Ethereum : chasse aux contrats intelligents vulnérables

Korantin Auguste
contact@palkeo.com

Indépendant : www.palkeo.com

1 Introduction

Cet article vise à présenter mes recherches autour de l'analyse de « contrats intelligents » sur la blockchain Ethereum.

Dans une première partie je présenterai le contexte, à savoir le fonctionnement de la blockchain Ethereum, des contrats intelligents, et les problématiques qu'ils présentent en termes de sécurité.

Puis je présenterai Pakala : un outil qui utilise de l'exécution symbolique pour chercher des contrats intelligents avec des bugs critiques (en l'occurrence des programmes qui pourraient envoyer plus d'argent qu'ils n'en reçoivent).

Je parlerai ensuite d'une seconde analyse que j'ai effectuée : au lieu de scanner le code des contrats intelligents à la recherche de bugs, j'ai aussi parcouru l'historique de la blockchain pour chercher des motifs particuliers qui trahiraient une attaque et une vidange des fonds de contrats.

Finalement je présenterai les résultats que j'ai obtenus via chacune de mes méthodes, mais aussi des exemples concrets de classes de vulnérabilités ou de contrats qui sont revenus sans cesse lors de mes analyses.

2 Contexte

Avant d'entrer dans le vif du sujet, il faut savoir que mon introduction à Ethereum se concentre surtout sur les aspects techniques qui nous seront utiles par la suite.

Il y a énormément d'autres choses à comprendre et à découvrir. Si le sujet vous intéresse je recommande les livres d'Andreas Antonopoulos (disponibles sur GitHub) : *Mastering Bitcoin* [1] et *Mastering Ethereum* [2].

2.1 Ethereum

Ethereum est un registre distribué (*distributed ledger*) comparable à Bitcoin, qui apporte un certain nombre d'innovations :

Langage de script Turing-complet. Bitcoin possède déjà un langage de script minimaliste (qui permet par exemple de programmer des portefeuilles contrôlés par plusieurs clés à la fois), mais il est très limité et n'est pas Turing-complet. Ethereum permet aux contrats de faire bien plus de choses, et d'interagir entre eux.

Mécanisme de consensus. La sécurité des registres distribués est fondée sur le mécanisme de consensus qu'ils utilisent : il s'agit pour un ensemble d'acteurs de s'accorder sur l'état du système à un temps T, sachant que certains acteurs peuvent être malveillants.

Bitcoin tout comme Ethereum utilise actuellement un système de preuve de calcul, c'est-à-dire que pour participer à la sécurisation du réseau (en générant des blocs, qui forment un état cohérent du système) il faut prouver qu'on a dépensé de l'énergie, et il est donc impossible de générer des blocs à volonté.

Toutefois, Ethereum souhaite à terme utiliser un mécanisme de preuve d'enjeu : les mineurs devront bloquer de grosses sommes d'argent (détruites s'ils se comportent mal) au lieu de gaspiller de l'énergie avec ces calculs.

Cela évitera à terme au réseau de consommer de l'énergie à outrance, mais offrira d'autres propriétés intéressantes, comme la *finalité* : un mineur sur un réseau en preuve de travail qui aurait énormément de puissance de calcul pourrait générer une succession de blocs, puis revenir dessus à partir d'un bloc plus vieux et générer un historique différent (*double-spending*). Or, avec la preuve d'enjeu on peut interdire à des mineurs de construire plus d'une succession valide de blocs signés par eux, sous peine de détruire leur caution. Cela permettra à une transaction d'un bloc validé d'être considérée comme irréversible, alors que pour Bitcoin il faut attendre plusieurs blocs avant de considérer qu'une attaque serait trop coûteuse.

Durée entre les blocs. Avec Bitcoin, le réseau vise une durée entre blocs de 10 minutes. Il faut attendre le prochain bloc pour avoir une chance de voir ses transactions validées, donc en général plusieurs minutes.

Sur Ethereum, la durée entre chaque bloc est très courte (sans nuire à la sécurité, grâce à un système de blocs « oncles » qui rémunèrent quand même les mineurs si des blocs se succèdent trop vite) : on passe à une durée de 15 secondes en moyenne.

UTXO vs. état global. Bitcoin utilise un système de *unspent transaction output (UTXO)*, c'est-à-dire que toute sortie d'argent depuis une adresse

nécessite de référencer une ou plusieurs entrées d'argent vers cette même adresse, qui n'ont pas encore été dépensées.

Ethereum a un modèle plus complexe (décrit dans le *Yellow Paper* [5]) : en plus d'une chaîne de blocs avec l'historique des transactions, il maintient une structure de données qui évolue à chaque transaction, l'état du système.

Ethereum définit une fonction de transition Υ qui permet de passer d'un état du système à l'état suivant en y appliquant une transaction. Le réseau démarre au premier bloc avec un état vide, et en appliquant successivement toutes les transactions on obtient l'état courant.

Contrairement à Bitcoin où il n'y a pas de notion de « solde du compte » (la seule chose qui compte c'est de dépenser des UTXO), l'état courant d'Ethereum contient le solde de tous les comptes, et une transaction consiste simplement à décrémenter un solde et en incrémenter un autre.

Unités de compte. Bitcoin utilise comme unité principale le Bitcoin, qui se décompose en 10^8 unités indivisibles, les Satoshis. De manière analogue, Ethereum utilise comme unité les Ethers, qui se décomposent en 10^{18} unités indivisibles, les Wei.

2.2 Contrats intelligents

En plus d'avoir des adresses « classiques » contrôlées par des humains, Ethereum définit des adresses « contrats intelligents » dont personne ne possède la clé et dont le code est immuable [1]. Lorsqu'ils sont appelés, ces contrats exécutent du code, qui est défini et exécuté dans la fonction de transition d'état Υ.

Cet état global permet de savoir efficacement combien possède chaque compte. Mais il contient également le code des contrats intelligents, et du stockage arbitraire qui peut être utilisé par ces derniers.

En pratique, les contrats intelligents sont donc des programmes (du code exécutable) qui sont enregistrés dans la blockchain Ethereum. Ils sont tous publiquement accessibles.

