flexibilité dans la manière dont les verrous horaires sont appliqués.

En termes simples, en ajoutant l'opcode CLTV dans le script de rachat d'une sortie, cela restreint la sortie, de sorte qu'elle ne peut être dépensée qu'une fois le temps spécifié écoulé.

Alors que nLocktime est un timelock au niveau de la transaction, CLTV est un timelock basé sur la sortie.

CLTV ne remplace pas nLocktime, mais restreint plutôt des UTXO spécifiques de sorte qu'ils ne peuvent être dépensés que dans une future transaction avec nLocktime défini sur une valeur supérieure ou égale.

L'opcode CLTV prend un paramètre en entrée, exprimé sous la forme d'un nombre au même format que nLocktime (soit une hauteur de bloc, soit un temps d'époque Unix). Comme indiqué par le suffixe VERIFY, CLTV est le type d'opcode qui arrête l'exécution du script si le résultat est FALSE. Si le résultat est TRUE, l'exécution continue.

Pour verrouiller une sortie avec CLTV, vous l'insérez dans le script de rachat de la sortie dans la transaction qui crée la sortie. Par exemple, si Alice paie l'adresse de Bob, la sortie contiendra normalement un script P2PKH comme celui-ci :

DUP HASH160 &lt;Hachage clé publique de Bob&gt; EQUALVERIFY CHECKSIG

Pour le verrouiller à un moment, disons dans 3 mois, la transaction serait une transaction P2SH avec un script de rachat comme celui-ci :

<now + 3 months>CHECKLOCKTIMEVERIFY DROP DUP HASH160<Bob's Public Key Hash> EQUALVERIFIER CHECKSIG

où <now {plus} 3 months> + est une hauteur de bloc ou une valeur temporelle estimée à 3 mois à partir du moment où la transaction est extraite : hauteur de bloc actuelle {plus} 12 960 (blocs) ou heure d&#39;époque Unix actuelle {plus} 7 760 000 (secondes). Pour l&#39;instant, ne vous inquiétez pas de l&#39;opcode +DROP qui suit CHECKLOCKTIMEVERIFY ; cela sera expliqué sous peu.

Lorsque Bob essaie de dépenser cet UTXO, il construit une transaction qui fait référence à l'UTXO comme entrée. Il utilise sa signature et sa clé publique dans le script de déverrouillage de cette entrée et définit la transaction nLocktime pour qu'elle soit égale ou supérieure au verrouillage temporel dans l'ensemble CHECKLOCKTIMEVERIFY Alice. Bob diffuse ensuite la transaction sur le réseau Bitcoin.

La transaction de Bob est évaluée comme suit. Si le paramètre CHECKLOCKTIMEVERIFY défini par Alice est inférieur ou égal au nLocktime de la transaction dépensière, l'exécution du script continue (agit comme si un opcode "no operation" ou NOP était exécuté). Sinon, l'exécution du script s'arrête et la transaction est considérée comme invalide.

Plus précisément, CHECKLOCKTIMEVERIFY échoue et arrête l'exécution, marquant la transaction invalide si (source : BIP-65) :

1. la pile est vide ; ou
2. l'élément du haut de la pile est inférieur à 0 ; ou
3. le type de verrou horaire (hauteur versus horodatage) de l'élément de la pile supérieure et le champ nLocktime ne sont pas les mêmes ; ou
4. l'élément supérieur de la pile est supérieur au champ nLocktime de la transaction ; ou
5. le champ nSequence de l'entrée est 0xffffffff.

CLTV et nLocktime utilisent le même format pour décrire les verrous horaires, soit une hauteur de bloc, soit le temps écoulé en secondes depuis l'époque Unix. De manière critique, lorsqu'ils sont utilisés ensemble, le format de nLocktime doit correspondre à celui de CLTV dans les sorties - ils doivent tous deux référencer soit la hauteur de bloc, soit le temps en secondes.

Après l'exécution, si CLTV est satisfait, le paramètre de temps qui l'a précédé reste l'élément supérieur de la pile et peut devoir être supprimé, avec DROP, pour une exécution correcte des opcodes de script suivants. Vous verrez souvent CHECKLOCKTIMEVERIFY suivi de DROP dans les scripts pour cette raison.

En utilisant nLocktime conjointement avec CLTV, le scénario décrit dans [Limitations du temps de verrouillage des transactions](#locktime_limitations) changements. Alice ne peut plus dépenser l'argent (car il est verrouillé avec la clé de Bob) et Bob ne peut pas le dépenser avant l'expiration du temps de verrouillage de 3 mois.

En introduisant la fonctionnalité timelock directement dans le langage de script, CLTV nous permet de développer des scripts complexes très intéressants.

La norme est définie dans [BIP-65 (CHECKLOCKTIMEVERIFY)](https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki).

### Verrous horaires relatifs

nLocktime et CLTV sont les deux *verrous horaires absolus* en ce sens qu'ils spécifient un point absolu dans le temps. Les deux prochaines fonctionnalités de verrou horaire que nous examinerons sont des *verrous horaires relatifs* en ce sens qu'elles spécifient, comme condition de dépense d'une sortie, un temps écoulé depuis la confirmation de la sortie dans la chaîne de blocs.

Les verrous temporels relatifs sont utiles car ils permettent de bloquer une chaîne de deux transactions interdépendantes ou plus, tout en imposant une contrainte de temps sur une transaction qui dépend du temps écoulé depuis la confirmation d'une transaction précédente. En d'autres termes, l'horloge ne commence pas à compter tant que l'UTXO n'est pas enregistré sur la chaîne de blocs. Cette fonctionnalité est particulièrement utile dans les canaux d'état bidirectionnels et les réseaux Lightning, comme nous le verrons dans [???](#state_channels).

Les verrous horaires relatifs, comme les verrous horaires absolus, sont implémentés à la fois avec une fonctionnalité au niveau de la transaction et un opcode au niveau du script. Le verrouillage temporel relatif au niveau de la transaction est implémenté comme une règle de consensus sur la valeur de nSequence, un champ de transaction qui est défini dans chaque entrée de transaction. Les verrous horaires relatifs au niveau du script sont implémentés avec l'opcode CHECKSEQUENCEVERIFY (CSV).

Les verrous horaires relatifs sont implémentés conformément aux spécifications de [BIP-68, Relative lock-time using consensus-enforced sequence numbers](https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki) et https:// github.com/bitcoin/bips/blob/master/bip-0112.mediawiki[BIP-112, CHECKSEQUENCEVERIFY].

BIP-68 et BIP-112 ont été activés en mai 2016 en tant que mise à niveau soft fork des règles de consensus.

### Verrous horaires relatifs avec nSequence

Des verrous horaires relatifs peuvent être définis sur chaque entrée d'une transaction, en définissant le champ nSequence dans chaque contribution.

#### Signification originale de nSequence

Le champ nSequence était à l'origine destiné (mais jamais correctement implémenté) à permettre la modification des transactions dans le mempool. Dans cette utilisation, une transaction contenant des entrées avec une valeur nSequence inférieure à 232 - 1 (0xFFFFFFFF) indiquait une transaction qui n'était pas encore "finalisée". Une telle transaction serait conservée dans le mempool jusqu'à ce qu'elle soit remplacée par une autre transaction dépensant les mêmes entrées avec une valeur nSequence plus élevée. Une fois qu'une transaction a été reçue dont les entrées avaient une valeur nSequence de 0xFFFFFFFF, elle serait considérée comme "finalisée" et minée.

La signification originale de nSequence n'a jamais été correctement implémentée et la valeur de nSequence est habituellement définie sur 0xFFFFFFFF dans les transactions qui n'utilisent pas de verrous horaires. Pour les transactions avec nLocktime ou CHECKLOCKTIMEVERIFY, la valeur nSequence doit être inférieure à 231 pour que les gardes de verrou horaire aient un effet, comme expliqué ci-dessous.

#### nSequence comme un verrou horaire relatif imposé par consensus

Depuis l'activation de BIP-68, de nouvelles règles de consensus s'appliquent pour toute transaction contenant une entrée dont la valeur nSequence est inférieure à 231 (le bit 1<<31 n'est pas défini). Par programmation, cela signifie que si le bit le plus significatif (bit 1<<31) n'est pas défini, c'est un indicateur qui signifie "temps de verrouillage relatif". Sinon (bit 1<<31 défini), la valeur nSequence est réservée à d'autres utilisations telles que l'activation de CHECKLOCKTIMEVERIFY, nLocktime, Opt-In-Replace-By-Fee et d'autres développements futurs.

