Démystification du FlashLoan : Cas d'usage lors d'un arbitrage sur MainNet
step-by-step guide

TORUS
Twitter : @0x8479828583
GitHub : https ://github.com/0x8479828583

21 août 2022

Résumé

Le mécanisme du **F**lash **L**oan (**FL**) est relativement méconnu du public sensible à l'univers des cryptomonnaies et plus particulièrement à celui de la **De**centralized **Fi**nance (**DEFI**). En effet, il suscite beaucoup de fantasmes et est associé à des actions uniquement malveillantes. Aussi, sa médiatisation est quasi intégralement liée à son utilisation lors de hacks à plusieurs millions ou centaines de millions de dollars [1] [2][3][4]. Dans cet article, nous vous proposons un cas d'utilisation concret du **FL** step-by-step qui vous permettra de comprendre concrètement son fonctionnement. L'avantage de ce papier est de vous proposer un cas d'usage, concret sur un MainNet au lieu d'une simple démonstration en local ou sur un testnet. Ce papier donne également accès dans son intégralité au code solidity utilisé sur GitHub prêt à être déployé.

1 FlashLoan

1.1 Définition

Un FlashLoan est un emprunt instantané. En effet, il vous permet d'emprunter une somme d'argent (théoriquement infinie) sans nécessiter une garantie (collatéral). Cela est possible si et seulement si vous réaliser l'emprunt et son remboursement dans une et unique transaction.

Un utilisateur lambda, avec son wallet, ne peut effectuer qu'une opération par transaction. Aussi, il n'est pas possible pour un utilisateur de réaliser un **FL**. En effet, il ne peut emprunter et rembourser en une seule transaction. Pour un utilisateur munit d'un wallet cela ne peut être réalisé qu'en un minimum de deux transactions : une pour emprunter et une autre pour rembourser. C'est pour cela que les **FL** sont déployés à l'aide de smartcontracts qui eux peuvent réaliser de multiples opérations dans une même transaction.

1.2 Implémentation technique

Le **FL** est standardisé et est défini par la norme EIP-3156[5]. Il est possible d'utiliser des libraires comme openzeppelin [6] pour faciliter l'exploitation des **FL**. En effet cette librairie intègre le standard ce qui facilite son exploitation.

Il y a d'un côté l'emprunteur (**Borrower**) et de l'autre le prêteur (**Lender**). Le prêteur peut permettre l'emprunt de un ou plusieurs tokens. Par exemple, AAVE[7] permet l'emprunt de multiples tokens ERC-20 avec 0.09% d'intérêt sur le montant emprunté. Dans notre cas d'usage, décrit à la section 3 nous ne pourrons emprunter qu'un seul et unique token à partir du prêteur. L'emprunteur et le prêteur sont des smartcontracts et doivent respectivement hériter des propriétés de **IERC3156FlashBorrower** et **IERC3156FlashLender** comme le spécifie la norme EIP-3156.

2 Decentralized Exchange (DEX)

2.1 Historique

Les **DEX** sont des **D**ecentralized **App**lication (**DAPP**) qui permettent les échanges de tokens qui répondent à la norme ERC-20 [1]. Le premier **DEX** véritablement popularisé et qui d'ailleurs porté une bonne partie de la **DEFI** à lui tout seul est Uniswap[8]. Il a été déployé à ses débuts sur la blockchain Ethereum. Désormais, le protocole est à sa version 3 et est présent sur 5 blockchains [2] pour un total d'actif sécurisé par ses smartcontracts de plus de 5 milliards de dollars[9].

Son succès a donné lieu à de nombreuses copies (**fork**) de son code vers d'autres blockchains qui ne disposaient pas de **DEX**s. La longévité de Uniswap ainsi que son nombre d'utilisateurs témoigne de la sécurité du protocole, aussi il n'est pas nécessaire pour les copieurs de réaliser un audit de sécurité. On observe donc beaucoup de **DEX**s qui sont des copies de Uniswap. Ils existent d'autres **DEX**s qui se basent sur la théorie introduite par Uniswap pour la gestion des prix, des transactions, mais qui n'est pas une pure copie, c'est le cas par exemple de Solidly.