Lesdits programmes sont implémentés en bytecode EVM (*Ethereum Virtual Machine*). C'est une architecture assez exotique, mais qui a le mérite d'être extrêmement simple à exécuter : il s'agit d'une machine à pile toute simple, sans registre.

1. Ou pas. Depuis le dernier *hard fork* d'Ethereum, il est possible de redéployer certains contrats via l'opcode CREATE2. Cela débloque des possibilités très intéressantes mais aussi des problématiques de sécurité non négligeables puisque le code d'un contrat n'est désormais plus nécessairement immuable sous certaines conditions.

Il y a environ 130 instructions (dont bon nombre sont sémantiquement très proches), pour pousser des constantes sur la pile, faire des sauts (conditionnels ou pas), des opérations arithmétiques, et interagir avec l'environnement (mémoire, *storage* persistant, appel à d'autres contrats).

Le listing 1 présente un exemple de bytecode EVM. Toutes les instructions font un octet, excepté le `PUSH` qui est suivi de la constante à empiler sur la pile. Ce code d'exemple accède à la taille des données passées en paramètre de l'appel (`CALLDATASIZE`), et la compare avec la constante `42`. S'il n'y a pas égalité, il saute à l'adresse 9, qui contient un `JUMPDEST` (marqueur de destination de saut conditionnel) puis exécute le `REVERT` qui lève une erreur et annule la transaction. Si les paramètres font `42` octets, alors le programme exécute `CALLER` qui empile l'adresse qui a appelé le contrat sur la pile, puis `SELFDESTRUCT`, qui cause l'autodestruction du programme et donne tout l'argent contenu dans le programme à l'adresse qu'il dépile (donc l'adresse appelante).

```
0 CALLDATASIZE
1 PUSH1 42
3 EQ
4 PUSH1 9
6 JUMPI
7 CALLER
8 SELFDESTRUCT
9 JUMPDEST
10 REVERT
```

Listing 1. Exemple de bytecode EVM.

L'exécution de code coûte de l'argent (du *gas*), pour s'assurer que toutes les exécutions se terminent. Cet aspect force les créateurs de contrats à les rendre les plus simples possible.

Pour en savoir plus, je vous recommande chaudement la lecture du *Yellow Paper* [5] qui décrit la machine virtuelle d'Ethereum dans les détails.

Ces propriétés les rendent très simples à analyser comparé à des exécutables classiques.

La plupart des contrats sont compilés depuis un langage de plus haut niveau appelé Solidity [14], qui ressemble un peu à du Javascript.

2.3 Déploiement et utilisation d'un contrat

Le listing 2 présente un exemple de contrat écrit dans le langage de haut niveau Solidity, qui est ensuite compilé en bytecode EVM.

Ce contrat définit deux fonctions : `deposit()` and `withdraw()` qui permettent respectivement de déposer et de retirer de l'argent. `withdraw()` prend un argument qui contient la somme que l'on souhaite retirer. Le mot-clé `public` indique que les fonctions sont appelables depuis l'extérieur. Le mot-clé `payable` autorise les appels de fonction à être assortis d'argent (par défaut, le code généré s'assure qu'on n'envoie pas d'argent, et lève une exception si c'est le cas).

```
contract Mapping {
    mapping(address => uint256) balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint256 to_withdraw) public {
        require(balances[msg.sender] >= to_withdraw);
        balances[msg.sender] -= to_withdraw;
        msg.sender.send(to_withdraw);
    }
}
```

Listing 2. Exemple de contrat Solidity.

La ligne `mapping(address => uint256) balances` définit un tableau associatif dans le *storage* persistant du contrat, qui permet de retenir la somme envoyée par chaque adresse. Si on lui envoie de l'argent, le contrat incrémente la variable `balances` de la somme reçue. Si on veut le retirer et qu'on dispose d'assez d'argent, il soustrait la somme retirée et nous envoie l'argent.

Les variables `msg.sender` et `msg.value` correspondent respectivement à l'adresse de l'émetteur de l'appel, et à la somme envoyée (en ethers).

Déploiement. Pour déployer ce contrat, on compile un code de déploiement, et on envoie une transaction sans spécifier de destinataire. Une adresse sera alors générée par le système, et le contrat sera déployé à cette adresse.

Utilisation. Sur Ethereum, une transaction effectue un appel (`CALL`) à une adresse destinataire. Un appel contient une somme qui est transférée vers le destinataire (`msg.value`), ainsi qu'un buffer de taille arbitraire qui contient des données (`msg.data`).

Une adresse peut être une adresse « normale », contrôlée par un humain. Un appel permettra alors de transférer de la valeur. Les données peuvent contenir un commentaire optionnel expliquant la raison de l'envoi.

Le récepteur peut aussi être l'adresse d'un contrat. Dans ce cas, le code du contrat est exécuté. Les données envoyées servent de « paramètres » pour le contrat. Une interface binaire (*ABI*) a été définie, et les quatre premiers octets de l'appel contiennent le *sélecteur de fonction* qui permet de savoir quelle fonction du contrat est appelée, puis suivent les éventuels paramètres de la fonction appelée.

Un contrat qui s'exécute peut lui-même faire d'autres appels : il peut donc envoyer de la valeur à quelqu'un d'autre, ou appeler un autre contrat. C'est le cas dans la fonction `withdraw()` de notre exemple : le contrat fait un appel à l'émetteur en lui transférant la somme qu'il souhaite retirer.

Un contrat peut aussi lever une exception pendant son exécution, et causer l'annulation de la transaction courante. Dans notre exemple, c'est ce qui se passerait si la condition du `require()` n'était pas satisfaite car l'émetteur n'aurait pas la somme qu'il souhaite retirer.