Les entrées de transaction avec des valeurs nSequence inférieures à 231 sont interprétées comme ayant un verrou horaire relatif. Une telle transaction n'est valide qu'une fois que l'entrée a vieilli du montant relatif du verrou horaire. Par exemple, une transaction avec une entrée et un verrou horaire relatif nSequence de 30 blocs n'est valide que lorsqu'au moins 30 blocs se sont écoulés depuis le moment où l'UTXO référencé dans l'entrée a été extrait. Étant donné que nSequence est un champ par entrée, une transaction peut contenir n'importe quel nombre d'entrées verrouillées dans le temps, qui doivent toutes avoir suffisamment vieilli pour que la transaction soit valide. Une transaction peut inclure à la fois des entrées à verrouillage temporel (nSequence < 231) et des entrées sans verrouillage temporel relatif (nSequence >= 231).

La valeur nSequence est spécifiée en blocs ou en secondes, mais dans un format légèrement différent de celui que nous avons vu utilisé dans nLocktime. Un indicateur de type est utilisé pour différencier les valeurs comptant les blocs et les valeurs comptant le temps en secondes. Le drapeau de type est défini dans le 23e bit le moins significatif (c'est-à-dire la valeur 1<<22). Si le type-flag est défini, alors la valeur nSequence est interprétée comme un multiple de 512 secondes. Si le drapeau de type n'est pas défini, la valeur nSequence est interprétée comme un nombre de blocs.

Lors de l'interprétation de nSequence comme un timelock relatif, seuls les 16 bits les moins significatifs sont pris en compte. Une fois que les drapeaux (bits 32 et 23) sont évalués, la valeur nSequence est généralement "masquée" avec un masque de 16 bits (par exemple, nSequence & 0x0000FFFF).

[simpara\_title](#bip_68_def_of_nseq) montre la disposition binaire de la valeur nSequence, telle que définie par BIP-68.

Définition .BIP-68 du codage nSequence (Source : BIP-68) image::images/mbc2\_0701.png["Définition BIP-68 du codage nSequence"]

Les verrous temporels relatifs basés sur l'application consensuelle de la valeur nSequence sont définis dans BIP-68.

La norme est définie dans [BIP-68, Relative lock-time using consensus-enforced sequence numbers](https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki).

### Verrous horaires relatifs avec CSV

Tout comme CLTV et nLocktime, il existe un opcode de script pour les verrous temporels relatifs qui exploite le Valeur nSequence dans les scripts. Cet opcode est CHECKSEQUENCEVERIFY, communément appelé CSV en abrégé.

L'opcode CSV, lorsqu'il est évalué dans le script de remboursement d'un UTXO, permet de dépenser uniquement dans une transaction dont la valeur d'entrée nSequence est supérieure ou égale au paramètre CSV. Essentiellement, cela limite la dépense de l'UTXO jusqu'à ce qu'un certain nombre de blocs ou de secondes se soient écoulés par rapport au moment où l'UTXO a été miné.

Comme pour CLTV, la valeur dans CSV doit correspondre au format de la valeur nSequence correspondante. Si CSV est spécifié en termes de blocs, il doit en être de même pour nSequence. Si CSV est spécifié en termes de secondes, alors nSequence doit également l'être.

Les timelocks relatifs avec CSV sont particulièrement utiles lorsque plusieurs transactions (chaînées) sont créées et signées, mais non propagées, lorsqu'elles sont conservées "hors chaîne". Une transaction enfant ne peut pas être utilisée tant que la transaction parent n'a pas été propagée, extraite et vieillie au moment spécifié dans le timelock relatif. Une application de ce cas d'utilisation peut être vue dans [???](#state_channels) et [???](#lightning_network).

CSV est défini en détail dans [BIP-112, CHECKSEQUENCEVERIFY](https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki).

### Temps médian passé

Dans le cadre de l'activation des verrous temporels relatifs, il y a également eu un changement dans la façon dont le "temps" est calculé pour les verrous temporels (à la fois absolus et relatifs). Dans le bitcoin, il existe une différence subtile, mais très significative, entre le temps du mur et le temps du consensus. Bitcoin est un réseau décentralisé, ce qui signifie que chaque participant a sa propre vision du temps. Les événements sur le réseau ne se produisent pas instantanément partout. La latence du réseau doit être prise en compte dans la perspective de chaque nœud. Finalement, tout est synchronisé pour créer un grand livre commun. Bitcoin atteint un consensus toutes les 10 minutes sur l'état du grand livre tel qu'il existait dans le passé.

Les horodatages définis dans les en-têtes de bloc sont définis par les mineurs. Il existe un certain degré de latitude autorisé par les règles de consensus pour tenir compte des différences de précision d'horloge entre les nœuds décentralisés. Cependant, cela crée une incitation malheureuse pour les mineurs à mentir sur le temps passé dans un bloc afin de gagner des frais supplémentaires en incluant des transactions verrouillées dans le temps qui ne sont pas encore matures. Voir la section suivante pour plus d'informations.

Pour supprimer l'incitation au mensonge et renforcer la sécurité des verrous temporels, un BIP a été proposé et activé en même temps que les BIP des verrous temporels relatifs. Il s'agit de BIP-113, qui définit une nouvelle mesure consensuelle du temps appelée *Median-Time-Past* (temps-médian-passé).

Le Median-Time-Past est calculé en prenant les horodatages des 11 derniers blocs et en trouvant la médiane. Ce temps médian devient alors le temps de consensus et est utilisé pour tous les calculs de verrou temporel. En prenant le point médian d'environ deux heures dans le passé, l'influence de l'horodatage de n'importe quel bloc est réduite. En incorporant 11 blocs, aucun mineur ne peut influencer les horodatages afin de gagner des frais sur les transactions avec un verrou temporel qui n'est pas encore arrivé à échéance.

Median-Time-Past modifie l'implémentation des calculs de temps pour nLocktime, CLTV, nSequence et CSV. L'heure de consensus calculée par Median-Time-Past est toujours d'environ une heure en retard sur l'heure de l'horloge murale. Si vous créez des transactions avec verrou temporel, vous devez en tenir compte lors de l'estimation de la valeur souhaitée à encoder dans nLocktime, nSequence, CLTV et CSV.

Median-Time-Past est spécifié dans [BIP-113](https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki).

### Défense contre le "sniping" avec un verrou temporel

("frais", "sniping")Le "sniping" ou le "ciblage" est un scénario d'attaque théorique, où les mineurs tentant de réécrire des blocs passés "snipe" ou "en ciblant" des frais plus élevés transactions des futurs blocs pour maximiser leur rentabilité.

Par exemple, disons que le bloc le plus élevé existant est le bloc #100 000. Si au lieu d'essayer d'exploiter le bloc #100 001 pour étendre la chaîne, certains mineurs tentent de réexploiter #100 000. Ces mineurs peuvent choisir d'inclure toute transaction valide (qui n'a pas encore été exploitée) dans leur bloc candidat #100 000. Ils n'ont pas à reminer le bloc avec les mêmes transactions. En fait, ils sont incités à sélectionner les transactions les plus rentables (frais les plus élevés par Ko) à inclure dans leur bloc. Ils peuvent inclure toutes les transactions qui se trouvaient dans "l'ancien" bloc #100 000, ainsi que toutes les transactions du mempool actuel. Essentiellement, ils ont la possibilité d'extraire des transactions du "présent" vers le "passé" réécrit lorsqu'ils recréent le bloc #100 000.

Aujourd'hui, cette attaque n'est pas très lucrative, car la récompense de bloc est bien supérieure au total des frais par bloc. Mais à un moment donné dans le futur, les frais de transaction représenteront la majorité de la récompense minière (ou même l'intégralité de la récompense minière). A ce moment-là, ce scénario devient inévitable.

Pour éviter le "ciblage de frais", lorsque Bitcoin Core crée des transactions, il utilise nLocktime pour les limiter au "bloc suivant", par défaut. Dans notre scénario, Bitcoin Core définirait nLocktime sur 100 001 sur toute transaction créée. Dans des circonstances normales, ce nLocktime n'a aucun effet - les transactions ne peuvent être incluses que dans le bloc #100 001 de toute façon ; c'est le bloc suivant.

Mais dans le cadre d'une attaque enfourchement de la chaîne de blocs/double dépense, les mineurs ne seraient pas en mesure d'extraire des transactions à frais élevés du mempool, car toutes ces transactions seraient bloquées dans le temps pour bloquer #100 001. Ils ne peuvent reminer que #100 000 avec les transactions valides à ce moment-là, ne gagnant essentiellement aucun nouveau frais.

Pour ce faire, Bitcoin Core définit le nLocktime sur toutes les nouvelles transactions à <current block # + 1> et définit la nSequence sur toutes les entrées à 0xFFFFFFFE pour activer nLocktime.

## Scripts avec contrôle de flux (clauses conditionnelles)

L'une des fonctionnalités les plus puissantes de Bitcoin Script est le contrôle de flux, également connu sous le nom de clauses conditionnelles. Vous êtes probablement familiarisé avec le contrôle de flux dans divers langages de programmation qui utilisent la construction IF...THEN...ELSE. Les clauses conditionnelles Bitcoin semblent un peu différentes, mais sont essentiellement la même construction.

