Valider qu'un contrat Ethereum est bien ce qu'il prétend être

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 29 mai 2016

https://www.bortzmeyer.org/valider-contrats-ethereum.html

La chaîne de blocs Ethereum permet bien plus que de gérer de l'argent : elle autorise la création de **programmes** (appelés **contrats**) qui peuvent être invoqués par des utilisateurs, qui envoient leur argent au contrat, espérant que celui-ci va bien faire ce qui est promis. Mais comment le vérifier ?

Les contrats (le terme marketing est "*smart contracts*") sont des programmes informatiques : dans le cas le plus courant, ils sont écrits par un programmeur dans un langage « de haut niveau » puis compilé dans le langage machine d'Ethereum.

Le contrat, comme son nom l'indique, est censé faire quelque chose. Imaginons par exemple un serveur de clés PGP dans la chaîne de blocs : il est censé recevoir des clés, les stocker, et les restituer à la demande. Les gens qui stockent des clés paient (en ethers) pour cela. Il est légitime de se demander « est-ce que ce programme va bien faire ce qui est prévu, ou bien va-t-il soudainement détruire toutes les clés, et j'aurais alors dépensé mes ethers pour rien ? » C'est d'autant plus critique qu'il existe des contrats qui brassent des sommes d'argent importantes. Les utilisateurs se fient typiquement à la documentation publique de ce contrat, alors qu'on ne peut pas être sûr qu'elle corresponde au contrat réel (soit par suite d'une erreur, soit par suite d'une volonté délibérée). Le code s'exécute dans la chaîne de blocs, sans intervention humaine (c'est le but des "*smart contracts*"). Comment valider ce code ? C'est à mon avis **le** gros problème de ces « contrats intelligents ».

Arrivé à ce stade, mes lecteurs partisans du logiciel libre ont déjà une réponse : on distribue le code source (rappelez-vous qu'un contrat n'est qu'un programme), des gens le liront et le vérifieront. Si suffisamment de gens compétents le lisent, le contrat est vérifié, on peut lui faire confiance. Mais c'est très insuffisant.

Déjà, analyser un code source complexe peut être difficile (on peut espérer que les auteurs de contrats, voulant inspirer confiance, essaieront d'écrire du code clair et lisible). Il est clair que la disponibilité du code source est un pré-requis : s'il n'est pas public, il ne faut jamais envoyer de l'argent à ce contrat. Seulement, ça ne suffit pas : ce qui est stocké et exécuté sur la blockchain, c'est du langage machine d'Ethereum (à peu près du même niveau d'abstraction que le langage des processeurs matériels). Ça, c'est vraiment dur à valider. Pour donner un exemple, voici un contrat très très simple écrit en Solidity, il stocke une valeur et permet de la récupérer :

```
contract SimpleStorage {
  uint storedData;

  function set(uint x) {
    storedData = x;
  }

  function get() constant returns (uint retVal) {
    return storedData;
  }
}
```

Il est trivial à analyser et à vérifier. Seulement, ce n'est pas ça qu'exécute la chaîne de blocs, mais un code machine. Voici sa version en assembleur, produite par l'option `--asm` du compilateur `solc` : (en ligne sur `https://www.bortzmeyer.org/files/storage.asm`) (je ne l'ai pas incluse dans cet article, elle est trop longue). Mais ce n'est pas non plus ce que stocke la chaîne de blocs, en fait, on a juste les "*opcodes*", sans les jolis commentaires et tout :

```
% solc --opcodes storage.sol

======= SimpleStorage =======
Opcodes:
PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x97 DUP1 PUSH1 0x10 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN PUSH1 0x60 PUSH1
```

À noter que c'était sans l'optimisation. Celle-ci réduit sérieusement le nombre d'instructions :

```
% solc --opcodes storage.sol |wc -w
137
% solc --opcodes --optimize storage.sol |wc -w
73
```

Si vous voulez observer ce contrat en ligne, il a été déployé sur la chaîne de blocs, à l'adresse `0x48b4cb193b587c6f2dab1a9123a7bd5e7d490ced` (et ça m'a coûté 69 558 unités d'essence, payées 0,00139116 ethers). Le résultat est visible sur les explorateurs de la chaîne, comme etherscan.io `<http://etherscan.io/address/0x48b4cb193b587c6f2dab1a9123a7bd5e7d490ced>`, Etherchain `<https://etherchain.org/account/0x48b4cb193b587c6f2dab1a9123a7bd5e7d490ced>` ou ether.camp `<https://live.ether.camp/account/48b4cb193b587c6f2dab1a9123a7bd5e7d490ced>`. Sur Etherscan.io, la fonction "*Switch To Opcodes View*" vous permet de passer des instructions en assembleur au code binaire. Sur Ether.camp, ce sont les liens "*code*" et "*asm*". Au passage, si vous voulez interagir avec ce contrat (après l'avoir vérifié !), les instructions en console sont :

```
storage = eth.contract([{"constant":false,"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[]
```