2.2 Implémentation technique

Dans l'organisation de leurs smartcontracts les **DEX**s gardent une structure similaire. En effet, Ils disposent de deux smartcontracts principaux appelé **Factory** et **Router**. Le smartcontract **Factory** va créer la pool/pair où auront lieu les échanges d'un token A vers un token B et inversement. Il garde en mémoire l'ensemble des pools qui ont été créés et permet pour un token donné ou une paire de tokens de récupérer l'adresse de la pool d'échange associé. Le **Router** quant à lui réalise la gestion des pools comme une sorte de contrat tampon entre le contrat de la pool (crée par **Factory**) et l'utilisateur final qui souhaite réaliser un échange ou swap. Ainsi le routeur permet à un utilisateur d'ajouter ou de retirer de la liquidité, mais également de connaitre le prix dans une pool donnée, réaliser un swap etc.

3 Cas d'usage

Il peut être assez compliqué de trouver un arbitrage rentable. Cela demande de la patience, mais également un peu de chance et d'intuition quant aux tokens qui vont être soumis à cet arbitrage. Bien sûr, il est possible de surveiller les prix d'un ensemble de pools pour prendre connaissance des différences de prix entre plusieurs paires, mais cela demande des ressources importantes. Aussi le cas d'usage qui est présenté ici est résultant d'une intuition personnelle et non d'une surveillance.

3.1 Stratégie d'arbitrage

Tethys[10] est un **DEX** présent sur le réseau **Metis**[11]. Il est l'un des principaux **DEX** de cette blockchain avec **Netswap**[12], **Hermes**[13] et **Hummus**[14] qui concentrent la majorité de la liquidité. Tethys dispose d'un token ERC-20 du même nom, utilisé pour récompenser ses utilisateurs et est également utilisé dans le carde de la gouvernance du protocol[15]. La plateforme propose de **staker** ses Tethys et en échange, comme preuve de dépôt un autre token ERC-20 nommé xTethys est transféré à l'utilisateur. Un rapport d'échange est tout de même effectué, **1xTethys = 1.62 Tethys** [3] .Le dépôt de Tethys permet à l'utilisateur d'obtenir un rendement qui est à ce jour de 10%[16] comme le montre la figure 1.

1. Il est cependant à noter qu'il existe des **DEX**s qui sont conçues pour supporter l'échange de **NFT**. Cela permet de rendre ces actifs majoritairement illiquides en des actifs qui sont liquides.
2. Ethereum, Polygon, Optimisme, Arbitrum, Celo
3. Attention, ce taux est variable et dépend de la **T**otal **V**alue **L**ock (**TVL**) : soit la quantité de Tethys déposée.

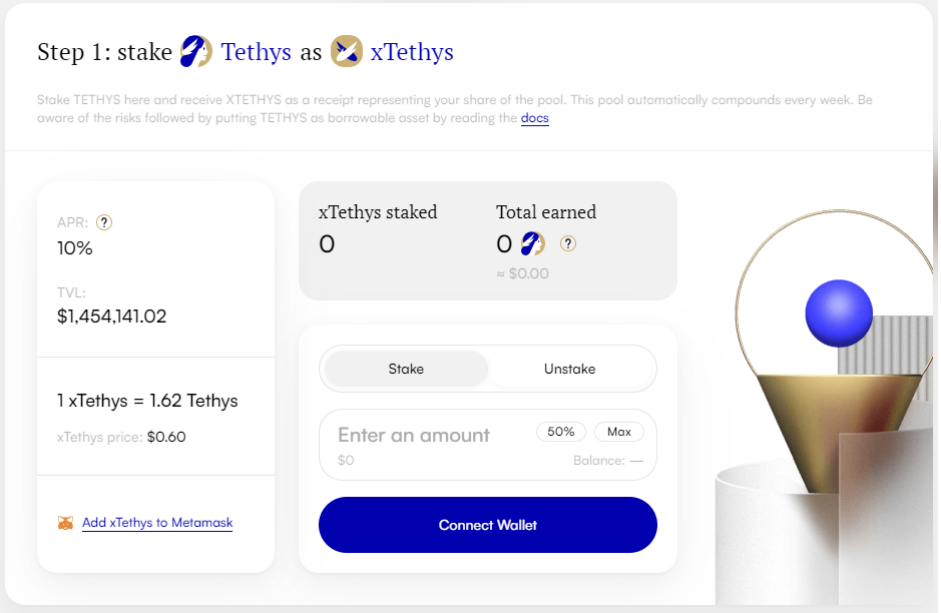

FIGURE 1 – Structure et interactions des smartcontracts au sein d'**DEX**

Le **DEX** Hermes a une pool qui permet d'échanger des Tethys et des xTethys où **1xTethys = 1.65424 Tethys** comme le montre la figure 2. **On constate donc une différence de prix entre le staking et la pool** :