À un niveau de base, les opcodes conditionnels bitcoin nous permettent de construire un script de rachat qui a deux façons d'être déverrouillé, en fonction d'un résultat TRUE/FALSE d'évaluation d'une condition logique. Par exemple, si x est TRUE, le script de rachat est A et le script de rachat ELSE est B.

De plus, les expressions conditionnelles Bitcoin peuvent être "imbriquées" indéfiniment, ce qui signifie qu'une clause conditionnelle peut en contenir une autre, qui en contient une autre, etc. Le contrôle de flux Bitcoin Script peut être utilisé pour construire des scripts très complexes avec des centaines voire des milliers de chemins d'exécution possibles. . Il n'y a pas de limite à l'imbrication, mais les règles de consensus imposent une limite à la taille maximale, en octets, d'un script.

Bitcoin implémente le contrôle de flux à l'aide des opcodes IF, ELSE, ENDIF et NOTIF. De plus, les expressions conditionnelles peuvent contenir des opérateurs booléens tels que BOOLAND, BOOLOR et NOT.

À première vue, vous pouvez trouver les scripts de contrôle de flux du bitcoin déroutants. En effet, Bitcoin Script est un langage de pile. De la même manière que 1 {plus} 1 semble "renversé" lorsqu'il est exprimé comme 1 1 ADD, les clauses de contrôle de flux dans bitcoin semblent également "renversés".

Dans la plupart des langages de programmation traditionnels (procéduraux), le contrôle de flux ressemble à ceci :

**Pseudocode de contrôle de flux dans la plupart des langages de programmation**

si (condition):  
 code à exécuter lorsque la condition est vraie  
autre:  
 code à exécuter lorsque la condition est fausse  
code à exécuter dans les deux cas

Dans un langage basé sur la pile comme Bitcoin Script, la condition logique vient avant le IF, ce qui le fait apparaître "en arrière", comme ceci :

**Contrôle de flux Bitcoin Script**

condition  
IF  
 code à exécuter lorsque la condition est vraie  
ELSE  
 code à exécuter lorsque la condition est fausse  
ENDIF  
code à exécuter dans les deux cas

Lors de la lecture du script Bitcoin, rappelez-vous que la condition évaluée vient *avant* l'opcode IF.

### Clauses conditionnelles avec les opcodes VERIFY

Une autre forme de conditionnel dans Bitcoin Script est tout opcode qui se termine par VERIFY. Le suffixe VERIFY signifie que si la condition évaluée n'est pas TRUE, l'exécution du script se termine immédiatement et la transaction est considérée comme invalide.

Contrairement à une clause IF, qui offre des chemins d'exécution alternatifs, le suffixe VERIFY agit comme une *clause de garde*, ne continuant que si une condition préalable est remplie.

Par exemple, le script suivant nécessite la signature de Bob et une pré-image (secret) qui produit un hachage spécifique. Les deux conditions doivent être remplies pour le déverrouiller :

**Un script de rachat avec une clause de garde EQUALVERIFY.**

HASH160 <expected hash> EQUALVERIFY <Bob's Pubkey> CHECKSIG

Pour racheter cela, Bob doit construire un script de déverrouillage qui présente une pré-image valide et une signature :

**Un script de déverrouillage pour satisfaire le script de rachat ci-dessus**

<Bob's Sig> <hash pre-image>

Sans présenter la pré-image, Bob ne peut pas accéder à la partie du script qui vérifie sa signature.

Ce script peut être écrit avec un IF à la place :

**Un script de rachat avec une clause de garde IF**

HASH160 <expected hash> EQUAL  
IF  
 <Bob's Pubkey> CHECKSIG  
ENDIF

Le script de déverrouillage de Bob est identique :

**Un script de déverrouillage pour satisfaire le script de rachat ci-dessus**

<Bob's Sig> <hash pre-image>

Le script avec IF fait la même chose que d'utiliser un opcode avec un suffixe VERIFY ; elles fonctionnent toutes deux comme des clauses de garde. Cependant, la construction VERIFY est plus efficace, utilisant deux opcodes de moins.

Alors, quand utilisons-nous VERIFY et quand utilisons-nous IF ? Si tout ce que nous essayons de faire est d'attacher une condition préalable (clause de garde), alors VERIFY est préférable. Si, toutefois, nous voulons avoir plus d'un chemin d'exécution (contrôle de flux), nous avons besoin d'une clause de contrôle de flux IF...ELSE.

Un opcode tel que EQUAL poussera le résultat (TRUE/FALSE) sur la pile, le laissant là pour évaluation par les opcodes suivants. En revanche, le suffixe de l'opcode EQUALVERIFY ne laisse rien sur la pile. Les opcodes qui se terminent par VERIFY ne laissent pas le résultat sur la pile.

### Utilisation du contrôle de flux dans les scripts

Une utilisation très courante du contrôle de flux dans Bitcoin Script consiste à construire un script de rachat qui offre plusieurs chemins d'exécution, chacun une manière différente de racheter l'UTXO.

Regardons un exemple simple, où nous avons deux signataires, Alice et Bob, et l'un ou l'autre est en mesure de racheter. Avec multisig, cela serait exprimé sous la forme d'un script multisig 1 sur 2. Pour des raisons de démonstration, nous ferons la même chose avec une clause IF :

IF  
 <Alice's Pubkey> CHECKSIG  
ELSE  
 <Bob's Pubkey> CHECKSIG  
ENDIF

En regardant ce script de rachat, vous vous demandez peut-être : "Où est la condition ? Il n'y a rien qui précède la clause IF !"

La condition ne fait pas partie du script de rachat. Au lieu de cela, la condition sera proposée dans le script de déverrouillage, permettant à Alice et Bob de "choisir" le chemin d'exécution qu'ils souhaitent.

Alice rachète cela avec le script de déverrouillage :

<Alice's Sig> 1

Le 1 à la fin sert de condition (TRUE) qui fera que la clause IF exécutera le premier chemin de rachat pour lequel Alice a une signature.

Pour que Bob puisse racheter cela, il devrait choisir le deuxième chemin d'exécution en donnant une valeur FALSE à la clause IF :

<Bob's Sig> 0

Le script de déverrouillage de Bob place un 0 sur la pile, ce qui oblige la clause IF à exécuter le deuxième script (ELSE), qui nécessite la signature de Bob.

Puisque les clauses IF peuvent être imbriquées, nous pouvons créer un "labyrinthe" de chemins d'exécution. Le script de déverrouillage peut fournir une "carte" sélectionnant le chemin d'exécution réellement exécuté :

IF  
 script A  
ELSE  
 IF  
 script B  
 ELSE  
 script C  
 ENDIF  
ENDIF

Dans ce scénario, il existe trois chemins d'exécution (script A, script B et script C). Le script de déverrouillage fournit un chemin sous la forme d'une séquence de valeurs TRUE ou FALSE. Pour sélectionner le chemin script B, par exemple, le script de déverrouillage doit se terminer par 1 0 (TRUE, FALSE). Ces valeurs seront poussées sur la pile, de sorte que la deuxième valeur (FALSE) se retrouve en haut de la pile. La clause IF externe extrait la valeur FALSE et exécute la première clause ELSE. Ensuite, la valeur TRUE se déplace vers le haut de la pile et est évaluée par le IF interne (imbriqué), en sélectionnant le chemin d'exécution B.

En utilisant cette construction, nous pouvons créer des scripts de rachat avec des dizaines ou des centaines de chemins d'exécution, chacun offrant une manière différente de racheter l'UTXO. Pour dépenser, nous construisons un script de déverrouillage qui navigue dans le chemin d'exécution en plaçant les valeurs TRUE et FALSE appropriées sur la pile à chaque point de contrôle de flux.

## Exemple de script complexe

Dans cette section, nous combinons de nombreux concepts de ce chapitre en un seul exemple.

Notre exemple utilise l'histoire de Mohammed, le propriétaire d'une entreprise à Dubaï qui exploite une entreprise d'import/export.

Dans cet exemple, Mohammed souhaite construire un compte de capital d'entreprise avec des règles flexibles. Le schéma qu'il crée nécessite différents niveaux d'autorisation en fonction des verrous temporels. Les participants au programme multisig sont Mohammed, ses deux partenaires Saeed et Zaira, et leur avocat Abdul. Les trois partenaires prennent des décisions basées sur une règle de majorité, donc deux des trois doivent être d'accord. Cependant, en cas de problème avec leurs clés, ils souhaitent que leur avocat puisse récupérer les fonds avec l'une des trois signatures d'associés. Enfin, si tous les associés sont indisponibles ou inaptes pendant un certain temps, ils souhaitent que l'avocat puisse gérer directement le compte.

