Protecting smart contracts of Decentralized Finance systems against Reentrancy attacks

Nafie El Coudi El Amrani , Oğuzhan Ersoy , Zekeriya Erkin
Delft University of Technology

Abstract

Reentrancy attacks target smart contracts of Decentralized Finance systems that contain coding errors caused by developers. This type of attacks caused, in the past 5 years, the loss of over 400 million USD. Several countermeasures were developed that use patterns to detect reentrancy attacks on smart contracts before deployment on the Ethereum blockchain. However, the smart contracts are by default public and immutable once deployed on the blockchain. That is why the research question is: How can we protect smart contracts of DeFi systems deployed on the Ethereum blockchain that are known to be vulnerable to reentrancy attacks? A solution that detects reentrancy attacks on smart contracts after their deployment is presented in this paper. It flags transactions when a difference is found between the users' funds on both the application and protocol layers before and after each transaction using special made smart wallets. A proof of concept shows that the proposed solution can detect reentrancy attempts and stop them during the execution phase of smart contracts.

1 Introduction

Decentralized finance (DeFi) is a system where a variety of financial applications are available publicly on a decentralized blockchain network in order to get rid of financial intermediaries. Users of DeFi can lend, borrow, trade and insure risks without any central interference. In January 2021, the total locked value (TLV) in Decentralized Finance systems was around 25 billion USD [1]. The logic behind decentralized finance systems is encoded in their respective smart contracts. It allows these systems, otherwise called: Decentralized applications (dApps), to perform complicated financial operations such as: lending and trading assets mainly in the form of cryptocurrencies. Therefore, any bug exploit or vulnerability on the smart contract level directly puts the entire decentralized finance system at risk of being exploited.

In the early days of DeFi, precisely in 2016, an attacker managed to drain 150 million USD [2] from a crowdfunding smart contract called DAO. The attack exploited a reentrancy bug present in the underlying smart contract of the protocol by calling recursively a payout function. This attack was significant to the point that the Ethereum [3] market cap dropped by over 600 million USD in one day [4]. This was not the only consequence of the attack, it caused a split in the community that led to a hard fork and the birth of Ethereum classic blockchain [2].

Another important example to cite is the delayed Constantinople update of Ethereum in January 2019. The update aimed to introduce gas cost for several smart contract operations. However one day prior to the update's launch date, it was found that the gas cost reduction can cause some already deployed smart contracts on the Ethereum blockchain to become vulnerable to reentrancy attacks. This has delayed the push of the update [5].

The academic community have shown interest in researching solutions to make smart contracts secure against reentrancy attacks. These proposed solutions range from creating a safer programming language [6] to creating tools that detect and prevent reentrancy attacks before deployment on the blockchain. Oyente [7] leverages symbolic execution to analyse smart contracts and flag security vulnerabilities.

Even though the academic community have conducted several research on the topic, most of the tools mentioned earlier try to detect reentrancy vulnerabilities during the development stage of the smart contract. Since these contracts are deployed on the Ethereum blockchain, they are immutable after deployment which makes correcting the mistakes in the code impossible. This leaves smart contracts vulnerable to attacks and in need of protection after deployment.

In order to keep Decentralized Finance systems safe and prevent costly attacks on this new thriving platform, more academic research should be conducted. The research question of this paper is: How can we protect smart contracts of DeFi systems deployed on the Ethereum blockchain that are known to be vulnerable to reentrancy attacks?

The paper presents the different parts of the research as follows. Section 2 formally defines all the key concepts, systems included in this research. It also presents a formal definition of reentrancy attacks and its types will be presented. Section 3 describes the methodology adopted during the research. Then, section 4 presents the different existing tools

and methods to protect smart contracts and discusses their limitations. In section 5, we present the suggested solution to the reentrancy problem in smart contracts. Next in Section 6, a proof of concept of the solution is conducted on three types of reentrancy attacks. Then, a discussion about the results of this research is presented in 7. Next, section 8 follows with a discussion on the ethical implications of this research. Finally, the paper ends with a conclusion in Section 9 and some suggestions for future work.

