Comparatif d'outils d'analyse des contrats intelligents pour le langage Solidity d'Ethereum

Mourad Djellouli
https://github.com/Mourad7/SolidityCode

Projet de session INF889A - Analyse de programmes pour la sécurité logicielle –

Jean Privat

Table des matières

Introduction :

Les contrats intelligents sont des bouts de codes présents dans une blockchain, et exécutés par une machine virtuelle (EVM dans le cas d'Ethereum), l'intérêt étant qu'il n'est pas nécessaire d'avoir d'autorité centralisée (financières ou non) ou d'intermédiaire. Les clauses d'un vrai contrat y sont codées, ainsi, une fois rempli ce contrat est exécuté de façon automatique.

Problème :

Étant donné que les contrats sont présents sur le grand registre de la Blockchain, le code doit être sain et sans bugs ni failles qui pourraient être potentiellement exploités par des hackers. Cela représente un challenge pour les développeurs qui devront s'assurer de la qualité du code avant de le déployer.

PS : Pour ne pas alourdir ce document, les descriptions détaillées des failles étudiées, ainsi que le fonctionnement des outils utilisés dans le cadre du Benchmark sont présentes sur le GitHub du projet, mais aussi dans ***l'annexe 1 : Descriptif des failles et outils.***

Collecte des instances

Une des premières difficultés rencontrées pour cette étape, est le fait d'installer les différents outils, sachant que certains sont disponibles sur Windows d'autres non, peuvent dépendre de npm ou Pypi. Et pour pouvoir comparer les outils « équitablement » il est important de les faire tourner sous les mêmes conditions (configuration) partout, j'ai opté pour la solution de conteneurs Docker, qui n'est malheureusement pas disponible pour tous les outils, et à cause de cette contrainte supplémentaire la comparaison du temps d'exécution n'a plus réellement de sens.

- SmartCheck / Securify : Version Web (contenant plus d'analyseurs)
- Slither / Manticore / Echidna : conteneur Docker
- MythX : Extension disponible sur VS Code

Après avoir choisi les outils et les présenter, il a fallu identifier les différentes failles à étudier, dont les descriptions sont présentes sur le GitHub du projet, mais également récolter des contrats intelligents contenant des instances de ces failles, et cela depuis plusieurs sources :

- Smart Contract Weakness Classification (SWC): https://swcregistry.io/
- Not-so Smart Contracts: https://github.com/crytic/not-so-smart-contracts

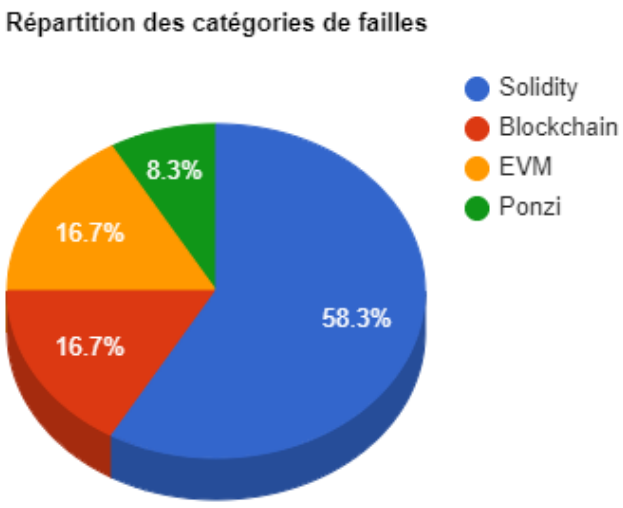
Ensuite j'ai procédé à la vérification du code de chacun des contrats pour mieux comprendre ce qu'ils faisaient, il fallait également définir une version de compilateur fixe pour tous les contrats dans le cadre du Benchmark.

Pour la prochaine étape il a fallu triller les différentes failles et les regrouper en catégories selon leurs origines.

Systèmes de Ponzi : Il a été observé dans le passé que certains contrats intelligents d'Ethereum ont étaient utilisés pour créer des systèmes de Ponzi, pour arnaquer des utilisateurs innocents en leur promettant des gains énormes sur leurs investissements. Même si en soi un système Ponzi ne représente pas une vulnérabilité de sécurité directe, j'ai jugé pertinent de les rajouter à cause de la menace que cela peut représenter pour l'argent des utilisateurs, mais également sur la crédibilité de la technologie Blockchain en général.