Voici le script de rachat que Mohammed conçoit pour y parvenir (préfixe de numéro de ligne en tant que XX) :

**Variable Multi-Signature avec verrou temporel**

01 IF  
02 IF  
03 2  
04 ELSE  
05 <30 days> CHECKSEQUENCEVERIFY DROP  
06 <Abdul the Lawyer's Pubkey> CHECKSIGVERIFY  
07 1  
08 ENDIF  
09 <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3 CHECKMULTISIG  
10 ELSE  
11 <90 days> CHECKSEQUENCEVERIFY DROP  
12 <Abdul the Lawyer's Pubkey> CHECKSIG  
13 ENDIF

Le script de Mohammed implémente trois chemins d'exécution à l'aide de clauses de contrôle de flux imbriquées IF...ELSE.

Dans le premier chemin d'exécution, ce script fonctionne comme un simple multisig 2 sur 3 avec les trois partenaires. Ce chemin d'exécution se compose des lignes 3 et 9. La ligne 3 définit le quorum du multisig à 2 (2-de-3). Ce chemin d'exécution peut être sélectionné en mettant TRUE TRUE à la fin du script de déverrouillage :

**Script de déverrouillage pour le premier chemin d'exécution (2-de-3 multisig)**

0 <Mohammed's Sig> <Zaira's Sig> TRUE TRUE

Le 0 au début de ce script de déverrouillage est dû à un bogue dans CHECKMULTISIG qui extrait une valeur supplémentaire de la pile. La valeur supplémentaire est ignorée par le CHECKMULTISIG, mais elle doit être présente ou le script échoue. Pousser 0 (habituellement) est une solution de contournement au bogue, comme décrit dans [Un bogue dans l'exécution de CHECKMULTISIG](#multisig_bug).

Le deuxième chemin d'exécution ne peut être utilisé qu'après 30 jours à compter de la création de l'UTXO. À ce moment-là, il faut la signature d'Abdul l'avocat et de l'un des trois partenaires (un multisig 1 sur 3). Ceci est réalisé par la ligne 7, qui fixe le quorum pour le multisig à 1. Pour sélectionner ce chemin d'exécution, le script de déverrouillage se terminerait par FALSE TRUE :

**Script de déverrouillage pour le deuxième chemin d'exécution (Avocat + 1 sur 3)**

0 <Abdul the Lawyer's Sig> <Saeed's Sig> FALSE TRUE

Pourquoi FAUX VRAI ? N'est-ce pas à l'envers ? Parce que les deux valeurs sont poussées sur la pile, avec FALSE poussé en premier, puis TRUE poussé en second. TRUE est donc dépilé *premier* par le premier opcode IF.

Enfin, la troisième voie d'exécution permet à Abdul l'avocat de dépenser les fonds seul, mais seulement après 90 jours. Pour sélectionner ce chemin d'exécution, le script de déverrouillage doit se terminer par FALSE :

**Script de déverrouillage pour le troisième chemin d'exécution (avocat uniquement)**

<Abdul the Lawyer's Sig> FALSE

Essayez d'exécuter le script sur papier pour voir comment il se comporte sur la pile.

Quelques éléments supplémentaires à prendre en compte lors de la lecture de cet exemple. Voyez si vous pouvez trouver les réponses :

* Pourquoi l'avocat ne peut-il pas racheter le troisième chemin d'exécution à tout moment en le sélectionnant avec FALSE sur le script de déverrouillage ?
* Combien de chemins d'exécution peuvent être utilisés 5, 35 et 105 jours, respectivement, après le minage de l'UTXO ?
* Les fonds sont-ils perdus si l'avocat perd sa clé ? Votre réponse change-t-elle si 91 jours se sont écoulés ?
* Comment les associés « remettent-ils à zéro » l'horloge tous les 29 ou 89 jours pour empêcher l'avocat d'accéder aux fonds ?
* Pourquoi certains opcodes CHECKSIG dans ce script ont-ils le suffixe VERIFY alors que d'autres non ?

## Témoin séparé

Le Segregated Witness (segwit) ou le "témoin séparé" est une mise à niveau des règles de consensus bitcoin et du protocole réseau, proposée et mise en œuvre en tant que embranchement convergent BIP-9 qui a été activé sur le réseau principal de bitcoin le 1er août 2017.

En cryptographie, le terme « témoin » est utilisé pour décrire une solution à un puzzle cryptographique. En termes de bitcoin, le témoin satisfait une condition cryptographique placée sur une sortie de transaction non dépensée (UTXO).

Dans le contexte du bitcoin, une signature numérique est *un type de témoin*, mais un témoin est plus largement toute solution qui peut satisfaire les conditions imposées à un UTXO et débloquer cet UTXO pour les dépenses. Le terme "témoin" est un terme plus général pour un "script de déverrouillage" ou "scriptSig".

Avant l'introduction de segwit, chaque entrée d'une transaction était suivie des données témoins qui la déverrouillaient. Les données témoins ont été intégrées à la transaction dans le cadre de chaque entrée. Le terme *témoin séparé*, ou *segwit* en abrégé, signifie simplement séparer la signature ou le script de déverrouillage d'une sortie spécifique. Pensez "scriptSig séparé" ou "signature séparée" dans sa forme la plus simple.

Le témoin séparé (Segregated Witness) est donc une modification architecturale du bitcoin qui vise à déplacer les données témoins du champ scriptSig (script de déverrouillage) d'une transaction vers une structure de données *témoin* distincte qui accompagne une transaction. Les clients peuvent demander des données de transaction avec ou sans les données de témoin qui les accompagnent.

Dans cette section, nous examinerons certains des avantages du témoin séparé, décrirons le mécanisme utilisé pour déployer et mettre en œuvre ce changement d'architecture et démontrerons l'utilisation du témoin séparé dans les transactions et les adresses.

Le témoin séparé est défini par les BIP suivants :

[BIP-141](https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki)

La principale définition du témoin séparé.

[BIP-143](https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki)

Vérification de la signature des transactions pour le programme témoin de la version 0

[BIP-144](https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki)

Services à pair; Nouveaux messages réseau et formats de sérialisation

[BIP-145](https://github.com/bitcoin/bips/blob/master/bip-0145.mediawiki)

Mises à jour de getblocktemplate pour le témoin séparé Segregated Witness (pour minage)

[BIP-173](https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki)

Format d'adresse Base32 pour les sorties témoins natives v0-16

### Pourquoi un témoin séparé ?

Le témoin séparé est un changement architectural qui a plusieurs effets sur l'évolutivité, la sécurité, les incitations économiques et les performances du bitcoin :

Malléabilité de transaction

En déplaçant le témoin en dehors des données de transaction, le hachage de transaction utilisé comme identifiant n'inclut plus les données de témoin. Étant donné que les données témoins sont la seule partie de la transaction qui peut être modifiée par un tiers (voir [Identifiants de transaction](#segwit_txid)), sa suppression supprime également la possibilité d'attaques de malléabilité des transactions. Avec le témoin séparé, les hachages de transaction deviennent immuables par toute personne autre que le créateur de la transaction, ce qui améliore considérablement la mise en œuvre de nombreux autres protocoles qui reposent sur la construction avancée de transactions bitcoin, tels que les canaux de paiement, les transactions chaînées et les réseaux Lightning.

Gestion des versions de script

Avec l'introduction des scripts Segregated Witness (témoin séparé), chaque script de verrouillage est précédé d'un numéro de *version de script*, similaire à la façon dont les transactions et les blocs ont des numéros de version. L'ajout d'un numéro de version de script permet au langage de script d'être mis à niveau de manière rétrocompatible (c'est-à-dire en utilisant des mises à niveau d’embranchement convergent) pour introduire de nouvelles opérandes de script, syntaxe ou sémantique. La possibilité de mettre à niveau le langage de script de manière non perturbatrice accélérera considérablement le taux d'innovation dans le bitcoin.

Mise à l'échelle du réseau et du stockage

Les données témoins contribuent souvent de manière importante à la taille totale d'une transaction. Les scripts plus complexes tels que ceux utilisés pour les canaux multisig ou de paiement sont très volumineux. Dans certains cas, ces scripts représentent la majorité (plus de 75 %) des données d'une transaction. En déplaçant les données témoins en dehors des données de transaction, le témoin séparé améliore l'évolutivité de Bitcoin. Les nœuds peuvent élaguer les données témoins après avoir validé les signatures, ou les ignorer complètement lors de la vérification simplifiée des paiements. Les données témoins n'ont pas besoin d'être transmises à tous les nœuds et n'ont pas besoin d'être stockées sur disque par tous les nœuds.

Optimisation de la vérification de signature

Le témoin séparé met à niveau les fonctions de signature (CHECKSIG, CHECKMULTISIG, etc.) pour réduire la complexité de calcul de l'algorithme. Avant segwit, l'algorithme utilisé pour produire une signature nécessitait un nombre d'opérations de hachage proportionnel à la taille de la transaction. Les calculs de hachage de données ont augmenté en O(n2) par rapport au nombre d'opérations de signature, introduisant une charge de calcul substantielle sur tous les nœuds vérifiant la signature. Avec segwit, l'algorithme est modifié pour réduire la complexité à O(n).