2 Background

In this section, Decentralized Finance systems and their applications are presented in subsection 2.1. Then, in subsection 2.2, we define smart contracts and their applications. Finally, a formal definition of reentrancy attacks is introduced and a description of some types of reentrancy attacks is presented in subsection 2.3.

2.1 Decentralized finance systems

As briefly introduced in the introduction, Decentralized Finance systems provide a financial architecture that allows users to benefit from financial products such as lending, borrowing, trading and exchanging currencies in a permissionless, non-custodial and anonymous way. This novel financial architecture is built on relatively new technology; blockchain networks. These networks allow users to create and innovate new financial protocols by developing new smart contracts, discussed in detail in subsection 2.2. Currently, the most used blockchain network to develop new smart contracts and new financial protocols is Ethereum [3].

Decentralized Finance is relatively new and has been growing rapidly during the last 6 years, reaching a total locked value of 88 billion USD in May 2021 [1]. However, every novel technology comes with its own risks and DeFi is no exception. Decentralized Finance systems are the target of dozens of attacks every year [8]. The attacks carried out on DeFi are of different types. For instance, exploits of technical errors such as exploits of bugs in the smart contracts, reentrancy attacks fall under this category. Or, economical exploits that can make users of DeFi lose funds such as market or oracle manipulations. Another important risk worth mentioning is the fact that DeFi facilitates all sorts of financial crime. Hackers, money launderers and scammers can freely move and clean funds easily. This is due to the anonymous nature of Decentralized Finance.

2.2 Smart contracts

Smart contracts are computer programs stored on the blockchain that gets executed by all the nodes when certain conditions are met. They consist of self-contained code written in specialized programming languages. In the case of the Ethereum blockchain, Solidity is the most used high-level programming language to create smart contracts of DeFi systems [9]. The code gets compiled to bytecode, which in its turn gets interpreted by the Ethereum Virtual Machine (EVM).

To incentivize the nodes of the Ethereum blockchain to execute smart contracts, the sender of the request to execute a smart contract pays a small fee called: gas to the nodes of the network. The amount of gas needed for a smart contract relates to the cost of executing a smart contract and is paid in Ether [10]. Moreover, every smart contract has a fallback function that gets executed when the name of the function in the data field doesn't get recognized by the corresponding smart contract. This function is important in the design of some reentrancy attacks that will be discussed in the next subsection 2.3.

Moreover, smart contracts are by default immutable, meaning that they cannot be updated or removed from the blockchain after deployment. Unless the creator(s) of the smart contract design it keeping mind upgradeability or destroyability. Unfortunately, many contracts already deployed on the Ethereum blockchain don't have the options to be deleted or updated. Thus any existing in these contracts cannot be fixed.

2.3 Reentrancy attacks

Reentrancy attacks occur when a smart contract A calls an (external) smart contract B which calls back the fallback function of A. These successive actions happen in one transaction. However, reentrancy in itself is not always malicious, it happens during legitimate smart contract execution and is one of the patterns supported by the Ethereum blockchain [11].

To explain in details how the reentrancy pattern should work normally, an example (inspired from [12]) is shown in listing 1, contract A tries to withdraw 10 units from its balance in contract B. Contract A starts by calling the *withdraw* method of contract B. Then contract B calls *msg.sender.value* (it is equivalent to calling contract A) to transfer 10 units to contract A. This last action is where contract A reenters contract B using its callback function. The function in line 2 without a name is the callback function of contract A. This is necessary for Ethereum and it is expected behavior to be able to transfer funds between smart contracts.

```
1  contract A {
2      function () public payable { }
3      function f() {
4          b.withdraw(10);
5      }
6  }
7
8  contract B {
9      function withdraw(uint amount) public
           {
10         msg.sender.send.value(amount)();
11     }
12 }
```

Listing 1: An example of legitimate reentrancy in a smart contract [12].