— En staking : 1 $xTethys = 1.62$ $Tethys$

— En Pool : 1 $xTethys = 1.65$ $Tethys$

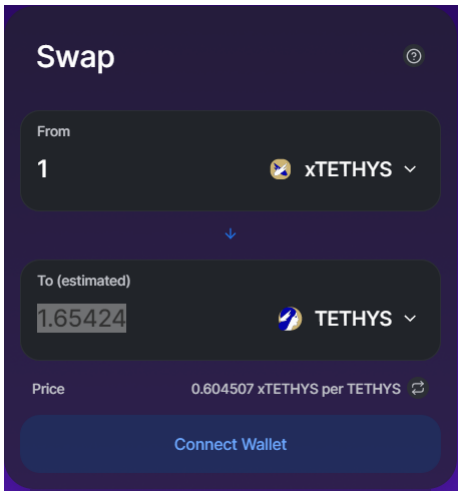

FIGURE 2 – Pool de liquidité Tethys/xTethys sur Hermes

Nous sommes alors face à une opportunité d'arbitrage. En effet, on pourrait emprunter des Tethys, réaliser le staking de ces Tethys pour obtenir des xTethys et enfin réaliser le swap sur le **DEX** Hermes des xTethys vers des Tethys et ainsi obtenir plus de Tethys que nous en avions au départ. Ainsi, nous pourrions rembourser notre emprunt en réalisant une plus petite value.

Pour réaliser notre arbitrage, nous allons donc devoir réaliser plusieurs étapes :

1. Emprunter des Tethys
2. Staker les Tethys pour obtenir des xTethys
3. Réaliser l'échange des xTethys vers les Tethys

3.2 Théorie sur les smartcontracts et leurs interfaces

Un smartcontract est un programme informatique. Les smartcontracts se comportent d'une part comme des acteurs que sont les wallets. C'est-à-dire qu'ils ont une adresse, identifiable, unique sur laquelle on peut envoyer des fonds. Cependant, comme c'est des programmes informatiques, il est nécessaire de programmer l'ensemble de leurs interactions avec un individu extérieur : un wallet ou un autre smartcontract. Il existe différentes blockchains sur lesquelles il est possible de déployer des smartcontracts. Et comme les smartcontracts sont des programmes informatiques, il est nécessaire de les programmer avec le marteau et l'enclume moderne : un langage de programmation ! Sur Metis le langage supporté est le solidity comme 90% des blockchains publiques [17]. Un smartcontract est caractérisé lors de son exploitation par son interface. Une interface en informatique est définie comme : **Dispositif qui permet la communication entre deux éléments d'un système informatique.** Aussi lorsque l'on effectue une communication d'un smartcontract A à un smartcontract B il est nécessaire que A dispose de l'interface de B qui décrit les différentes méthodes que A peut utiliser. La description d'une interface est réalisée en solidity par le mot clef **interface** auquel on ajoute à la suite le nom de celle-ci. On pourra y alors y spécifier les différents attributs ou méthodes que l'on peut utiliser. [4]

3.3 Code solidity Step-by-step

Dans cette dernière partie nous allons décrire ensemble step-by-step les grandes lignes de notre smartcontract pour réaliser notre arbitrage. Le but de cette dernière partie est de vous donner les grandes lignes techniques, mais nous ne pouvons nous arrêter sur tous les détails qui rentreraient plus dans la théorie de la programmation plutôt que de l'utilisation des outils de la **DEFI**.