Amélioration de la signature hors ligne

Les signatures des témoins séparés incorporent la valeur (montant) référencée par chaque entrée dans le hachage qui est signé. Auparavant, un dispositif de signature hors ligne, tel qu'un portefeuille matériel, devait vérifier le montant de chaque entrée avant de signer une transaction. Cela était généralement accompli en diffusant une grande quantité de données sur les transactions précédentes référencées en tant qu'entrées. Étant donné que le montant fait désormais partie du hachage d'engagement qui est signé, un appareil hors ligne n'a pas besoin des transactions précédentes. Si les montants ne correspondent pas (sont mal représentés par un système en ligne compromis), la signature sera invalide.

### Comment fonctionne le témoignage séparé

À première vue, le témoignage séparé semble être un changement dans la façon dont les transactions sont construites et donc une fonctionnalité au niveau de la transaction, mais ce n'est pas le cas. Au lieu de cela, le témoin séparé est un changement dans la façon dont les UTXO individuels sont dépensés et est donc une fonctionnalité par sortie.

Une transaction peut dépenser des sorties de témoin séparé ou des sorties traditionnelles (témoin en ligne) ou les deux. Par conséquent, cela n'a pas beaucoup de sens de se référer à une transaction comme une "transaction de témoin séparé". Nous devrions plutôt faire référence à des sorties de transaction spécifiques en tant que "sorties de témoins séparés".

Lorsqu'une transaction passe un UTXO, elle doit fournir un témoin. Dans un UTXO traditionnel, le script de verrouillage nécessite que les données témoins soient fournies *en ligne* dans la partie d'entrée de la transaction qui dépense l'UTXO. Un UTXO de témoin séparé, cependant, spécifie un script de verrouillage qui peut être satisfait avec des données de témoin en dehors de l'entrée (séparées).

### Embranchement convergent (rétrocompatibilité)

Le témoin séparé est un changement significatif dans la manière dont les sorties et les transactions sont architecturées. Un tel changement nécessiterait normalement un changement simultané de chaque nœud et portefeuille Bitcoin pour modifier les règles de consensus, ce que l'on appelle une fourche dure. Au lieu de cela, un témoin séparé est introduit avec un changement beaucoup moins perturbateur, qui est rétrocompatible, connu sous le nom d’embranchement convergent. Ce type de mise à niveau permet aux logiciels non mis à niveau d'ignorer les modifications et de continuer à fonctionner sans aucune interruption.

Les sorties de témoins séparés sont construites de manière à ce que les systèmes plus anciens qui ne sont pas sensibles au segwit puissent toujours les valider. Pour un ancien portefeuille ou nœud, une sortie de témoin séparé ressemble à une sortie que *n'importe qui peut dépenser*. De telles sorties peuvent être dépensées avec une signature vide, donc le fait qu'il n'y ait pas de signature à l'intérieur de la transaction (elle est séparée) n'invalide pas la transaction. Cependant, les portefeuilles et les nœuds de minage plus récents voient la sortie du témoin séparé et s'attendent à trouver un témoin valide pour celle-ci dans les données de témoin de la transaction.

==== Exemples de témoins séparés et exemples de transactions

Examinons quelques-uns de nos exemples de transactions et voyons comment ils changeraient avec le témoin séparé. Nous verrons d'abord comment un paiement Pay-to-Public-Key-Hash (P2PKH) est transformé avec le programme de témoignage séparé. Ensuite, nous examinerons l'équivalent des témoins séparés pour les scripts Pay-to-Script-Hash (P2SH). Enfin, nous verrons comment les deux programmes de témoignage séparé précédents peuvent être intégrés dans un script P2SH.

#### Pay-to-Witness-Public-Key-Hash (P2WPKH)

Dans [???](#cup_of_coffee), Alice a créé une transaction pour payer Bob pour une tasse de café. Cette transaction a créé une sortie P2PKH d'une valeur de 0,015 BTC pouvant être dépensée par Bob. Le script de sortie ressemble à ceci :

**Exemple de script de sortie P2PKH**

DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 EQUALVERIFY CHECKSIG

Avec le témoin séparé, Alice créerait un script Pay-to-Witness-Public-Key-Hash (P2WPKH), qui ressemble à ceci :

**Exemple de script de sortie P2WPKH**

0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

Comme vous pouvez le voir, le script de verrouillage d'une sortie de témoin séparé est beaucoup plus simple qu'une sortie traditionnelle. Il se compose de deux valeurs qui sont transmises à la pile d'évaluation du script. Pour un ancien client Bitcoin (non conscient de la technologie), les deux poussées ressembleraient à une sortie que n'importe qui peut dépenser et ne nécessite pas de signature (ou plutôt, peut être dépensée avec une signature vide). Pour un client segwit plus récent, le premier chiffre (0) est interprété comme un numéro de version (la *version témoin*) et la seconde partie (20 octets) est l'équivalent d'un script de verrouillage connu sous le nom de *programme témoin*. Le programme témoin de 20 octets est simplement le hachage de la clé publique, comme dans un script P2PKH.

Examinons maintenant la transaction correspondante que Bob utilise pour dépenser cette sortie. Pour le script d'origine (nonsegwit), la transaction de Bob devrait inclure une signature dans l'entrée de transaction :

**Transaction décodée montrant une sortie P2PKH dépensée avec une signature**

[...]  
“Vin” : [  
"txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",  
"vout": 0,  
 "scriptSig": “<Bob’s scriptSig>”,  
]  
[...]

Cependant, pour passer la sortie du témoin séparé, la transaction n'a pas de signature dans la partie d'entrée. Au lieu de cela, la transaction de Bob a un scriptSig vide dans les données de transaction (la première partie d'une transaction, qui inclut la partie d'entrée) et inclut sa signature dans les données témoins (la deuxième partie d'une transaction, qui est séparée des données de transaction ):

**Transaction décodée montrant une sortie P2WPKH dépensée avec des données témoins séparées**

[...]  
“Vin” : [  
"txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",  
"vout": 0,  
 "scriptSig": “”,  
]  
[...]  
“witness”: “<Bob’s witness data>”  
[...]

#### Construction de portefeuille de P2WPKH

Il est extrêmement important de noter que P2WPKH ne doit être créé que par le bénéficiaire (destinataire) et non converti par l'expéditeur à partir d'une clé publique connue, d'un script P2PKH ou d'une adresse. Le destinataire n'a aucun moyen de savoir si le portefeuille de l'expéditeur a la capacité de construire des transactions segwit et de dépenser les sorties P2WPKH.

De plus, les sorties P2WPKH doivent être construites à partir du hachage d'une clé publique *compressée*. Les clés publiques non compressées ne sont pas standard dans segwit et peuvent être explicitement désactivées par un futur embranchement convergent. Si le hachage utilisé dans le P2WPKH provient d'une clé publique non compressée, il peut être inutilisable et vous risquez de perdre des fonds. Les sorties P2WPKH doivent être créées par le portefeuille du bénéficiaire en dérivant une clé publique compressée à partir de sa clé privée.

P2WPKH doit être construit par le bénéficiaire (destinataire) en convertissant une clé publique compressée en un hachage P2WPKH. Vous ne devez jamais transformer un script P2PKH, une adresse Bitcoin ou une clé publique non compressée en un script témoin P2WPKH.

#### Pay-to-Witness-Script-Hash (P2WSH)

Le deuxième type de programme témoin correspond à un script Pay-to-Script-Hash (P2SH). Nous avons vu ce type de script dans [Pay-to-Script-Hash (P2SH)](#p2sh). Dans cet exemple, P2SH a été utilisé par la société de Mohammed pour exprimer un script multisignature. Les paiements à l'entreprise de Mohammed étaient encodés avec un script de verrouillage comme celui-ci :

**Exemple de script de sortie P2SH**

HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL

Ce script P2SH fait référence au hachage d'un *script d'échange* qui définit une exigence multisignature 2 sur 5 pour dépenser des fonds. Pour dépenser cette sortie, la société de Mohammed présenterait le script de rachat (dont le hachage correspond au hachage du script dans la sortie P2SH) et les signatures nécessaires pour satisfaire ce script de rachat, le tout dans l'entrée de transaction :

**Transaction décodée montrant qu'une sortie P2SH est dépensée**

[...]  
“Vin” : [  
"txid": "abcdef12345...",  
"vout": 0,  
 "scriptSig": “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”,  
]

Maintenant, regardons comment cet exemple entier serait mis à niveau vers segwit. Si les clients de Mohammed utilisaient un portefeuille compatible segwit, ils effectueraient un paiement, créant une sortie Pay-to-Witness-Script-Hash (P2WSH) qui ressemblerait à ceci :

**Exemple de script de sortie P2WSH**

0 a9b7b38d972cabc7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89

Encore une fois, comme dans l'exemple de P2WPKH, vous pouvez voir que le script équivalent Segregated Witness est beaucoup plus simple et omet les différents opérandes de script que vous voyez dans les scripts P2SH. Au lieu de cela, le programme de témoignage séparé se compose de deux valeurs poussées vers la pile : une version témoin (0) et le hachage SHA256 de 32 octets du script de rachat.