Exemple. Supposons que nous contrôlions l'adresse `0xbabababa...` et que le contrat de l'exemple est déployé à l'adresse `0xcafecafe...`. On peut alors envoyer la transaction suivante :

```
{"from": "0xbabababababababababababababababababababa",
 "to": "0xcafecafecafecafecafecafecafecafecafecafe",
 "value": "0x1",
 "data": "0xd0e30db0"}
```

Le sélecteur de fonction de `deposit()` étant bien `d0e30db0`, c'est cette méthode qui sera exécutée. Ici, on envoie 1 wei (le wei est l'unité de base d'Ethereum). Le contrat va donc écrire, dans son *storage* :

```
balances[0xbabababababababababababababababababababa] = 1
```

Un appel à `withdraw()` permet de retirer le wei envoyé :

```
{"from": "0xbabababababababababababababababababababa",
 "to": "0xcafecafecafecafecafecafecafecafecafecafe",
 "value": "0x1",
 "data": "0x2e1a7d4d
     0000000000000000000000000000000000000000000000000000000000000001"}
```

Ici, on envoie le sélecteur de fonction `2e1a7d4d` (`withdraw()`) suivi du paramètre de type `uint256`, qui contiendra 1. Le contrat va s'assurer que `balances[0xbabababa...]` contient au moins 1. C'est le cas, donc il va retrancher 1 (et notre solde va tomber à zéro), et nous envoyer 1 wei.

Si cette seconde transaction était envoyée par une autre adresse a, alors `balance[`a`]` vaudrait 0, donc il serait impossible de retirer de l'argent.

2.4 Sécurité des contrats intelligents

Au début du réseau Ethereum, ce sujet n'était pas vraiment central. Puis des bugs critiques ont mené à une prise de conscience générale du problème.

Bugs célèbres. Je tiens à citer quelques exemples de bugs qui ont mené à des pertes ou verrouillage de sommes très conséquentes :

Le bug de la DAO. Le premier programme déployé sur Ethereum qui a eu beaucoup de succès est la DAO [3] (pour *decentralized autonomous organization*). L'idée était de permettre à chacun de verser de l'argent dessus, et de récupérer des « parts » (proportionnelles aux sommes versées) dans cette organisation en échange. Il est ensuite possible de voter pour des « propositions », qui consistent à investir de l'argent contenu dans ce pot commun envers des projets précis (qui s'engagent à y reverser des dividendes s'ils ont du succès). Il s'agit d'une sorte de fond d'investissement géré collectivement, où l'argent est détenu par un programme.

La DAO a levé l'équivalent de 150 millions de dollars US, avant de se faire pirater par un acteur malveillant qui a exploité un nouveau type de vulnérabilité qui venait d'être découvert : une vulnérabilité de récursivité réentrante.

L'idée est que si on demande un retrait, la DAO :

1. vérifie qu'elle doit bien la somme demandée à l'adresse qui fait la demande,
2. envoie l'argent à l'adresse qui retire les fonds (via un appel, `CALL`, sans données mais avec l'argent),
3. puis soustrait la somme envoyée du compte de l'adresse.

Le retrait a lieu de manière atomique, et si la dernière étape échouait l'intégralité de la transaction serait annulée, donc il est impossible de retirer les fonds et de trouver un moyen de ne pas faire exécuter la soustraction.

Par contre, la nouvelle attaque consiste à faire en sorte que ce soit un programme spécialement conçu qui possède, puis retire des fonds. Il appelle la DAO pour retirer les fonds, et la DAO effectue un sous-appel pour lui envoyer l'argent retiré (étape 2). Ce sous-appel vers notre programme lui redonne la main, et il va rappeler une seconde fois la DAO pour effectuer le même retrait. La DAO est donc rappelée récursivement, alors qu'un appel plus haut sur la pile d'appel est toujours en cours !

Ce premier appel dans la DAO n'a pas encore soustrait les fonds, donc le sous-appel pour faire un retrait identique va s'effectuer avec succès.

C'est une fois que le premier appel reprend la main qu'il soustrait les fonds une seconde fois (sans revérifier qu'on avait bien assez d'argent, c'est censé être le cas après l'étape 1), et l'attaquant a pu soustraire deux fois la somme qu'il possède.

Pour résumer, l'idée est que certains appels peuvent potentiellement causer un sous-appel récursif vers le même contrat, et donc qu'il est possible qu'un appel externe modifie le *storage* persistant du contrat qui s'exécute, par effet de bord, alors même qu'il est déjà en cours d'exécution.

Le bug du portefeuille multi-signatures Parity. Parity, qui développe un client et un portefeuille Ethereum, a aussi implémenté un contrat intelligent qui peut servir de portefeuille multi-signatures : c'est un contrat qui a une liste de N clés publiques, et il sera possible de sortir les fonds du portefeuille seulement si K clés signent la transaction ($K \leq N$).

Pour diminuer les coûts de déploiement du contrat, ils ont déployé un premier contrat « bibliothèque » qui implémente toutes les fonctions du portefeuille. Les portefeuilles déployés ne contiennent quasiment pas de code, et se content d'appeler le code de cette bibliothèque partagée (son code est immuable, bien entendu) dans leur contexte.

En effet, il existe une instruction `DELEGATECALL` qui effectue l'appel au code d'un autre contrat tout en restant dans le contexte courant (avec le *storage* persistant du contrat appelant, il reste aussi l'émetteur d'éventuels sous-appels, etc...). Elle conçue pour permettre l'appel à du code dont on fait confiance, et le cas d'usage typique est justement le développement de bibliothèques partagées.

Toutefois, personne n'a pensé au fait que le code de cette bibliothèque était également appelable directement, dans son propre contexte. Or elle implémente une fonction d'autodestruction (pour détruire un portefeuille qui a fini de servir), que quelqu'un a appelée directement, et qui a causé sa destruction. Cela a été rendu possible car, appelée directement, la bibliothèque n'a rien dans son *storage* persistant, donc n'est pas initialisée et n'importe qui peut la détruire.

Une fois la bibliothèque détruite, tous les portefeuilles qui en dépendent deviennent des coquilles vides et il est impossible de les utiliser. Et donc de retirer l'argent qu'ils contiennent. C'est ainsi que l'équivalent de 150 millions de dollars (au moment des faits) ont été bloqués à vie dans divers contrats intelligents implémentant ce portefeuille multi-signatures [4].

Dans une *issue* GitHub (voir figure 1), le responsable de l'autodestruction de la bibliothèque plaide l'incompétence. Son « I accidentally killed it. » est devenu mythique dans les milieux de la sécurité sur Ethereum.