Since reentrancy is expected in smart contracts and supported by the Ethereum blockchain, a reentrancy is considered an attack when it is unexpected by the creators of the smart contract making it operate in an inconsistent internal state. In other words, a reentrancy call is malicious if the update of the internal state happened after an external call is

returned. In listing 4, an example (inspired from [13]) of a vulnerable function to reentrancy attacks is presented. The *withdraw()* function should perform 3 actions. First, check if the calling smart contract is entitled to the requested amount. Second, transfer the requested amount to the calling party. And, finally, update the balance of the caller. The main issue in the *withdraw()* function is the order of the second and third actions because it allows a malicious third party to reenter the contract and call *withdraw()* function with the same amount as the first call. Hence, an attacker can repeatedly call the *withdraw()* function and keep reentering the smart contract until the latter is drained of funds. A relatively easy fix for this problem would be to swap lines 3 and 4, making the function update the internal state of the smart contract before calling the attacker's contract. Preventive methods on how to prevent reentrancy attacks will be briefly presented in subsection 4.2. The example presented here is one of the main type of reentrancy attacks: **Single function reentrancy attack**.

```
1 function withdraw(uint amount) public {
2     if (credit[msg.sender] >= amount) {
3         msg.sender.call.value(amount)();
4         credit[msg.sender] -= amount;
5     }
6 }
```

Listing 2: An example of vulnerable function to single function reentrancy attacks [12].

Other types of reentrancy attacks were presented by academia in different articles and can be summarized in several additional types.

The first type is **Cross-Function Reentrancy**. Its main idea is to reenter the victim's smart contract from different functions instead of only one. It was shown that this type of reentrancy attacks is as dangerous as the single function reentrancy attack [12] because they both do achieve the same result which is repeatedly withdrawing funds until all funds of the victim's smart contract are drained. This type of attack is possible when a vulnerable function in a smart contract shares a state with another function in the same contract that has an effect desired by an attacker. In listing 3, an example (inspired from [14]) shows that the *withdraw* calls the attacker's fallback function the same as in single function reentrancy attacks. However, the difference lies in the callback function that makes a call to the *transfer()* function instead of the the *withdraw()* function because the balance has not been reset to 0 yet before this call. Hence, allowing the *transfer()* to transfer an amount that has already been transferred.

```
1  function transfer(address to, uint amount
       ) public {
2      if (balances[msg.sender] >= amount) {
3          balances[to] += amount;
4          balances[msg.sender] -= amount;
5      }
6  }
7
8  function withdraw() public {
9      uint amount = balances[msg.sender];
10     require(msg.sender.call.value(amount)
           ());
11     balances[msg.sender] = 0;
12 }
```

Listing 3: An example of a smart contract vulnerable to Cross-Function reentrancy attack [14].

The final type is different from the previously mentioned attacks because it represents a reentrancy vulnerability but can only be feasible under specific conditions such as the introduction of low gas costs in the Constantinople update of Ethereum in 2019. No formal name was given to this attack, so we will refer to it as the **Constantinople reentrancy** attack. The attack can be performed on smart contracts that allow users to share funds. This issue was found by [5] and caused a delay in the roll-out of the new update of Ethereum. An attacker with two accounts can perform the attack by following the next steps:

1. Create a shared account between his two accounts.
2. Update the split of the share, 100% to the main account and 0% to the other.
3. Withdraw 100% of the funds to his main account.
4. In the fallback function, update the split again; 0% to the main account and 100% to the other.
5. Withdraw 100% of the funds again to his second account.

At the end of the transaction, the attacker will steal funds from the other participants in the smart contract. To be able to perform this attack steps 3,4 and 5 should happen in one transaction block. That is why this attack wasn't possible before introducing the lowering of gas costs.

```
1 function splitFunds(uint id) public {
2     address payable a = first[id];
3     address payable b = second[id];
4     uint depo = deposits[id];
5     deposits[id] = 0;
6     a.transfer(depo * splits[id] / 100);
7     b.transfer(depo * (100 - splits[id])
          / 100);
8 }
```

Listing 4: An example of vulnerable function to Constantinople reentrancy attacks [5]

3 Methodology

To answer the main question of this research project, literature analysis of existing research and innovations presented by both the academic community and the companies involved in the Decentralized Finance systems world. Several academic research has been conducted on protecting smart contracts of DeFi systems on the Ethereum blockchain. A formal discussion about these research projects is presented in section 4. Then, a new solution to protect already deployed smart contracts on the Ethereum blockchain is presented. Followed by a proof of concept that is shown by implementing the proposed tool in 3 different test cases for three reentrancy attack types: single function, Cross-Function and Constantinople reentrancy attacks.