La société de Mohammed peut dépenser la sortie P2WSH en présentant le bon script de rachat et suffisamment de signatures pour le satisfaire. Le script de rachat et les signatures seraient séparés *en dehors* des données de transaction de dépenses dans le cadre des données de témoin. Dans l'entrée de transaction, le portefeuille de Mohammed mettrait un scriptSig vide :

**Transaction décodée montrant une sortie P2WSH dépensée avec des données témoins séparées**

[...]  
“Vin” : [  
"txid": "abcdef12345...",  
"vout": 0,  
 "scriptSig": “”,  
]  
[...]  
“witness”: “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”  
[...]

Alors que P2SH utilise le hachage RIPEMD160(SHA256(script)) de 20 octets, le programme témoin P2WSH utilise un hachage SHA256(script) de 32 octets. Cette différence dans le choix de l'algorithme de hachage est délibérée et offre une sécurité renforcée à P2WSH (128 bits de sécurité dans P2WSH contre 80 bits de sécurité dans P2SH). Il est également utilisé pour différencier les deux types de programmes témoins (P2WPKH et P2WSH) en utilisant la longueur du hachage (voir ci-dessous).

#### Différenciation entre P2WPKH et P2WSH

Dans les deux sections précédentes, nous avons présenté deux types de programmes de témoins : [Pay-to-Witness-Public-Key-Hash (P2WPKH)](#p2wpkh) et [Pay-to-Witness-Script-Hash (P2WSH)](#p2wsh). Les deux types de programmes témoins se composent d'un numéro de version à un seul octet suivi d'un hachage plus long. Ils se ressemblent beaucoup, mais sont interprétés très différemment : l'un est interprété comme un hachage de clé publique, qui est satisfait par une signature et l'autre comme un hachage de script, qui est satisfait par un script de rachat. La différence critique entre eux est la longueur du hachage :

* Le hachage de clé publique dans P2WPKH est de 20 octets
* Le hachage du script dans P2WSH est de 32 octets

C'est la seule différence qui permet à un portefeuille de différencier les deux types de programmes de témoins. En examinant la longueur du hachage, un portefeuille peut déterminer de quel type de programme témoin il s'agit, P2WPKH ou P2WSH.

### Mise à niveau vers un témoin séparé

Comme nous pouvons le voir dans les exemples précédents, la mise à niveau vers les témoins séparés est un processus en deux étapes. Tout d'abord, les portefeuilles doivent créer des sorties spéciales de type segwit. Ensuite, ces sorties peuvent être dépensées par des portefeuilles qui savent comment construire des transactions de témoin séparé. Dans les exemples, le portefeuille d'Alice était sensible au segwit et capable de créer des sorties spéciales avec des scripts de témoignage séparés. Le portefeuille de Bob est également sensible au segwit et capable de dépenser ces sorties. Ce qui n'est peut-être pas évident à partir de l'exemple, c'est qu'en pratique, le portefeuille d'Alice doit *savoir* que Bob utilise un portefeuille sensible au segwit et peut dépenser ces sorties. Sinon, si le portefeuille de Bob n'est pas mis à niveau et qu'Alice essaie d'effectuer des paiements en segwit à Bob, le portefeuille de Bob ne pourra pas détecter ces paiements.

Pour les types de paiement P2WPKH et P2WSH, les portefeuilles de l'expéditeur et du destinataire doivent être mis à niveau pour pouvoir utiliser segwit. De plus, le portefeuille de l'expéditeur doit savoir que le portefeuille du destinataire est sensible au segwit.

Le témoin séparé ne sera pas mis en œuvre simultanément sur l'ensemble du réseau. Au lieu de cela, Segregated Witness est implémenté comme une mise à niveau rétrocompatible, où *les anciens et les nouveaux clients peuvent coexister*. Les développeurs de portefeuille mettront indépendamment à niveau le logiciel de portefeuille pour ajouter des fonctionnalités segwit. Les types de paiement P2WPKH et P2WSH sont utilisés lorsque l'expéditeur et le destinataire sont conscients de segwit. Les P2PKH et P2SH traditionnels continueront de fonctionner pour les portefeuilles non mis à niveau. Cela laisse deux scénarios importants, qui sont abordés dans la section suivante :

* Capacité du portefeuille d'un expéditeur qui n'est pas conscient de segwit à effectuer un paiement sur le portefeuille d'un destinataire qui peut traiter des transactions segwit
* Capacité du portefeuille d'un expéditeur qui est conscient de segwit à reconnaître et à distinguer les destinataires qui sont conscients de segwit et ceux qui ne le sont pas, par leurs *adresses*.

#### Intégration d'un témoin séparé dans P2SH

Supposons, par exemple, que le portefeuille d'Alice n'est pas mis à niveau vers segwit, mais que le portefeuille de Bob est mis à niveau et peut gérer les transactions segwit. Alice et Bob peuvent utiliser les "anciennes" transactions non-segwit. Mais Bob voudrait probablement utiliser segwit pour réduire les frais de transaction, en profitant de la remise qui s'applique aux données des témoins.

Dans ce cas, le portefeuille de Bob peut construire une adresse P2SH qui contient un script segwit à l'intérieur. Le portefeuille d'Alice considère cela comme une adresse P2SH "normale" et peut y effectuer des paiements sans aucune connaissance de segwit. Le portefeuille de Bob peut ensuite dépenser ce paiement avec une transaction segwit, tirant pleinement parti de segwit et réduisant les frais de transaction.

Les deux formes de scripts témoins, P2WPKH et P2WSH, peuvent être intégrées dans une adresse P2SH. Le premier est noté P2SH(P2WPKH) et le second est noté P2SH(P2WSH).

#### Pay-to-Witness-Public-Key-Hash dans Pay-to-Script-Hash

La première forme de script témoin que nous examinerons est P2SH(P2WPKH). Il s'agit d'un programme témoin Pay-to-Witness-Public-Key-Hash, intégré dans un script Pay-to-Script-Hash, afin qu'il puisse être utilisé par un portefeuille qui n'est pas conscient de segwit.

Le portefeuille de Bob construit un programme témoin P2WPKH avec la clé publique de Bob. Ce programme témoin est ensuite haché et le hachage résultant est encodé sous la forme d'un script P2SH. Le script P2SH est converti en une adresse Bitcoin, celle qui commence par un "3", comme nous l'avons vu dans le [Pay-to-Script-Hash (P2SH)](#p2sh) rubrique.

Le portefeuille de Bob commence par le programme témoin P2WPKH que nous avons vu plus tôt :

**Programme de témoins P2WPKH de Bob**

0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

Le programme témoin P2WPKH se compose de la version témoin et du hachage de clé publique de 20 octets de Bob.

Le portefeuille de Bob hache ensuite le programme témoin précédent, d'abord avec SHA256, puis avec RIPEMD160, produisant un autre hachage de 20 octets.

Utilisons bx sur la ligne de commande pour répliquer cela :

**HASH160 du programme témoin P2WPKH**

echo \  
'0 [ab68025513c3dbd2f7b92a94e0581f5d50f654e7]'\  
 | bx script-encode | bx sha256 | bx ripemd160  
3e0547268b3b19288b3adef9719ec8659f4b2b0b

Ensuite, le hachage du script de rachat est converti en une adresse Bitcoin. Utilisons à nouveau bx sur la ligne de commande :

**Adresse P2SH**

echo \  
'3e0547268b3b19288b3adef9719ec8659f4b2b0b' \  
| bx address-encode -v 5  
37Lx99uaGn5avKBxiW26HjedQE3LrDCZru

Maintenant, Bob peut afficher cette adresse pour que les clients paient leur café. Le portefeuille d'Alice peut effectuer un paiement au 37Lx99uaGn5avKBxiW26HjedQE3LrDCZru, comme il le ferait avec n'importe quelle autre adresse Bitcoin.

Pour payer Bob, le portefeuille d'Alice verrouillerait la sortie avec un script P2SH :

HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b EQUAL

Même si le portefeuille d'Alice ne prend pas en charge segwit, le paiement qu'il crée peut être dépensé par Bob avec une transaction segwit.

#### Pay-to-Witness-Script-Hash dans Pay-to-Script-Hash

De même, un programme témoin P2WSH pour un script multisig ou un autre script compliqué peut être intégré dans un script et une adresse P2SH, permettant à n'importe quel portefeuille d'effectuer des paiements compatibles avec segwit.

Comme nous l'avons vu dans [Pay-to-Witness-Script-Hash (P2WSH)](#p2wsh), la société de Mohammed utilise des paiements de témoins séparés avec des scripts multisignatures. Pour permettre à tout client de payer son entreprise, que ses portefeuilles soient ou non mis à niveau pour segwit, le portefeuille de Mohammed peut intégrer le programme témoin P2WSH dans un script P2SH.