3.3.1 Layout

Les premières lignes d'un code solidity commencent généralement par deux lignes que sont la licence et la version de solidity utilisée. La majorité des codes déployés et utilisés sur les différentes blockchains sont publics. Afin de se protéger de tous les aspects légaux, mais également favoriser la réutilisation de codes, [5] le compilateur solidity encourage à spécifier une licence pour permettre à tout utilisateur, développeur de savoir si il est possible ou non de réutiliser, modifier le code. La version de solidity quant à elle permet de disposer de certaines fonctionnalités du langage suivant la version utilisée. Aussi les premières lignes de notre contrat sont visibles figure 3

```
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

FIGURE 3 – Entête fichier solidity

4. Dans le cadre d'une communication d'un wallet à un smartcontract on utilisera l'**ABI** qui sera chargé par une librarie comme etherjs ou web3.js
5. pour les forks notamment

3.3.2 Importation de la libraire openzeppelin

La librairie **openzeppelin** est riche, mais nous n'avons besoin que de quelques interfaces telles que IERC-20, IERC3156FlashBorrower, IERC3156FlashLender. IERC-20 correspond à l'interface des tokens ERC-20, elle nous sera nécessaire pour récupérer les méthodes qui permettent d'autoriser la dépense de tokens, mais également réaliser des transferts. IERC3156FlashLender sera utilisé pour récupérer la méthode qui permet de déclencher le **FL** depuis l'emprunteur au prêteur. Enfin, IERC3156FlashBorrower est l'interface nécessaire à notre contrat pour pouvoir recevoir un **FL**.L'importation de ces différentes interfaces est décrite par la figure 4

```
import "@openzeppelin/contracts/interfaces/IERC-20.sol";
import "@openzeppelin/contracts/interfaces/IERC3156FlashBorrower.sol";
import "@openzeppelin/contracts/interfaces/IERC3156FlashLender.sol";
```

FIGURE 4 – Importation openzeppelin

3.3.3 Importation des interfaces spécifiques

Nous avons besoin d'autres interfaces qui ne sont pas prises en charge par openzeppelin. En effet, certains contrats que nous allons utiliser ne répondent pas à un standard. Nous allons donc devoir ajouter manuellement leurs interfaces. Nous avons besoin d'ajouter l'interface pour staker nos Tethys et d'une autre interface pour le routeur de Hermes qui nous permettra de connaitre le prix d'échange Tethys/xTethys, mais également de réaliser le swap des xTethys vers les Tethys.

Après rétro-ingénierie des contrats et de différentes transactions qui sont exécutés sur les premiers, le staking de Tethys est permis par le contrat à l'adresse **0x939Fe893E728f6a7A0fAAe09f236C9a6F4b67A18**. C'est sur cette même adresse qu'est déployé le token ERC-20 xTethys [6]. En bref, la méthode du contrat qui permet de staker nos Tethys est **enter**. Elle prend en argument le montant de Tethys que l'on souhaite staker. On peut alors définir une interface pour ce contrat (Voir figure 5).

```
interface IStaking {
    function enter(uint256 _amount) external returns (uint256 sharesAmount);
}
```

FIGURE 5 – Interface pour le staking de Tethys

La seconde interface qui nous est nécessaire est le routeur de Hermes à l'adresse **0x2d4F788fDb262a25161Aa6D6e8e1f18458da8441** afin de connaitre le prix courant dans la pool [7] et réaliser notre swap. Après rétro-ingénierie du routeur la méthode **getAmountOut** permet de récupérer le nombre de tokens de sorties en fonction du montant de token en entrée dans une pool. Enfin, **swapExactTokensForTokensSimple** nous permet de réaliser le swap d'un token A vers un token B. L'interface du routeur est décrite ci-dessous figure 6

6. https ://andromeda-explorer.metis.io/address/0x939Fe893E728f6a7A0fAAe09f236C9a6F4b67A18/transactions
7. Combien de Tethys pour mon montant de xTethys

```
interface IBaseV1Router01 is IERC-20 {
    function getAmountOut(
        uint256 amountIn,
        address tokenIn,
        address tokenOut
    ) external view returns (uint256 amount, bool stable);