On se retrouve ainsi avec 4 catégories principales de failles réparties comme suit :

A. **Type S :** Failles dues à une mauvaise utilisation du le langage Solidity.
B. **Type EVM :** Failles engendrées durant la compilation par a machine EVM.
C. **Type BC :** Failles présentes dans l'architecture de la Blockchain Ethereum en général
D. **Type PS :** Systèmes de Ponzi (Ponzi Schemes)

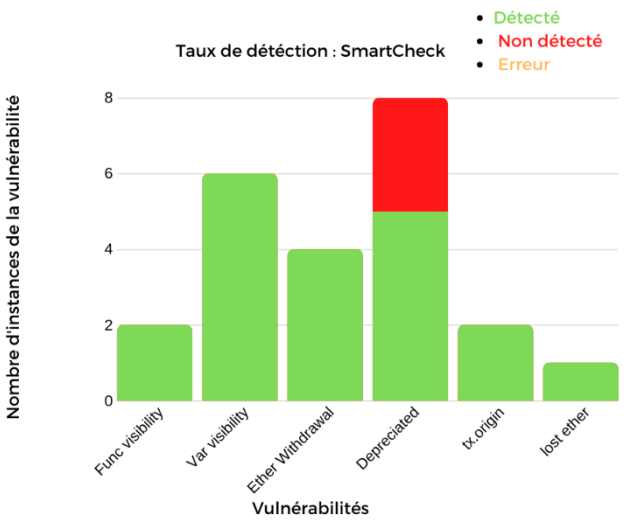


Résultats et analyses

I. Résultats individuels

1. SmartCheck

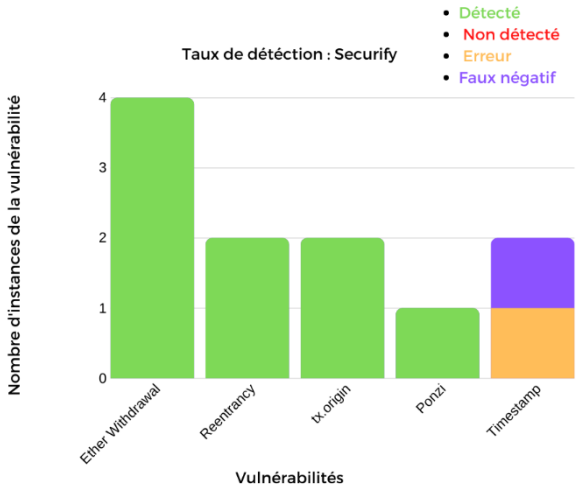
Les performances de SmartCheck sont assez bonnes, arrivant à détecter la majorité des instances des failles qu'il peut analyser, seulement on remarque qu'il n'arrive pas à nous flaguer toutes instances des fonctionnalités dépréciées, ou encore parmi les outils étudiés c'est le seul qui ne détecte pas la réentrance, sachant que c'est un problème très commun dans les contrats intelligents.



2. Securify

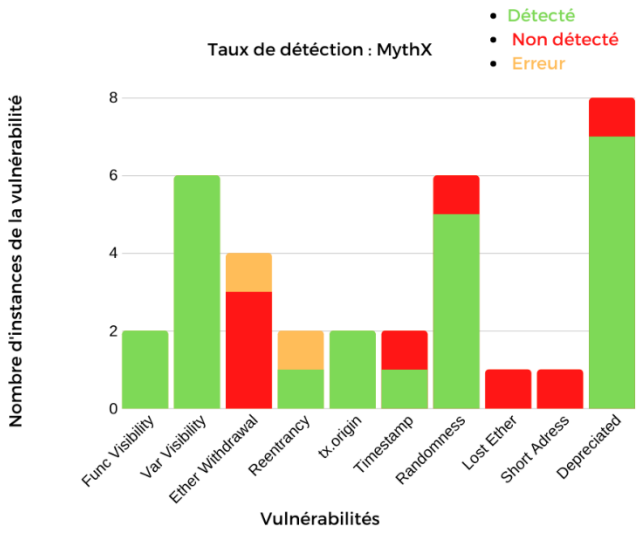
Securify s'en sort moins bien que les autres outils, ne couvrant pas le même nombre de failles, seulement il se distingue des autres en arrivant à détecter les deux instances de la réentrance. Mais également la faille présente dans le schéma de Ponzi, Rubixi.

Securify est le seul outil qui signale un contrat comme étant « sain » d'une vulnérabilité particulière, mais s'il se trompe en déclarant un contrat « sain », alors que ce n'est pas le cas, on considère cela comme étant un faux négatif.