Tout d'abord, le portefeuille de Mohammed hache le script de rachat avec SHA256 (une seule fois). Utilisons bx pour faire cela sur la ligne de commande :

**Le portefeuille de Mohammed crée un programme témoin P2WSH**

echo \  
2 \ [04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] \  
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] \  
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] \  
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] \  
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] \  
5 CHECKMULTISIG \  
| bx script-encode | bx sha256  
9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73

Ensuite, le script de rachat haché est transformé en un programme témoin P2WSH :

0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73

Ensuite, le programme témoin lui-même est haché avec SHA256 et RIPEMD160, produisant un nouveau hachage de 20 octets, tel qu'utilisé dans le P2SH traditionnel. Utilisons bx sur la ligne de commande pour faire cela :

**Le HASH160 du programme témoin P2WSH**

echo \  
'0 [9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73]'\  
 | bx script-encode | bx sha256 | bx ripemd160  
86762607e8fe87c0c37740cddee880988b9455b2

Ensuite, le portefeuille construit une adresse Bitcoin P2SH à partir de ce hachage. Encore une fois, nous utilisons bx pour calculer sur la ligne de commande :

**P2SH Adresse bitcoin**

echo \  
'86762607e8fe87c0c37740cddee880988b9455b2'\  
 | bx address-encode -v 5  
3Dwz1MXhM6EfFoJChHCxh1jWHb8GQqRenG

Désormais, les clients de Mohammed peuvent effectuer des paiements à cette adresse sans avoir besoin de prendre en charge segwit. Pour envoyer un paiement à Mohammed, un portefeuille verrouillerait la sortie avec le script P2SH suivant :

**Script P2SH utilisé pour verrouiller les paiements sur le multisig de Mohammed**

HASH160 86762607e8fe87c0c37740cddee880988b9455b2 EQUAL

La société de Mohammed peut alors construire des transactions segwit pour dépenser ces paiements, en tirant parti des fonctionnalités segwit, notamment des frais de transaction moins élevés.

#### Adresses de témoins séparés

Même après l'activation de segwit, il faudra un certain temps avant que la plupart des portefeuilles soient mis à niveau. Dans un premier temps, segwit sera intégré à P2SH, comme nous l'avons vu dans la section précédente, pour faciliter la compatibilité entre les portefeuilles prenant en charge et ignorants de segwit.

Cependant, une fois que les portefeuilles prennent largement en charge segwit, il est logique d'encoder les scripts témoins directement dans un format d'adresse natif conçu pour segwit, plutôt que de l'intégrer dans P2SH.

Le format d'adresse segwit natif est défini dans BIP-173 :

[BIP-173](https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki)

Format d'adresse Base32 pour les sorties témoins natives v0-16

BIP-173 encode uniquement les scripts témoins (P2WPKH et P2WSH). Il n'est pas compatible avec les scripts P2PKH ou P2SH non-segwit. BIP-173 est un encodage Base32 à somme de contrôle, par rapport à l'encodage Base58 d'une adresse Bitcoin "traditionnelle". Les adresses BIP-173 sont également appelées adresses *bech32*, prononcées "beh-ch trente-deux", faisant allusion à l'utilisation d'un algorithme de détection d'erreur "BCH" et d'un ensemble de codage à 32 caractères.

Les adresses BIP-173 utilisent un jeu de 32 caractères alphanumériques en minuscules uniquement, soigneusement sélectionnés pour réduire les erreurs de lecture ou de frappe. En choisissant un jeu de caractères en minuscules uniquement, bech32 est plus facile à lire, à parler et 45 % plus efficace pour encoder dans les codes QR.

L'algorithme de détection d'erreurs BCH est une grande amélioration par rapport à l'algorithme de somme de contrôle précédent (de Base58Check), permettant non seulement la détection mais aussi la *correction* des erreurs. Les interfaces de saisie d'adresse (telles que les champs de texte dans les formulaires) peuvent détecter et mettre en évidence le caractère le plus susceptible d'être mal saisi lorsqu'elles détectent une erreur.

De la spécification BIP-173, voici quelques exemples d'adresses bech32 :

Mainnet P2WPKH

bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4

Testnet P2WPKH

tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx

Mainnet P2WSH

bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3

Testnet P2WSH

tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7

Comme vous pouvez le voir dans ces exemples, une chaîne segwit bech32 contient jusqu'à 90 caractères et se compose de trois parties :

La partie lisible par l'homme

Ce préfixe "bc" ou "tb" identifiant mainnet ou testnet

Le séparateur

Le chiffre "1", qui ne fait pas partie du jeu de codage à 32 caractères et ne peut apparaître qu'à cette position en tant que séparateur

La partie données

Un minimum de 6 caractères alphanumériques, le script témoin encodé par somme de contrôle

À l'heure actuelle, seuls quelques portefeuilles acceptent ou produisent des adresses segwit bech32 natives, mais à mesure que l'adoption de segwit augmente, vous les verrez de plus en plus souvent.

[table\_title](#segwit_addresses) affiche les adresses bitcoin non-segwit (héritées) et segwit.

Bitcoin non-segwit (hérité) et adresses segwit

|  |  |  |
| --- | --- | --- |
| Type | Encodage | Préfixe |
| Adresse P2PKH héritée | Base58 | 1 |
| Ancienne adresse Testnet P2PKH | Base58 | m ou n |
| Ancienne adresse P2SH | Base58 | 3 |
| Ancienne adresse Testnet P2SH | Base58 | 2 |
| Adresse Segwit P2SH(P2WPKH) imbriquée (intégrée) | Base58 | 3 |
| Adresse Segwit P2SH(P2WSH) imbriquée (intégrée) | Base58 | 3 |
| Adresse native Segwit P2WPKH | Bech32 | bc1 |
| Adresse native Segwit Testnet P2WPKH | Bech32 | tb1 |
| Adresse native Segwit P2WSH | Bech32 | bc1 |
| Adresse native Segwit Testnet P2WSH | Bech32 | tb1 |

#### Identifiants de transaction

L'un des plus grands avantages des témoin séparés est qu'il élimine la malléabilité des transactions tierces.

Avant segwit, les transactions pouvaient voir leurs signatures subtilement modifiées par des tiers, changeant leur ID de transaction (hachage) sans changer aucune propriété fondamentale (entrées, sorties, montants). Cela a créé des opportunités d'attaques par déni de service ainsi que des attaques contre des logiciels de portefeuille mal écrits qui supposaient que les hachages de transaction non confirmés étaient immuables.

Avec l'introduction des témoins séparés, les transactions ont deux identifiants, txid et wtxid. L'identifiant de transaction traditionnel txid est le hachage double SHA256 de la transaction sérialisée, sans les données témoins. Une transaction wtxid est le hachage double SHA256 du nouveau format de sérialisation de la transaction avec des données témoins.

Le txid traditionnel est calculé exactement de la même manière qu'avec une transaction non segwit. Cependant, étant donné qu'une transaction segwit pure (une transaction qui ne contient que des entrées segwit) a des scriptSig vides dans chaque entrée, aucune partie de la transaction ne peut être modifiée par un tiers. Par conséquent, dans une transaction segwit pure, le txid est immuable par un tiers, même lorsque la transaction n'est pas confirmée.

Le wtxid est comme un identifiant "étendu", en ce que le hachage incorpore également les données du témoin. Si une transaction est transmise sans données témoins, alors le wtxid et le txid sont identiques. Notez que puisque le wtxid inclut des données témoins (signatures) et que les données témoins peuvent être malléables, le wtxid doit être considéré comme malléable jusqu'à ce que la transaction soit confirmée. Seul le txid d'une pure transaction segwit peut être considéré comme immuable par des tiers.

Les transactions de témoin séparé ont deux identifiants : txid et wtxid. Le txid est le hachage de la transaction sans les données témoins et le wtxid est le hachage incluant les données témoins. Seules les transactions segwit pures (transactions qui ne contiennent que des entrées segwit) ont un txid qui n'est pas sensible à la malléabilité des transactions tierces.

### Nouvel algorithme de signature des témoins séparés

Le témoin séparé modifie la sémantique des quatre fonctions de vérification de signature (CHECKSIG, CHECKSIGVERIFY, CHECKMULTISIG et CHECKMULTISIGVERIFY), changeant la façon dont un hachage d'engagement de transaction est calculé.

Les signatures dans les transactions bitcoin sont appliquées sur un *hachage d'engagement*, qui est calculé à partir des données de transaction, verrouillant des parties spécifiques des données indiquant l'engagement du signataire envers ces valeurs. Par exemple, dans une signature de type SIGHASH\_ALL simple, le hachage d'engagement inclut toutes les entrées et sorties.