4 Related work and their limitations

In this section, detection tools created by previous research are presented in Subsection 4.1. Then, known preventive methods that can protect smart contracts during development are shown in Subsection 4.2. And in the last Subsection 4.3, some limitations of the existing methods are presented.

4.1 Existing detection tools

In this subsection, a variety of tools are used to detect reentrancy vulnerabilities off-chain (before deployment on the Ethereum blockchain) and on-chain (after deployment).

Off-chain tools

Several strategies are used to detect reentrancy vulnerabilities off-chain. One of the most used ones is Symbolic execution. It is an old program analysis technique introduced to computer science in the 70s, used to determine the inputs that can cause the execution of each part of the program [15]. Using this strategy, the tools can reason statically path by path about a program.

Oyente [7], for instance, obtains the path condition before the execution of each CALL transaction. It then checks if this condition still holds with different variables such as storage values in the smart contract of the DeFi system. ***Maian*** [16] and ***teEther*** [17], on the other hand, use symbolic execution differently. They try to find a sequence of invocations leading to vulnerabilities such as reentrancy attacks.

Securify [18] leverages static analysis to analyze smart contracts in a scalable and automated way. It detects unsafe behavior of smart contracts based on provided patterns and properties. It first, symbolically analyses the dependency graph of the smart contract's code to extract the semantic information. Then, it checks whether the smart contract contains compliance and violence patterns to decide if the contract has any reentrancy vulnerability.

Several other tools use symbolic execution in combination with other strategies such as ***Mythril*** [19]. It uses both taint analysis and symbolic execution to detect cybersecurity vulnerabilities including reentrancy attacks.

Several other researchers have decided to approach the problem with dynamic solutions. For instance, some used fuzzing-based techniques to test the smart contracts. This has the benefit of allowing the detection tools to analyze the smart contracts by themselves or in combination with each other, thus detecting cross-function reentrancy attacks more accurately. ***ReGuard*** [20] first translates Solidity code of the smart contracts to C++ code, then it uses a fuzzing engine to generate random sequences of transactions. This generated input is run in an automata designed to flag sequences that can reach one error state. Reaching this state means that a reentrancy vulnerability has been detected. Other tools have opted to use fuzzing engines in a more structured way. In [21], the authors have used the patterns discovered in previous research by symbolic execution to generate sequences of transactions that have a higher probability of finding reentrancy vulnerabilities than sequences generated randomly. ***Harvey*** [22] uses greybox fuzzing to generate tailored sequences to detect reentrancy attacks.

Other strategies were used to tackle this problem, ***Zeus*** [23] uses symbolic model checking, constrained horn clauses and abstract interpretation to verify the safety of smart contracts. ***SmartCheck*** [24] translates the smart contract's source code to a parse-tree based on XML and then looks for violation patterns using XPath queries.

On-chain tools

All tools mentioned previously can find reentrancy attacks outside of the Ethereum blockchain, the 3 tools presented next are tools used to protect the smart contracts that are already deployed on the Ethereum blockchain. ***ECFChecker*** [25] leverages Effectively Callback Free (ECF) to detect reentrancy attacks. An execution with an equivalent execution that can achieve the same state without callbacks is called ECF. And a smart contract is called an ECF contract if all its executions are ECFs. Since callbacks can affect state transitions during contract execution, non-ECF smart contracts are vulnerable to reentrancy attacks. It is important to mention here that statically proving that a smart contract is ECF is considered an undecided problem up until now. However, the creators of ***ECFChecker*** have come up with a way to dynamically prove if the ECF property of a smart contract has been violated by a transaction.