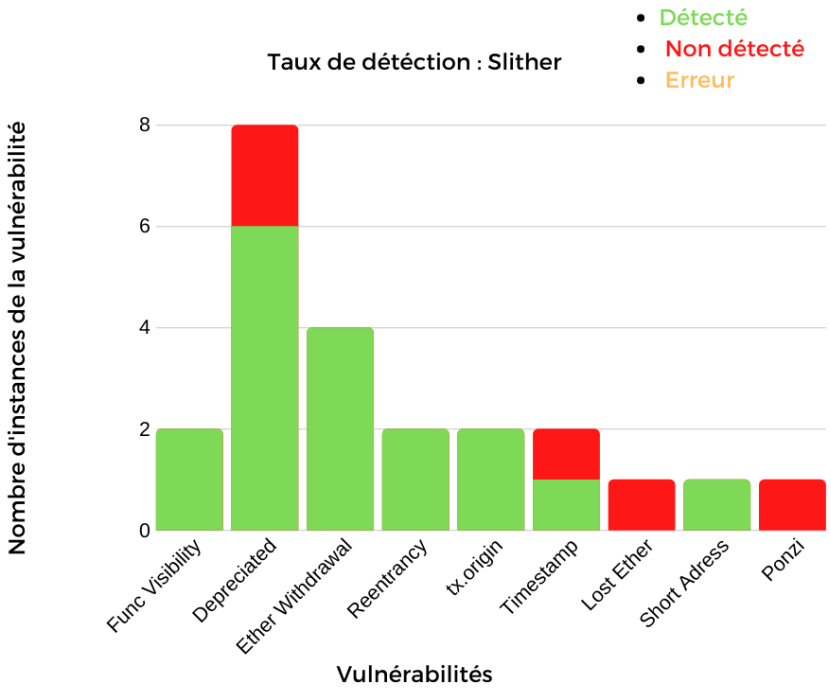
3. MythX

MythX est l'outil qui s'en sort le mieux, arrivant à détecter 7 vulnérabilités, dont la majorité des instances des faibles sources de génération aléatoires. Et cela est dû au fait qu'il aligne trois types d'analyses : une statique, une dynamique avec Mythril et du fuzzing. Seulement sur les 4 contrats implémentant des retraits d'Ether non protégés, il n'a pas réussi à en compiler un et n'a rien trouver sur les 3 autres.



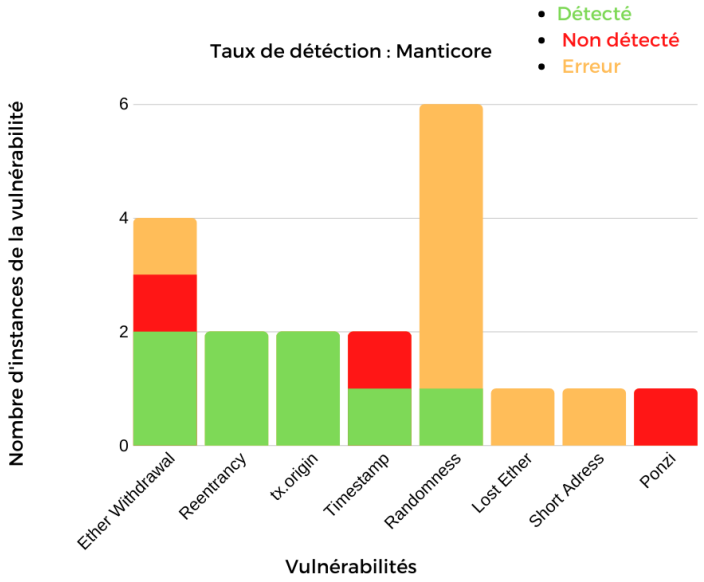
4. *Slither*

Slither nous donne des résultats aussi satisfaisants que MythX et cela sur la majorité des catégories des vulnérabilités, et cela sans aucune erreur. Il est à noter cependant, que comparé aux autres outils, il ne fonctionne pas avec les contrats écrits avec une version Solidity antérieure à 0.4, ce qui peut rendre l'audit de contrats déjà déployés difficile, voire impossible, et c'est la principale raison pour laquelle la version choisie pour ce Benchmark est 0.4.24.



5. *Manticore*

Manticore est l'outil qui s'en sort le moins bien si on prend en compte le ratio succès / erreurs, résultant en plusieurs timeout et ne trouvant souvent pas de chemins d'exécutions résultants à des vulnérabilités ainsi que des erreurs de compilations, à noter cependant qu'en dehors de MythX c'est le seul qui arrive à trouver les deux vulnérabilités de type BC (Blockchain) qui sont la dépendance de l'horodatage et les faibles sources de génération aléatoire, et il a de bonnes performances pour les failles de réentrances.