Malheureusement, la façon dont le hachage d'engagement a été calculé a introduit la possibilité qu'un nœud vérifiant la signature puisse être forcé d'effectuer un nombre important de calculs de hachage. Plus précisément, les opérations de hachage augmentent en O(n2) par rapport au nombre d'opérations de signature dans la transaction. Un attaquant pourrait donc créer une transaction avec un très grand nombre d'opérations de signature, obligeant l'ensemble du réseau Bitcoin à effectuer des centaines ou des milliers d'opérations de hachage pour vérifier la transaction.

Segwit représentait une opportunité de résoudre ce problème en modifiant la façon dont le hachage d'engagement est calculé. Pour les programmes témoins segwit version 0, la vérification de la signature se produit à l'aide d'un algorithme de hachage d'engagement amélioré, comme spécifié dans BIP-143.

Le nouvel algorithme atteint deux objectifs importants. Premièrement, le nombre d'opérations de hachage augmente d'un O(n) beaucoup plus graduel au nombre d'opérations de signature, réduisant la possibilité de créer des attaques par déni de service avec des transactions trop complexes. Deuxièmement, le hachage d'engagement inclut désormais également la valeur (montants) de chaque entrée dans le cadre de l'engagement. Cela signifie qu'un signataire peut s'engager sur une valeur d'entrée spécifique sans avoir besoin de "récupérer" et de vérifier la transaction précédente référencée par l'entrée. Dans le cas d'appareils hors ligne, tels que les portefeuilles matériels, cela simplifie grandement la communication entre l'hôte et le portefeuille matériel, supprimant la nécessité de diffuser les transactions précédentes pour validation. Un portefeuille matériel peut accepter la valeur d'entrée "telle qu'elle est indiquée" par un hôte non approuvé. Étant donné que la signature n'est pas valide si cette valeur d'entrée n'est pas correcte, le portefeuille matériel n'a pas besoin de valider la valeur avant de signer l'entrée.

### Incitations économiques pour les témoins isolés

Les nœuds miniers Bitcoin et les nœuds complets entraînent des coûts pour les ressources utilisées pour prendre en charge le réseau Bitcoin et la chaîne de blocs. À mesure que le volume de transactions bitcoin augmente, le coût des ressources (CPU, bande passante réseau, espace disque, mémoire) augmente également. Les mineurs sont indemnisés pour ces coûts par des frais proportionnels à la taille (en octets) de chaque transaction. Les nœuds complets non miniers ne sont pas rémunérés, ils encourent donc ces coûts parce qu'ils ont besoin d'exécuter un nœud d'index complet faisant autorité, peut-être parce qu'ils utilisent le nœud pour exploiter une entreprise bitcoin.

Sans frais de transaction, la croissance des données bitcoin augmenterait sans doute de façon spectaculaire. Les frais sont destinés à aligner les besoins des utilisateurs de bitcoins sur le fardeau que leurs transactions imposent au réseau, grâce à un mécanisme de découverte des prix basé sur le marché.

Le calcul des frais en fonction de la taille de la transaction traite toutes les données de la transaction comme ayant un coût égal. Mais du point de vue des nœuds complets et des mineurs, certaines parties d'une transaction entraînent des coûts beaucoup plus élevés. Chaque transaction ajoutée au réseau Bitcoin affecte la consommation de quatre ressources sur les nœuds :

Espace disque

Chaque transaction est stockée dans la chaîne de blocs, ajoutant à la taille totale de la chaîne de blocs. La chaîne de blocs est stockée sur disque, mais le stockage peut être optimisé en « élaguant » (supprimant) les anciennes transactions.

CPU

Chaque transaction doit être validée, ce qui nécessite du temps CPU.

Bande passante

Chaque transaction est transmise (par propagation d'inondation) sur le réseau au moins une fois. Sans aucune optimisation du protocole de propagation de bloc, les transactions sont à nouveau transmises dans le cadre d'un bloc, ce qui double l'impact sur la capacité du réseau.

Mémoire

Les nœuds qui valident les transactions conservent l'index UTXO ou l'intégralité de l'ensemble UTXO en mémoire pour accélérer la validation. Étant donné que la mémoire est au moins un ordre de grandeur plus chère que le disque, la croissance de l'ensemble UTXO contribue de manière disproportionnée au coût d'exécution d'un nœud.

Comme vous pouvez le voir dans la liste, toutes les parties d'une transaction n'ont pas le même impact sur le coût d'exécution d'un nœud ou sur la capacité du bitcoin à évoluer pour prendre en charge davantage de transactions. La partie la plus coûteuse d'une transaction sont les sorties nouvellement créées, car elles sont ajoutées à l'ensemble UTXO en mémoire. En comparaison, les signatures (c'est-à-dire les données témoins) ajoutent le moins de charge au réseau et au coût d'exécution d'un nœud, car les données témoins ne sont validées qu'une seule fois et ne sont jamais réutilisées. De plus, immédiatement après avoir reçu une nouvelle transaction et validé les données témoins, les nœuds peuvent supprimer ces données témoins. Si les frais sont calculés en fonction de la taille de la transaction, sans faire de distinction entre ces deux types de données, les incitations du marché en matière de frais ne sont pas alignées sur les coûts réels imposés par une transaction. En fait, la structure tarifaire actuelle encourage en fait le comportement inverse, car les données des témoins constituent la plus grande partie d'une transaction.

Les incitations créées par les frais sont importantes car elles affectent le comportement des portefeuilles. Tous les portefeuilles doivent mettre en œuvre une stratégie pour assembler les transactions qui prend en considération un certain nombre de facteurs, tels que la confidentialité (réduction de la réutilisation des adresses), la fragmentation (faire beaucoup de monnaie) et les frais. Si les frais motivent massivement les portefeuilles à utiliser le moins d'entrées possible dans les transactions, cela peut conduire à la sélection d'UTXO et à la modification des stratégies d'adresse qui gonflent par inadvertance l'ensemble UTXO.

Les transactions consomment les UTXO dans leurs entrées et créent de nouveaux UTXO avec leurs sorties. Par conséquent, une transaction qui a plus d'entrées que de sorties entraînera une diminution de l'ensemble UTXO, tandis qu'une transaction qui a plus de sorties que d'entrées entraînera une augmentation de l'ensemble UTXO. Considérons la *difference* entre les entrées et les sorties et appelons cela le "Net-new-UTXO". Il s'agit d'une mesure importante, car elle nous indique l'impact qu'une transaction aura sur la ressource la plus chère à l'échelle du réseau, l'ensemble UTXO en mémoire. Une transaction avec Net-new-UTXO positif ajoute à ce fardeau. Une transaction avec un Net-new-UTXO négatif réduit la charge. Nous voudrions donc encourager les transactions qui sont soit Net-new-UTXO négatives, soit neutres avec zéro Net-new-UTXO.

Examinons un exemple des incitations créées par le calcul des frais de transaction, avec et sans témoin séparé. Nous examinerons deux transactions différentes. La transaction A est une transaction à 3 entrées et 2 sorties, qui a une métrique Net-new-UTXO de -1, ce qui signifie qu'elle consomme un UTXO de plus qu'elle n'en crée, ce qui réduit l'UTXO défini d'un. La transaction B est une transaction à 2 entrées et 3 sorties, qui a une métrique Net-new-UTXO de 1, ce qui signifie qu'elle ajoute un UTXO à l'ensemble UTXO, imposant un coût supplémentaire à l'ensemble du réseau Bitcoin. Les deux transactions utilisent des scripts multisignatures (2 sur 3) pour démontrer comment des scripts complexes augmentent l'impact d'un témoin séparé sur les frais. Supposons un taux de transaction de 30 satoshi par octet et une remise de 75 % sur les données des témoins :

Sans témoin séparé

Frais de transaction A : 28 590 satoshi

Frais de transaction B : 20 760 satoshi

Avec témoin séparé

Frais de transaction A : 12 255 satoshi

Frais de transaction B : 10 425 satoshi

Les deux transactions sont moins coûteuses lorsque le témoin séparé est mis en œuvre. En comparant les coûts entre les deux transactions, nous voyons qu'avant le témoin séparé, la transaction avec le Net-new-UTXO positif permet de réaliser d'importantes économies. Avec le témoin séparé, la différence de coût diminue considérablement en termes absolus et relatifs. Bien qu'il faudrait que les intrants deviennent moins chers que les extrants pour inciter à la consolidation de l'ensemble UTXO, cette remise réduit l'incitation à créer de nouveaux UTXO afin d'éviter d'utiliser plus d'intrants.

Segregated Witness a donc deux effets principaux sur les frais payés par les utilisateurs de bitcoin. Premièrement, segwit réduit le coût global des transactions en actualisant les données des témoins et en augmentant la capacité de la blockchain Bitcoin. Deuxièmement, la remise de segwit sur les données des témoins atténue partiellement un désalignement des incitations qui peut avoir créé par inadvertance plus de ballonnement dans l'ensemble UTXO.