    function swapExactTokensForTokensSimple(
        uint256 amountIn,
        uint256 amountOutMin,
        address tokenFrom,
        address tokenTo,
        bool stable,
        address to,
        uint256 deadline
    ) external returns (uint256[] memory amounts);
}
```

FIGURE 6 – Interface pour le routeur de Hermes

Vous remarquez que les contrats ont beaucoup plus de méthodes, mais nous n'avons besoin que d'une petite partie des méthodes/fonctions qu'ils implémentent. Aussi, il n'est pas nécessaire de toutes les ajouter lors de la déclaration de l'interface.

3.3.4 Initialiser le FL

Pour réaliser un **FL**, nous devons exécuter la méthode **flashLoan** de notre prêteur (qui se trouve à la même adresse que Tethys). Elle prend plusieurs arguments :

— receiver : l'adresse qui va recevoir le FlashLoan et qui doit implémenter IERC3156FlashBorrower

— token : L'adresse du token qui est emprunté

— amount : le montant que l'on souhaite emprunter

— data : d'éventuelles données que l'on souhaite passer à la callback du **FL**

Avant d'exécuter cette méthode, nous devons d'abord récupérer le montant des intérêts liés au **FL** et autoriser le prêteur (le contrat auquel on réalise le **FL**) à récupérer automatiquement le montant du prêt avec les intérêts. Nous pouvons rassembler l'ensemble de ces opérations au sein de la méthode **arbitrage** avec laquelle notre wallet pourra interagir (voir figure 7).

Pour récupérer les intérêts liés au **FL**, on utilise la méthode **flashFee**. Elle prend en arguments l'adresse du token que l'on souhaite emprunter ainsi que le montant. Le prêteur peut faire varier les intérêts suivant le token et/ou le montant emprunté. Pour autoriser le prêteur à dépenser les tokens, on doit à partir du contrat du token que l'on souhaite emprunter autoriser le prêteur à dépenser le montant du prêt plus les intérêts. Cela est réalisé par la méthode **approve** du token et rend 2 arguments :

— spender : l'adresse qui est autorisée à dépenser les tokens

— amount : Le montant que l'adresse *spender* est autorisée à dépenser

```
function abitrage(uint256 amount, uint256 deadline_) public {
    uint256 _allowance = ITethys.allowance(address(this), address(lender));

    // On récupère le montant des fees à payer pour le flashloan. Pour Tethys = 0 Mais p
    uint256 _fee = lender.flashFee(address(ITethys), amount);

    // Montant total à rembourser
    uint256 _repayment = amount + _fee;

    // On précise que lender peut dépenser peut dépensser _repayment
    ITethys.approve(address(lender), _allowance + _repayment);

    // Appel du flash loan
    lender.flashLoan(this, address(ITethys), amount, data);
}
```

FIGURE 7 – Appel du **FL**

3.3.5 FlashLoan Callback

Un smartcontract est défini en solidity comme l'indique la figure 8. Le mot clef **contract** nous permet de définir le début d'un contrat auquel on ajoute à la suite le nom. À la figure 8 le nom du smartcontract est donc **MonSmartContract**.

```
contract MonSmartContract {
    constructor() {
    }
}
```

FIGURE 8 – Exemple de définition d'un smartcontract

On peut réaliser un ou des héritages[8] grâce au mot clef **is**. L'héritage nous permet d'ajouter des fonctionnalités à notre contrat en reprenant un modèle de contrat déjà rédigé. Le standard ERC-3156 impose à notre smartcontract d'être un contrat de type IERC3156FlashBorrower. Grâce à openzeppelin, nous avons pu importer l'implémentation associée. Pour faire hériter à notre smartcontract l'IERC3156FlashBorrower il suffit de procéder comme indiqué figure 9.

```
contract MonSmartContract is IERC3156FlashBorrower{
    constructor() {
    }
}
```

FIGURE 9 – Héritage IERC3156FlashBorrower

Notre contrat doit également disposer d'une méthode appelée **onFlashLoan** comme décrit dans le standard ERC-3156. Celle-ci est exécutée en tant que callback dès que le **FL** est validé et que le montant est transmis au contrat. Elle prend plusieurs arguments :

— initiator : l'adresse qui a initié le FlashLoan

— token : l'adresse du token qui a été emprunté

8. L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Ici on ajoute des fonctionnalités à notre contrat en lui faisant hériter des fonctionnalités issues d'autres contrats

— amount : le montant de l'emprunt

— fee : les intérêts de l'emprunt (dans notre cas 0, mais sur AAVE c'est 0.09%),

— data : des données que l'on souhaite transmettre suite au déclenchement de notre emprunt.

Cette méthode doit retourner la valeur keccak256("ERC3156FlashBorrower.onFlashLoan"). Je ne rentre pas dans les détails, mais une vérification est réalisée par le contrat prêteur sur la valeur retourné, il est possible de le vérifier sur l'implémentation du prêteur[18]. Le problème avec notre méthode **onFlashLoan** est qu'elle agit comme callback à l'appel de n'importe quel **FL**. N'importe qui peut donc exécuter notre méthode. Il faut donc vérifier par sécurité que c'est bien l'adresse de notre prêteur qui appelle notre méthode **onFlashLoan** et que l'adresse à l'initiative du **FL** est bien celle de notre contrat. On peut réaliser ces vérifications grâce à des assertions que l'on implémente via le mot clef **require**. Les assertions vont tester une condition et renvoyer une erreur si elle n'est pas satisfaite. Dans notre cas, voir figure 10