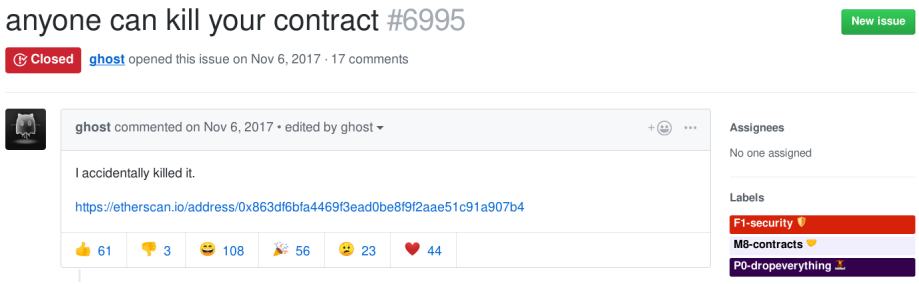

Fig. 1. « I accidentally killed it. »

Modèle d'exécution. Avant de finir, je tiens à citer/traduire l'extrait d'un article d'Adrian Colyer [10] :

> On a vu que les contrats intelligents ont de la valeur comme cibles. Ils ont aussi une combinaison de caractéristiques qui devraient faire froncer les sourcils de n'importe quel développeur expérimenté :
> — Ils s'exécutent sur un réseau décentralisé que n'importe quel participant peut rejoindre.
> — Les mineurs et/ou ceux qui appellent les contrats ont le contrôle de l'environnement dans lequel la transaction s'exécute (quelle transaction accepter, l'ordre des transactions, le *timestamp* du bloc, des manipulations de la pile d'appel...).
> — Tout ceci se passe dans un environnement qui punit toute personne qui ne fait pas un code parfait du premier coup. Il n'y a pas de mécanisme pour faire des patchs.

2.5 État de l'art

Avant d'attaquer le vif du sujet, je souhaite clarifier le fait que, contrairement aux attaques sus-citées qui ont nécessité beaucoup d'analyse manuelle et de créativité autour du fonctionnement de contrats précis, ce qui m'a intéressé était surtout de faire une analyse massive de tous les contrats qui ont été déployés pour y chercher des vulnérabilités pas forcément aussi compliquées.

Je tiens aussi à mentionner qu'il y a de nombreux outils similaires d'exécution symbolique. Je détaillerai les différences de Pakala par la suite.

De nombreux papiers scientifiques ont également été publiés sur le sujet. Il est intéressant de voir que dans un premier temps, des bilans assez catastrophistes ont vu le jour, se concentrant sur la somme d'argent contenue dans des programmes qui pourraient être vulnérables. On citera

en particulier *Finding The Greedy, Prodigal, and Suicidal Contracts at Scale* [6] qui mentionne que 6 millions de dollars auraient pu être volés car contenus dans des contrats vulnérables.

Plus récemment, d'autres articles se sont plus concentré sur l'activité passée, et ont montré au contraire que les outils d'analyse ont chacun des résultats très différents, et que dans majorité des cas les bugs vers lesquels ils pointent n'ont jamais pu être exploités en production. On citera ici *Smart Contract Vulnerabilities : Does Anyone Care ?* [7]

3 Pakala : exécution symbolique

L'outil que j'ai développé, Pakala [8] est un moteur d'exécution symbolique, conçu pour exécuter du bytecode EVM.

Il exécute du code de manière symbolique : les valeurs en mémoire peuvent être soit « concrètes », soit des « symboles » qui représentent des entrées utilisateurs et ne sont pas connues à l'avance. Il construit ensuite une liste des exécutions valides avec leurs préconditions, et effets de bords.

Une seconde passe va prendre tous les appels valides possibles, et les empiler pour essayer de modéliser une succession d'appels à différentes fonctions des contrats.

Cela permet de trouver des bugs plus complexes dans lesquels il faudrait d'abord appeler une première fonction pour modifier une variable stockée en dur, puis effectuer un autre appel pour déclencher un bug.

On essaie ensuite de voir si une de ces suites d'exécutions est « intéressante ». À l'heure actuelle, Pakala modélise seulement un unique test : est-ce possible de faire en sorte que le contrat nous envoie plus d'argent que ce qu'on lui envoie ? Cela revient à dire : **est-ce possible de lui voler de l'argent ?** C'est ce comportement que l'on appellera un « bug » dans la suite.

Nous expliquerons donc la manière dont Pakala est conçu, et nous ferons une démonstration de l'outil. Nous montrerons également des exemples de contrats vulnérables trouvés grâce à Pakala.

3.1 Exécution symbolique avec Z3

Pakala utilise Claripy [15] : c'est un wrapper de Z3 développé par l'équipe derrière Angr. Angr [16] est un outil d'exécution symbolique en Python, qui avait donc des besoins similaires.

Claripy peut faire de multiples analyses (comme de la *value-set analysis*), mais l'analyse qui nous intéresse ici, c'est l'utilisation de Z3 [11], qui

est un *SMT solver*, qui utilise une théorie mathématique pour nous dire si un ensemble de contraintes sur des inconnues est satisfiable (il existe une solution), et trouver des exemples de solutions.

Pakala contient en son cœur une implémentation de machine virtuelle, assez classique.

Si on rencontre un saut conditionnel, on rajoute simplement deux successeurs à l'état courant (pour les deux branches du saut), chaque successeur ayant une nouvelle contrainte correspondant à la condition pour que le saut se réalise dans cette branche.

La machine virtuelle d'Ethereum a toutefois une mémoire, et un *storage* persistant qui ont été les parties les plus difficiles à modéliser. Ce sont des problèmes bien connus en exécution symbolique [17].

La mémoire est gérée avec un dictionnaire Python, et supporte la lecture/écriture de symboles Claripy. Toutefois, il est impossible de lire/écrire des variables de taille inconnue. Or ceci peut arriver quand on copie une chaîne de caractères contrôlée par l'utilisateur dont la taille est a priori inconnue.

Pour traiter ces cas où la taille d'un objet n'est pas connue, on a recours à une stratégie plus proche du *fuzzing* : on va faire comme pour un saut conditionnel, et rajouter plusieurs successeurs à l'état courant. Chaque successeur aura une valeur possible pour la taille (parmi une liste de valeurs courantes). On force donc la « concrétisation » de la variable en essayant plusieurs solutions. Cela fonctionne bien, mais a le désavantage de provoquer une « explosion » du nombre de chemins à explorer, et peut augmenter de manière significative le temps d'analyse.