Et cela permet de stocker et de récupérer des valeurs depuis la console de `geth` (cela vous coûtera 26 563 unités d'essence, soit, aujourd'hui, 0,00053126 ether) :

```
> storage.set(42,{from:eth.accounts[0]})
"0x36084ff3d5bf0235326277b2d5ba30b1964f6d8b3720b9aef0578faeaeb2678b"
...
> storage.get()
42
```

Quittons ce détour technique et revenons à la question de fond, la vérification. Cette préoccupation est relativement fréquente dans le monde Ethereum. Il y a quelques mois, elle ne suscitait guère d'intérêt. Il y a plusieurs solutions pour vérifier un contrat avant de lui confier ses précieux ethers :

— On compile le code source soi-même et on vérifie que le binaire correspond. S'il correspond, on peut être rassuré : sauf bogue/malhonnêteté <http://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf> du compilateur, le contrat fait bien ce que dit le code source. C'est ainsi que fonctionnent les services de vérification de code des explorateurs (comme Etherscan <https://etherscan.io/verifyContract> ou Etherchain <https://etherchain.org/account_verify/info>), ce qui ajoute d'ailleurs une confiance supplémentaire : il faut faire confiance à ce service.
— On vérifie directement le code machine, par rétro-ingénierie, comme le font les "*reverseurs*" qui analysent le logiciel malveillant (pour lequel ils n'ont évidemment pas les sources). C'est très sûr mais très complexe, puisqu'on travaille sur du matériau de très bas niveau. À ma connaissance, personne n'a encore développé cette compétence mais cela se fera sans doute, étant donné l'inquiétude sur la sécurité des contrats <http://vessenes.com/ethereum-contracts-are-going-to-be-ca>. Cela fera de beaux articles de blog :-)
— On copie le code machine et on fait tourner le contrat dans une chaîne de blocs de test (comme Morden <https://github.com/ethereum/wiki/wiki/Morden>), en utilisant ce contrat comme une boîte noire : on n'essaie pas de comprendre le code, mais on voit s'il répond bien aux fonctions qu'on appelle. Cette méthode est plus simple mais limitée : si le contrat est complexe, on aura du mal à tester tous les cas. Et, même avec un contrat trivial comme celui donné en exemple plus haut, on ne peut pas ainsi découvrir si certaines valeurs (comme 42...) ne vont pas activer la porte dérobée.
— Il y a aussi cette technique <https://github.com/ethereum/solidity/issues/611>, spécifique à Solidity, que je n'ai pas encore creusée.

Dans le cas de la première méthode, la recompilation, si le binaire obtenu correspond à celui déployé, on est tranquille. Mais s'il ne correspond pas ? Est-ce que cela veut dire que quelqu'un essaie de nous tromper ? Non, car le code obtenu dépend du compilateur, et de sa version exacte. Si le binaire correspond, on est rassuré, s'il ne correspond pas, on n'est pas plus avancé. Et le problème devient plus compliqué avec le temps (certains contrats peuvent être éternels) : comment on retrouve le compilateur utilisé deux ans après ? Il faudrait que le code soit étiqueté avec la version du compilateur, ce qui n'est pas le cas <https://github.com/ethereum/solidity/issues/529>, et qu'on dispose d'une copie de cet ancien compilateur.

Bref, il n'y a pas de solution parfaite aujourd'hui. Lorsque vous mettez de l'argent dans un contrat sérieux, vous avez forcément une incertitude. Vous pouvez la réduire en regardant les vérifications faites par d'autres. Pour The DAO, le code source est en ligne <https://github.com/slockit/DAO>. Il existe une documentation sur cette vérification <https://github.com/slockit/DAO/wiki/The-DAO-v1.0-Code> et des détails <https://blog.daohub.org/daohub-verification-of-0xbb9bc244d798123fde783f
Et le code a été vérifié par certains explorateurs, comme Etherscan <http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code>.

Et, pour finir, une copie d'écran d'un cas où ça a marché, la vérification du contrat montré plus haut :