```
require(
    msg.sender == address(lender),
    "FlashBorrower: Untrusted lender"
);

require(
    initiator == address(this),
    "FlashBorrower: Untrusted loan initiator"
);
```

FIGURE 10 – Sécurité

On déduit alors l'implémentation minimale de notre callback ci-dessous figure 11

```
contract MonSmartContract is IERC3156FlashBorrower{
    constructor() {
    }
    function onFlashLoan(
        address initiator,
        address token,
        uint256 amount,
        uint256 fee,
        bytes calldata data
    ) external override returns (bytes32) {
        require(
            msg.sender == address(lender),
            "FlashBorrower: Untrusted lender"
        );

        require(
            initiator == address(this),
            "FlashBorrower: Untrusted loan initiator"
        );


        // ICI ON FAIT TOUTES LES OPERATIONS QUE L'ON SOUHAITE
        // AVEC NOTRE FLASH LOAN

        return keccak256("ERC3156FlashBorrower.onFlashLoan");
    }
}
```

FIGURE 11 – Implémentation minimale pour la callback du **FL**

3.4 Opération à partir du FlashLoan

À la section précédente, nous avons déterminé le code minimal de la callback pour notre **FL**. Maintenant il ne reste plus qu'à réaliser l'arbitrage. D'abord le staking de Tethys en xTethys. Nous avons vu que cela était réalisé à partir de la méthode **enter** auquel on ajoutait en argument à savoir le montant que l'on souhaite de Tethys que l'on souhaite staker (ici la totalité de notre FlashLoan). Auparavant nous devons autoriser le contrat de staking à récupérer les Tethys disponibles sur notre contrat. Cela se fait comme dans les sections précédentes avec la méthode **approve**. On peut ensuite récupérer le montant de XTethys dont on dispose à la suite de dépôt. Voir figure 12

```
ITethys.approve(address(ITethysStaking), amount);
ITethysStaking.enter(amount);
uint256 xTethysAmount = IxTethys.balanceOf(address(this));
```

FIGURE 12 – Staking Tethys

Ensuite, on veut réaliser le swap de xTethys vers Tethys. On doit autoriser le routeur d'Hermes à dépenser nos xTethys pour les échanger contre des Tethys (méthode **approve**). Ensuite, avec la méthode **getAmountOut** on récupère le montant de Tethys que l'on va obtenir à la suite de notre swap xTethys vers Tethys. Et enfin avec **swapExactTokensForTokensSimple** on réalise notre swap.

```
IxTethys.approve(address(HermesRouter), xTethysAmount);

(amountOutMin, stable) = HermesRouter.getAmountOut(
    xTethysAmount,
    address(ITethysStaking),
    address(ITethys)
);