3.2 Exemple sur un contrat Solidity

Étudions l'exemple du contrat du listing 3 : il s'agit d'un contrat écrit avec Solidity, qui définit plusieurs méthodes qui peuvent être appelées indépendamment.

Ce contrat contient un tableau associatif `balances` qui suit la somme donnée par chaque participant, et l'incrémente quand il reçoit de l'argent via la méthode `invest()`.

Il permet aussi à son propriétaire (et seulement au propriétaire) de retirer l'intégralité des fonds avec `withdrawfunds()`.

La fonction `crowdfunding()` est censée être le constructeur du contrat, appelée seulement à son initialisation pour initialiser le propriétaire (`owner`) comme étant le créateur du contrat. Or ici elle n'a pas exactement le même nom que le contrat ; ce n'est donc pas un constructeur, et il peut être

appelée par n'importe qui. N'importe qui peut ainsi devenir propriétaire du contrat puis retirer tous les fonds. Ce problème de sécurité était très courant au début d'Ethereum.

```
contract Crowdfunding {
  mapping(address => uint) public balances;
  address public owner;

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function crowdfunding() public {
    owner = msg.sender;
  }

  function withdrawfunds() public onlyOwner {
    msg.sender.transfer(address(this).balance);
  }

  function invest() public payable {
    balances[msg.sender] += msg.value;
  }

  function getBalance() public view returns (uint) {
    return balances[msg.sender];
  }
}
```

Listing 3. Exemple de contrat vulnérable.

Si on analyse ce contrat, Pakala sort une liste d'exécutions valides. Parmi elles, l'exécution de la fonction `crowdfunding()` est décrite au listing 4. La structure produite contient un résumé de cette exécution :

- la liste des appels du contrat vers l'extérieur dans `calls` (aucun),
- les variables symboliques importantes définies pour cette exécution (c'est l'environnement `env`, qui définit la somme présente sur le contrat, l'adresse depuis laquelle on effectue l'appel, la somme envoyée lors de l'appel...),
- les emplacements du *storage* du contrat qui sont lus et écrits. Les clés lues le sont dans des symboles mentionnés dans le dictionnaire `storage_read`. Les clés sur lesquelles on écrit sont dans `storage_written` avec la valeur écrite.
- `solver` contient les contraintes symboliques qui doivent être satisfaites pour que cette exécution soit valide (pour rentrer dans les

bonnes branches des conditions ou des boucles). Les mentions à `calldata[]` sont des lectures des arguments passés au programme pour cet appel.

L'important ici est de voir que `storage_written` écrit à la clé `0x1` (correspondant à la variable `owner`) le nombre concret `0xcafe...` qui correspond à notre adresse appelante.

Étudions à présent la trace de la fonction `withdrawfunds()` (listing 5). On a un appel (dans `calls`) qui retire l'intégralité des fonds du contrat. Pour que cette exécution soit réalisée il faut toutefois satisfaire les contraintes (dans `constraints`). Une des contraintes est que `storage[0x1][159:0] == 0xcafe...` : la variable `owner` doit contenir notre adresse appelante.

Pakala va combiner ces deux exécutions, et se rendre compte que la seconde exécution devient possible si on la fait succéder à la première (listing 6).

Il nous montre ici un exemple de données concrètes avec lesquelles appeler le contrat pour déclencher le bug. En l'occurrence, le début des `calldata` doit contenir la *signature* des fonctions à appeler (qui identifie la fonction en question), et c'est tout, car ces fonctions n'ont pas d'argument. L'appelant doit être `0xcafe...` dans tous les cas, pour mettre la variable `owner` à la bonne valeur et l'utiliser.

3.3 Comparaison avec d'autres outils

Il existe de nombreux autres outils d'exécution symbolique pour Ethereum. Parmi les plus aboutis on pourra citer Mythril [12] et Manticore [13], qui utilisent également Z3.

La différence majeure est que ces outils cherchent à reproduire fidèlement la machine virtuelle d'Ethereum, et peuvent rentrer récursivement dans des sous-appels. Cela les rend plus puissants et génériques. Toutefois ils sont plus prompts à une explosion d'états possibles. De plus, leur sortie est plus difficile à interpréter.

Pakala se borne à lister toutes les exécutions possibles d'un contrat seul (sans interaction avec d'autres contrats). Il peut ainsi servir à comprendre ce que fait un contrat, puisqu'il suffit de regarder la liste des exécutions valides pour comprendre sa sémantique.

Je m'en suis déjà servi pour faire de l'ingénierie inverse de contrats complexes : en cas d'explosion d'états, je rajoute des contraintes pour limiter les états à explorer à ceux qui m'intéressent. J'obtiens ensuite une liste d'exécutions possibles, facile à comprendre.

```
{'calls': [],
 'env': {'balance': <BV256 0x128dfa6a90b28000>,
         'caller': <BV256 0xcafebabefffffffff0202fffffffff7cff7247c9>,
         'value': <BV256 value_38_256>},
 'selfdestruct_to': None,
 'solver': {'constraints': [<Bool 0x20 <= calldata_size_42_256>,
                            <Bool calldata[0]_45_256[255:224] == 0x56885cd8>,
                            <Bool value_38_256 == 0x0>],
            'hashes': {}},
 'storage_read': {<BV256 0x1>: <BV256 storage[<BV256 0x1>]_47_256>},
 'storage_written': {<BV256 0x1>: <BV256 0
     xcafebabefffffffff0202fffffffff7cff7247c9 | (0
     xffffffffffffffffffffffff0000000000000000000000000000000000000000 &
     storage[<BV256 0x1>]_47_256)>}}
```