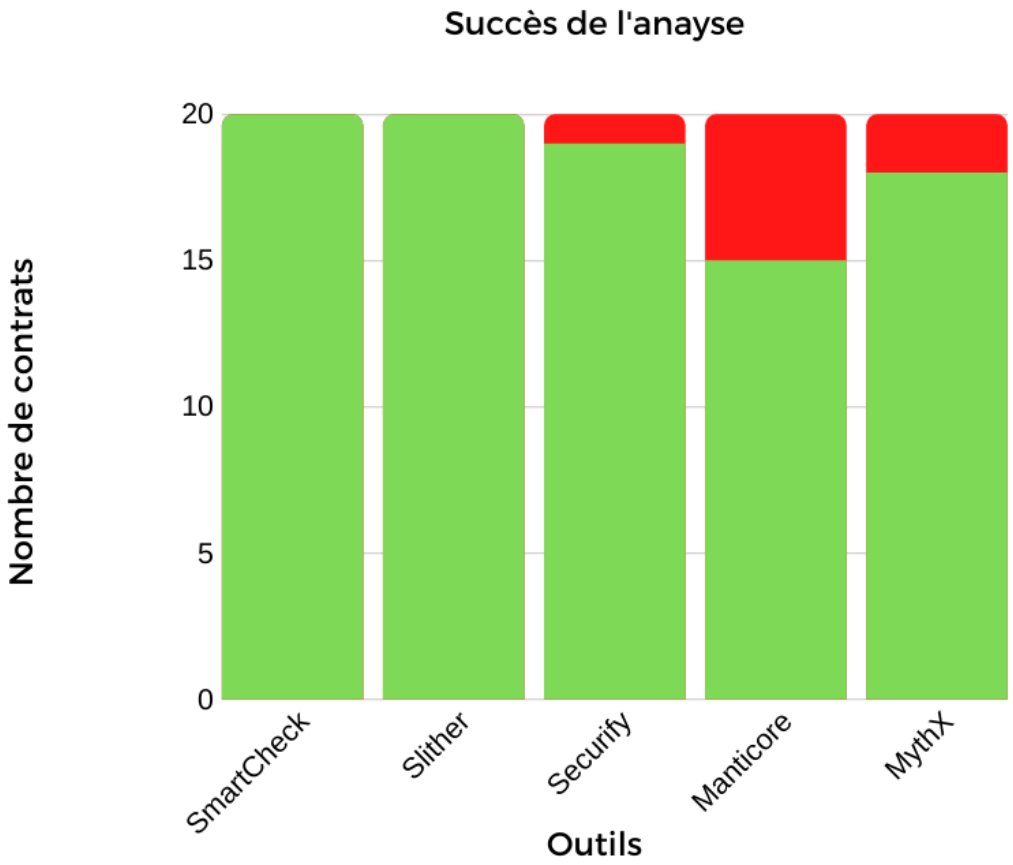


II. Temps d'exécution et couverture

Slither et SmartCheck sont les deux outils qui s'en sortent le mieux en termes de temps d'exécution, tant dit que Manticore qui est purement basé sur de l'exécution symbolique, est celui qui prend le plus de temps (Plus de 48 minutes et 300 chemins d'exécutions générés pour Rubixi et au final ne rien trouver), et cela confirme l'idée générale que les analyseurs purement statiques sont plus rapides.

Un autre constat, est que ces mêmes outils d'analyse statique sont les seuls qui arrivent à terminer l'analyse des contrats sans erreurs de compilation :

- MythX :
    1. Ether2.sol
    2. Reentrancy1.sol
- Manticore :
    1. Ether1.sol
    2. Random1.sol
    3. Random2.sol
    4. LostEther.sol
    5. ShortAdress.sol
- Securify :
    1. Timestamp1.sol



III. Résultats finaux

Le tableau suivant montre la matrice de vulnérabilité de tous les outils utilisés dans ce comparatif, la colonne Max démontre le nombre d'instances de chacune des failles, cela peut paraitre déséquilibré à certains endroits. Seulement il faut savoir qu'une faille peut prendre plusieurs formes alors qu'une autre pas forcément. Prenons l'exemple l'utilisation des fonctions dépréciées qui selon la documentation officielle, est du nombre de 8 actuellement. Alors qu'au contraire l'autorisation via tx.origin ne peut prendre qu'une seule forme, L'objectif via ce tableau est de démontrer la versatilité et l'adaptation des outils par apport aux failles, prenons le cas de la génération aléatoire qui a pu être implémentée de 6 façons via 4 contrats différents ; seuls MythX et Manticore arrivent à détecter cette faille, seulement Manticore a pu en identifier qu'une seule occurrence, tant dis que MythX en a trouvé cinq sur six possibles.