HermesRouter.swapExactTokensForTokensSimple(
    xTethysAmount,
    amountOutMin,
    address(IxTethys),
    address(ITethys),
    stable,
    address(this),
    deadline
);
```

FIGURE 13 – Swap XTethys vers Tethys

4 Perspectives

Nous venons de réaliser le code dédié à arbitrage Tethys/xTethys avec une plus-value grâce à un **FL** implémenté dans notre smartcontract. Nous pouvons grâce à cela exploiter une différence de prix et ainsi réaliser un profit sans pour autant disposer de fonds. Cependant, nous pouvons établir des critiques dans notre cas précis d'utilisation. En effet, nous exploitons uniquement la différence de prix entre le staking et une pool de liquidité. Mais nous ne profitons pas des rendements issus du dépôt en staking de Tethys à 10%. En effet, nous n'avons pas implémenté la gestion du staking afin de profiter des revenues issues de celui-ci. À chaque arbitrage de Tethys/xTethys nous augmentons la proportion de Tethys stakée par notre contrat et donc notre rendement. Imaginons si nous réalisons 5 Arbitrages à 163 Tethys, nous aurons alors déposé 815 Tethys en staking avec un revenu de 81,5 Tethys par an. À son plus haut niveau, le Tethys valait 6.61$ [19] soit un gain potentiel de 538$ et cela sans aucun fonds nécessaire. Plus vous allez réaliser des arbitrages, plus cette valeur va augmenter. Il serait donc intéressant d'ajouter au smartcontract la possibilité de récupérer les récompenses liées au staking de Tethys.

5 Conclusion

Le code complet solidity ainsi que cet article est disponible sur le GitHub de l'utilisateur 0x8479828583 (https ://github.com/0x8479828583) au repository TethysArbitrageFlashLoan.

Les différentes transactions réalisées avec un arbitrage **FL** peuvent être consultées à la Table 1 avec les hashs des transactions.

TX HASH	Montant du FlashLoan	Profit
0x1f25f1d1c7e82ad215ae6ffeec9858efb9b1205ea73c74bc13fdefe4b7d0e212	163 TETHYS	5.38 TETHYS
0xc38fed11e2b766babf3b17c20b2bf19140803cb5745441dbc044b2cc86ba8853	163 TETHYS	3.86 TETHYS
0xb83d917dfab0079015331eddfc309fca40141f53ce5dd434afe365458e386afa	163 TETHYS	2.36 TETHYS

TABLE 1 – Visualisation des profits pour différents arbitrages

Références

[1] Ce hacker plus malin que les autres se fait 820 000 dollars en un clic, sur la binance smart chain. Journal du Coin. [Online]. Available : https://journalducoin.com/defi/

hacker-malin-820000-dollars-clic-binance-smart-chain/

[2] Deus Finance flash loan exploit nets hacker $13 million. The Block. [Online]. Available : https://www.theblock.co/linked/143969/deus-finance-flash-loan-exploit-nets-hacker-13-million

[3] Flash loan attack on One Ring protocol nets crypto-thief $1.4 million. The Daily Swig | Cybersecurity news and views. [Online]. Available : https://portswigger.net/daily-swig/flash-loan-attack-on-one-ring-protocol-nets-crypto-thief-1-4-million

[4] S. Ikeda. Flash Loan Attack Takes Beanstalk Defi Platform for $182 Million, Largest Yet of Its Type. CPO Magazine. [Online]. Available : https://www.cpomagazine.com/cyber-security/flash-loan-attack-takes-beanstalk-defi-platform-for-182-million-largest-yet-of-its-type/

[5] EIP-3156 : Flash Loans. Ethereum Improvement Proposals. [Online]. Available : https://eips.ethereum.org/EIPS/eip-3156

[6] Interfaces - OpenZeppelin Docs. [Online]. Available : https://docs.openzeppelin.com/contracts/4.x/api/interfaces

[7] Aave - Open Source Liquidity Protocol. [Online]. Available : https://aave.com/

[8] Home. Uniswap Protocol. [Online]. Available : https://uniswap.org/

[9] DefiLlama. DefiLlama. [Online]. Available : https://defillama.com/protocol/uniswap

[10] Tethys. [Online]. Available : https://tethys.finance/

[11] Metis. [Online]. Available : https://www.metis.io/

[12] Netswap. [Online]. Available : https://netswap.io/#/home

[13] Hermes Protocol. [Online]. Available : https://hermes.maiadao.io/#/swap

[14] Hummus. [Online]. Available : https://www.hummus.exchange/

[15] Propositions Tethys Finance. [Online]. Available : https://gov.tethys.finance/#/

[16] Tethys. [Online]. Available : https://tethys.finance/staking/tethys

[17] DefiLlama. DefiLlama. [Online]. Available : https://defillama.com/languages

[18] "OpenZeppelin/openzeppelin-contracts," OpenZeppelin. [Online]. Available : https://github.com/OpenZeppelin/openzeppelin-contracts/blob/ecd2ca2cd7cac116f7a37d0e474bbb3d7d5e1c4d/contracts/token/ERC20/extensions/ERC20FlashMint.sol

[19] Cours de tethys finance, graphique de tethys et capitalisation boursière. CoinGecko. [Online]. Available : https://www.coingecko.com/fr/pi%C3%A8ces/tethys-finance