Listing 4. Trace de `crowdfunding()` avec Pakala.

```
{'calls': [[<BV256 0x0>,
            <BV256 0x80>,
            <BV256 0x0>,
            <BV256 0x80>,
            <BV256 0x128dfa6a90b28000>,
            <BV256 0xcafebabefffffffff0202fffffffff7cff7247c9>,
            <BV256 0x0>]],
 'env': {'balance': <BV256 0x128dfa6a90b28000>,
         'caller': <BV256 0xcafebabefffffffff0202fffffffff7cff7247c9>,
         'value': <BV256 value_122_256>},
 'selfdestruct_to': None,
 'solver': {'constraints': [<Bool 0x20 <= calldata_size_126_256>,
                            <Bool calldata[0]_129_256[255:224] == 0x6c343ffe>,
                            <Bool value_122_256 == 0x0>,
                            <Bool storage[<BV256 0x1>]_131_256[159:0] == 0
                                xcafebabefffffffff0202fffffffff7cff7247c9>],
            'hashes': {}},
 'storage_read': {<BV256 0x1>: <BV256 storage[<BV256 0x1>]_131_256>},
 'storage_written': {}}
```

Listing 5. Trace de `withdrawfunds()` avec Pakala.

```
Transaction 1, example solution:
{'data': '56885cd8000000000000000000000000000000000000000000000000000000000000000',
 'from': '0xcafebabefffffffff0202fffffffff7cff7247c9',
 'value': 0}

Transaction 2, example solution:
{'data': '6c343ffe0000000000000000000000000000000000000000000000000000000000000000',
 'from': '0xcafebabefffffffff0202fffffffff7cff7247c9',
 'value': 0}

=======> Bug found! Need 2 transactions. <=======
```

Listing 6. Combinaison des exécutions de `crowdfunding()` et de `withdrawfunds()` avec Pakala.

Pakala est aussi le seul outil à empiler des exécutions successives de cette manière (sans ré-exécuter le code), ce qui le rend plus propice à explorer de longues chaînes de transactions.

Toutefois, comme Pakala analyse des contrats uniques et pas des interactions entre contrats, le bug de la DAO ou le bug du portefeuille multi-signature de Parity ne pourront pas être trouvés avec cet outil. Ce sont des bugs qui nécessitent des interactions entre plusieurs contrats et seront mieux détectés par des outils comme Mythril.

Comparaison avec Mythril. J'ai effectué une comparaison avec Mythril sur ma suite de tests, fin mars 2019. Mythril s'est globalement bien débrouillé, mais il semble qu'il ne supporte pas à l'heure actuelle d'enchaîner l'écriture de variables à un emplacement symbolique avec une lecture de cette même variable. C'est très pénalisant car de nombreux contrats permettent de transférer des *tokens* à une autre adresse de son choix.

4 Recherche de transactions suspectes

Pakala permet de trouver les programmes ayant l'air vulnérable, mais j'ai voulu aussi aborder le problème d'un autre point de vue : chercher, dans l'historique de la blockchain, des interactions menant à des vols évidents d'argent.

J'ai alors conçu un système pour rechercher de tels motifs dans l'historique des transactions, que j'ai fait tourner pour obtenir toutes les transactions suspectes. En particulier, j'ai cherché des séquences d'interactions du type :

1. création d'un contrat ;
2. diverses transactions à partir d'adresses d'un ensemble A ;
3. *le contrat contient de l'argent* ;
4. période d'inactivité ;
5. une ou plusieurs transactions à partir d'une adresse b n'appartenant pas à l'ensemble A ;
6. *le contrat ne contient plus d'argent.*

Cela correspond à un attaquant b semblant vider un contrat contenant de l'argent, alors qu'il ne se passe plus rien dessus.

4.1 En pratique

La recherche de ces interactions a nécessité de rejouer l'intégralité de l'historique de la blockchain : ce ne fut pas une mince affaire.

En effet, des explorateurs en ligne listent toutes les transactions effectuées, par contrat. Mais ils disposent de leur propre base de données qui n'est pas du tout dans le format natif. Et comme je voulais analyser toutes les transactions, sur tous les contrats, il aurait été inenvisageable d'utiliser l'API d'un explorateur en ligne.

Par défaut, un nœud Ethereum nous permet seulement de lister les transactions (par haché ou par numéro de bloc), et d'accéder à la structure avec l'état du système (pour un état récent).

J'ai donc créé un système qui rejoue toutes les transactions de tous les blocs, depuis le début de la chaîne, et indexe les données qui m'intéressent pour chercher des contrats qui correspondent à mes critères. Il m'a fallu plus d'un mois pour faire l'analyse.

De plus, j'ai eu besoin d'accéder au montant disponible sur les contrats, non pas immédiatement, mais il y a longtemps. Ceci nécessite de consulter l'état du système pour n'importe quel bloc. Or, par défaut, les nœuds Ethereum tournent seulement avec l'état courant qu'ils mettent à jour en continu : pas besoin de garder l'historique de cette structure. J'ai donc dû utiliser le mode « archive » de mon nœud, qui garde l'historique de la structure d'état à chaque bloc : la synchronisation est bien plus lente, ne peut se faire que sur SSD, et le nœud utilise 2 To d'espace disque, une fois synchronisé.

5 Résultats

Cette section présente deux types de résultats :
— ceux obtenus via l'analyse du code avec Pakala ;
— ceux obtenus par l'analyse de l'historique de la blockchain.

5.1 Pakala

J'ai fait tourner Pakala sur l'intégralité des contrats intelligents déployés par le passé sur Ethereum, en notant les contrats pour lesquels il aurait été possible de voler de l'argent.

Cela a mis en évidence une centaine de contrats vulnérables par le passé, et la plupart de ces contrats ont été vidés depuis. Les sommes en jeu n'étaient jamais supérieures à quelques centaines d'euros.

J'ai aussi vu quelques contrats que je pense être toujours vulnérables après une analyse manuelle. Ils contiennent en général quelques centimes d'euros, ce qui explique le fait qu'ils n'aient pas été exploités, je pense.

J'ai aussi fait tourner Mythril sur les mêmes contrats que Pakala. Il a été intéressant de constater qu'il est extrêmement rare qu'ils trouvent tous les deux des bugs sur les mêmes contrats (à creuser). Mythril trouve deux fois plus de bugs, mais la majorité de ceux qu'il trouve sont en fait des faux positifs.