On remarque que toutes les failles sont au moins couvertes par un outil, l'autorisation via tx.origin est détectée par tous. Seulement ce qu'on recherche également est la capacité de détecter des failles provenant de plusieurs catégories, malheureusement aucun n'arrive à le faire pour toutes, celui qui s'en rapproche le plus est Slither qui arrive à détecter des failles sur 3 catégories (Solidity, Blockchain, EVM), alors que tous les autres ne le font que sur 2 au mieux.

Les cellules surlignées en vert indiquent que toutes les instances de cette vulnérabilité présentes dans le Benchmark sont détectées avec succès, tant dit que le gris met en évidence les instances de vulnérabilités maximales (mais pas toutes) détectées par les outils.

		Manticore	Slither	SmartCheck	MythX	Securify	Max
Solidity	Function visibility		2	2	2		**2**
	Var visibility			6	6		**6**
	Retrait d'Ether	2	4	4		4	**4**
	Réentrance	2	2		1	2	**2**
	Déprécié		6	5	7		**8**
	Tx.origin	2	2	2	2	2	**2**
B	Timestamp	1	1		1		**2**
	Randomness	1			5		**6**
EVM	Lost Ether			1			**1**
	Short Address		1				**1**
PS	Ponzi (Rubixi)					1	**1**
Total		**8**	**18**	**20**	**24**	**9**	**35**

Pour chacun des contrats, la version du compilateur a été délibérément laissée flottante, Si on considère qu'un outil trouve au moins une instance d'une faille donnée, veut dire qu'il est capable de détecter cette faille en général, et en rajoutant faille du pragma flottant, on se retrouve avec le tableau suivant :

		Manticore	Slither	SmartCheck	MythX	Securify
Solidity	Function visibility		Y	Y	Y	
	Var visibility			Y	Y	
	Retrait d'Ether	Y	Y	Y		Y
	Réentrance	Y	Y		Y	Y
	Fonctions dépréciées		Y	Y	Y	
	Pragma Flottant		Y	Y	Y	
	Autorisation Tx.origin	Y	Y	Y	Y	Y
B	Timestamp	Y	Y		Y	
	Randomness	Y			Y	
EVM	Lost Ether			Y		
	Short Address		Y			
PS	Ponzi (Rubixi)					Y
Total (12 failles)		**5**	**8**	**7**	**8**	**4**

Après avoir mis à jour notre tableau, on se retrouve finalement avec deux outils qui arrivent à détecter 8 failles (sur 12 possibles), qui représente en soit un excellent résultat, notamment pour Slither qui fait « que de l'analyse statique » par apport a MythX qui est bien plus complet, et si on rajoute à cela le fait qu'il produit ses résultats plus vite que les autres et sans aucune erreur de compilation, en plus d'être open source, cela le rend sans doute le meilleur outil à utiliser actuellement pour toute personne souhaitant faire du développement sur la plateforme Ethereum avec le langage Solidity. Cependant ce que l'on ne peut pas retirer à MythX est ses Dashboard intuitifs et le fait de pouvoir sauvegarder ces résultats pour les consulter plus tard ou encore les partager facilement (PDF, JSON), cela s'inscrit mieux dans le cycle de développement d'application, notamment quand le projet inclut plusieurs personnes.

Conclusion

On se rend compte malheureusement qu'il n'existe pas encore d'outil qui fait l'unanimité, seulement on voit que pour une technologie qui existe que depuis 2014, on se trouve déjà avec un large éventail d'outils proposant des approches assez différentes. Et comme discuté durant la présentation cette sécurité nécessaire représente un enjeu primordial pour le développement sur Ethereum, sachant que les contrats intelligents sont immuables, ce qui met une pression supplémentaire sur les développeurs qui vont devoir redoubler d'efforts pour s'assurer de la qualité de leurs codes avant de pouvoir les déployer et ainsi éviter des scandales du type DAO, et le meilleur moyen de le faire est de rajouter à leurs projets, un outil d'analyse ou Linter assez performant qui serait capable de mettre en évidence les bugs dans le code, aussi mineurs et basiques soit ils, et l'utiliser tout au long du développement.