5.2 Analyse de l'historique des contrats

Cette autre méthode m'a également rapporté des dizaines d'attaques, assez différentes de mes résultats avec Pakala : seule une dizaine de contrats a été trouvée par les deux méthodes. Cela prouve qu'elles sont complémentaires mais je compte creuser pour comprendre les raisons de cette faible intersection.

On y retrouve toutefois quelques adresses d'attaquants, et de contrats vulnérables qui ont été exploités en commun.

À la base, mon système a généré un fichier avec des milliers d'occurrences. J'ai dû ensuite filtrer manuellement pour enlever les faux-positifs et garder seulement les contrats intéressants.

En effet, la plupart des occurrences étaient des portefeuilles multisignatures ou des levées de fonds, dans lesquelles une adresse était explicitement mise sur liste blanche, mais n'avait jamais interagi avec le contrat. Ainsi, quand elle envoie une transaction pour la première fois et vide le contrat, tous les voyants sont au rouge !

Le fichier brut avec toutes les attaques que j'ai trouvées est disponible sur mon blog [9].

5.3 Types de contrats trouvés

Cette section décrit des contrats classiques trouvés en partie avec Pakala, et en partie avec ma méthode d'analyse de l'historique des contrats.

Honeypots J'ai vu de nombreux contrats faits pour « scammer » d'éventuels attaquants, en leur faisant croire à une vulnérabilité (nécessitant l'envoi d'une petite somme), là où une subtilité empêche l'envoi de l'argent qu'ils pensent gagner.

Ces contrats ont été souvent remontés par Pakala, car certains d'entre eux utilisent des subtilités de l'EVM qui sont incorrectement modélisées par ce dernier.

Le contract `Giveaway` (listing 7) est un bon exemple de honeypot : la variable `SecretNumber` est publiquement accessible, comme l'est l'état de tous les programmes. On a l'impression qu'en envoyant plus de 0.07 ethers, on va récupérer l'argent contenu dans le contrat (`this.balance`), ainsi que l'argent qu'on a envoyé (`msg.value`). Sauf que `this.balance` contient déjà l'argent qu'on a envoyé, donc le contrat va tenter d'envoyer plus d'argent que ce qu'il contient, et l'envoi va rater. On ne récupérera donc rien, et on aura perdu l'argent envoyé.

```
contract Giveaway {

    address private owner = msg.sender;
    uint public SecretNumber = 24;

    function() public payable {
    }

    function Guess(uint n) public payable {
        if(msg.value >= this.balance && n == SecretNumber && msg.
            value >= 0.07 ether) {
            // Previous Guesses makes the number easier to guess so
                you have to pay more
            msg.sender.transfer(this.balance + msg.value);
        }
    }

    function kill() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }
}
```

Listing 7. `Giveaway`: contrat honeypot.

`enigma_x`, présenté au listing 8, est le honeypot le plus complexe que j'ai vu. On est censé envoyer 1 Ether (~100€), ainsi que la réponse à une énigme. Le programme compare le haché cryptographique de notre réponse avec un haché stocké. Sauf qu'on peut voir dans l'historique des transactions le moment où la réponse est initialisée, avec sa valeur en clair (le contrat calcule puis stocke son haché).

On pense donc avoir la réponse, et on appelle `Play()` en envoyant notre argent, ce qui ne fonctionnera pas car la transaction qui initialise le haché à partir de la réponse est en fait une « no-op », car le haché est déjà initialisé secrètement au moment où le contrat est créé.

```
contract enigma_x
{
    function Play(string _response) external payable
    {
        require(msg.sender == tx.origin);
        if(responseHash == keccak256(_response) && msg.value>1 ether
            )
        {
            msg.sender.transfer(this.balance);
        }
    }

    string public question;
    address questionSender;
    bytes32 responseHash;

    function StartGame(string _question,string _response) public
        payable
    {
        if(responseHash==0x0)
        {
            responseHash = keccak256(_response);
            question = _question;
            questionSender = msg.sender;
        }
    }

    function StopGame() public payable
    {
        require(msg.sender==questionSender);
        msg.sender.transfer(this.balance);
    }

    function NewQuestion(string _question, bytes32 _responseHash)
        public payable
    {
        require(msg.sender==questionSender);
        question = _question;
        responseHash = _responseHash;
    }
}
```

Listing 8. `enigma_x`: contrat honeypot.

J'ai vu des dizaines d'instance de ce honeypot qui arrivent souvent à piéger au moins une personne, et je pense que son créateur a dû gagner au minimum quelques milliers d'euros.

Erreur de nommage du constructeur Comme expliqué plus haut dans mon exemple sur Pakala, le contrat `HOTTO` du listing 9 montre une erreur très classique : le code du « constructeur » censé être appelé à l'initialisation du contrat devient utilisable par n'importe qui. Ici, n'importe qui peut l'appeler pour devenir « propriétaire » du contrat, puis ensuite retirer tous les fonds.

```
contract HOTTO is ERC20 {
    address owner = msg.sender;

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    function HT () public {
        owner = msg.sender;
        distr(owner, totalDistributed);
    }

    function withdraw() onlyOwner public {
        address myAddress = this;
        uint256 etherBalance = myAddress.balance;
        owner.transfer(etherBalance);
    }
}
```

Listing 9. `HOTTO`: erreur de nommage du constructeur.

Pour ce contrat `HOTTO` [2], on peut voir une adresse (`0xacc...`) appeler le constructeur, devenir propriétaire et retirer tous les fonds, à quelques reprises (détournement de ~100€ au total).

J'ai vu des dizaines de contrats ainsi, remontés à la fois par Pakala et par mon analyse de l'historique des transactions.

Overflows / entiers non signés Toutes les opérations arithmétiques de l'EVM *wrappent* en cas d'overflow. Le compilateur Solidity ne fait, par défaut, aucune vérification là-dessus et ne lèvera pas d'exceptions.

2. L'adresse de ce contrat est `0x612f1BDbe93523b7f5036EfA87493B76341726E3`.

On peut donc avoir toutes sortes de problèmes liés à des overflows, en particulier quand on soustrait des entiers non signés.

Un contrat qui montre ce genre de problème est `0xeBE6c7a839A660a0F04BdF6816e2eA182F5d542C`, pour lequel je n'ai pas de code source (mais une version décompilée est générable via le site `eveem.com`) L'idée ici est que la fonction `transfer()` du contrat permet de récupérer un montant `value`, à condition d'envoyer une somme supérieure à `value`. En effet, elle vérifie que `call.value - value >= 0`. Toutefois cette condition est toujours vraie, puisque les deux nombres sont des entiers non signés, donc le résultat est également non signé. On peut donc passer `value > call.value` sans souci, et vider le contrat.

Pakala m'a remonté quelques contrats vulnérables à cause de ce type d'erreur.

Autres problèmes Le contrat `lottery` du listing 10 est un bon exemple qui synthétise d'autres erreurs, plus rares :

- Les tickets coûtent « 1/10 », sans spécifier l'unité. Par défaut c'est l'unité indivisible la plus petite : le wei, qui est un entier. Le ticket coûte donc « 1/10 » wei, c'est à dire 0 wei : il est gratuit.
- L'absence d'accolages après les if(). Seule la première instruction fait partie des conditions (vu plusieurs fois...)
- L'utilisation de `now` (date où le bloc courant est miné) comme source d'entropie. On peut très facilement créer un contrat qui appellerait la loterie et achèterait le dernier ticket seulement si on est sûr que le dernier ticket gagnerait le jackpot.

5.4 Attaques trouvées

J'ai vu des dizaines et des dizaines de contrats contenant des bugs. Aucun ne contenait toutefois encore de l'argent, ou seulement des sommes vraiment négligeables.

Par contre, dans de nombreux cas, j'ai vu certaines adresses revenir souvent, et ces dernières semblent avoir exploité ces bugs et vidé les contrats qui contenaient des sommes non négligeables. Je ne suis donc pas le premier à avoir pensé à effectuer ce genre de scan offensif.

Ma recherche dans l'historique des transactions m'a permis de mettre en évidence quelques adresses intéressantes :

- `0x80028f80C7d5959C3Eaf45A95bf3a1A0724352f6`
- `0x90a0c432DA9d200f496FCDa91F4A3D42Ea3A6089`

```
contract lottery{

    //Wallets in the lottery
    //A wallet is added when 0.1E is deposited
    address[] public tickets;

    //create a lottery
    function lottery(){
    }

    //Add wallet to tickets if amount matches
    function buyTicket(){
        //check if received amount is 0.1E
        if (msg.value != 1/10)
            throw;

        if (msg.value == 1/10)
            tickets.push(msg.sender);
            address(0x88a1e54971b31974b2be4d9c67546abbd0a3aa8e).send
                (msg.value/40);

        if (tickets.length >= 5)
            runLottery();
    }

    //find a winner when 5 tickets have been purchased
    function runLottery() internal {
        tickets[addmod(now, 0, 5)].send((1/1000)*95);
        runJackpot();
    }

    //decide if and to whom the jackpot is released
    function runJackpot() internal {
        if(addmod(now, 0, 150) == 0)
            tickets[addmod(now, 0, 5)].send(this.balance);
        delete tickets;
    }
}
```

Listing 10. `lottery`: un contrat, plusieurs bugs.

Dans les deux cas, on les voit interagir avec de nombreux contrats, avec clairement pour objectif de les vider. La première est à l'origine de plus de 900 transactions ! La seconde est responsable de beaucoup moins de transactions, mais semble avoir gagné plus : si elle a uniquement exploité des bugs, elle a gagné plus de 5000€ ainsi.

6 Conclusion

On a vu que les contrats intelligents sur Ethereum sont constamment scannés, et qu'un contrat vulnérable contenant de l'argent sera rapidement siphonné.

Toutefois, ce genre d'attaques concerne majoritairement des contrats avec des bugs simples : on a vu, dans l'histoire de réseau, des bugs beaucoup plus subtils se faire exploiter manuellement pour des gains bien plus importants.

Le développement de contrats intelligents se fait actuellement sans processus rigoureux : au mieux, ils sont audités par des entreprises spécialisées après leur développement. Or, certains contiennent des millions d'euros, ou plus. On peut faire un parallèle avec la manière dont les systèmes critiques sont développés en avionique, où le processus est bien plus rigoureux, où des systèmes de preuves formelles sont utilisés, etc.

Une remarque qui revient très souvent est que le langage Solidity est fait pour être très simple à aborder par un développeur qui fait un peu de Javascript. Il est donc relativement facile de développer des contrats intelligents.

Or, pour développer correctement des contrats intelligents, il faut de solides bases en théorie des jeux, en cryptographie, savoir que toutes les transactions sont publiques, que les mineurs qui génèrent les blocs peuvent générer des transactions qui passeront avant toutes les autres, qu'il n'y a pas vraiment de bonne source d'entropie à l'heure actuelle, etc.

Il y a énormément de problèmes à résoudre dans cet écosystème, qui est encore extrêmement jeune.

Références

1. Andreas M. Antonopoulos, *Mastering Bitcoin*
`https://github.com/bitcoinbook/bitcoinbook`
2. Andreas M. Antonopoulos, *Mastering Ethereum*
`https://github.com/ethereumbook/ethereumbook`
3. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`

4. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/
5. Ethereum Yellow Paper : https://ethereum.github.io/yellowpaper/paper.pdf
6. Nikolic et al. *Finding The Greedy, Prodigal, and Suicidal Contracts at Scale* https://arxiv.org/abs/1802.06038
7. Perez et al. *Smart Contract Vulnerabilities : Does Anyone Care ?* https://arxiv.org/abs/1902.06710
8. https://github.com/palkeo/pakala
9. https://www.palkeo.com/fr/projets/ethereum/stolen_ether.html
10. https://blog.acolyer.org/2017/02/23/making-smart-contracts-smarter/
11. https://github.com/Z3Prover/z3
12. https://github.com/ConsenSys/mythril-classic
13. https://github.com/trailofbits/manticore
14. https://solidity.readthedocs.io
15. https://github.com/angr/claripy
16. https://angr.io
17. Baldoni et al. *A Survey of Symbolic Execution Techniques.* https://dl.acm.org/citation.cfm?id=3182657 Section 3 : *Memory Model.*
18. https://github.com/paritytech/parity-ethereum/issues/6995