ÆGIS [4] has introduced a strategy different than of the existing tools. It analyses the source code data flow then it compares the pattern of the data flow to several patterns stored in its smart contract. This is not particularly different than the strategies adopted by other tools. However, the new idea that helps *ÆGIS* as an online tool is the users' ability to propose new patterns to ban and vote whether the new proposed pattern should be incorporated in the tool or not. This helps *ÆGIS* to keep detecting any newly detected vulnerability. However, this option introduces other problems that other Decentralized Finance systems are already facing like how to determine the eligibility of a voter or how to incentivize users to take part in the voting process. These problems are outside the scope of this paper.

One of the competitors of *ÆGIS* is ***Sereum*** [12]. It tackles the problem of detecting reentrancy attacks based on validation and run-time monitoring in a backwards-compatible way. It leverages taint analyses and is built on the assumption that every vulnerability originates from a transaction.

4.2 Preventive methods

In this subsection, some preventive methods are suggested by the Ethereum community to prevent before deploying smart contracts on the blockchain.

A document called: Ethereum Smart Contract Security Best Practices, maintained by ConsenSys Diligence [26] presents a wide range of best practices to help developers avoid several types of security vulnerabilities including reentrancy attacks.

- **Checks-effects-interactions pattern**: It forces the developers to first check if the state of the contract is the one the developer is expecting using `require` or `assert`. Then they should resolve any effects of the methods. After the first two steps are done, the developers can interact with an external contract.

- **Usage of Mutex**: Mutual exclusion makes sure that a part of the program cannot be accessed a second time unless the first call has already been done. Libraries like: *ReentrancyGuard* [27] are accessible on Solidity and helps developers to lock parts of the code until the call is done to avoid reentrancy.

Other members of the academic and Ethereum community have opted to focus on building safer programming languages for smart contracts. One of the most maintained languages for smart contracts other than Solidity is ***Vyper*** which is a high-level programming language with the goal of better code security and easier security audits. Its creators have excluded, on purpose, features that can lead to misleading code such as recursive calling and inheritance [28]. Another language, less used than Vyper, is ***Obsidian***. It leverages linear types to flag abuse of assets and typestate to detect malicious state manipulation [29]. The paper by Tyurin et al. gives a detailed overview of several programming languages for smart contracts [30].

4.3 Limitations

In this subsection, the limitations of the previously mentioned tools and methods are discussed.

- Since most smart contracts already deployed on Ethereum are immutable or there is no way to find their creators, off-chain tools cannot protect them.
- Several tools discussed in Subsection 4.1 such as ***Oyente*** and ***Securify*** use known patterns to detect reentrancy vulnerabilities. This limits their ability to detect new types of reentrancy attacks, and they need manual updates to add them once the patterns are publicly known.
- Tools leveraging symbolic execution suffer from the path explosion problem [31] that is still an ongoing research topic.
- Dynamic strategies such as fuzzing use significant amounts of resources by running tests on input that doesn't lead to reentrancy vulnerabilities.
- The preventive methods can only be performed during the production of smart contracts which renders them useless in protecting already deployed contracts.

5 The *SmartTool* solution

In this section, a solution *SmartTool* to protect smart contracts already deployed on Ethereum is presented. *SmartTool* can be implemented in different ways, can be used on the Ethereum blockchain and can flag malicious transactions without using any patterns.

The main idea behind SmartTool is the same one used by [32]. It all comes down to the idea that reentrancy exploits are caused by the difference of balance values between the protocol layer and the application layer. All smart contracts that provide fund managing services rely on two values to perform the internal bookkeeping. The first one is the value stored on the application layer, which is usually stored in a mapping between the addresses of participants and their corresponding funds. The second one is the value stored on the protocol layer which makes it protected by the miners who maintain the Ethereum blockchain. The changes on both values should be the same before and after every transaction to avoid reentrancy vulnerabilities.

The attackers try to trick the smart contracts to cause a difference in their balance values on the application and protocol layers. The only way they can do this is to manipulate the values on the application since they cannot change the values on the protocol layer that is protected by the miners. If the attackers succeed in creating a discrepancy between the layers by having a higher balance on the application layer than the balance in the protocol layer, they can steal funds from the smart contract. And the latter will not be able to stop it since it is not aware of the discrepancy.

Therefore, the solution is to check the values on both layers before and after each transaction and flag malicious transactions that can create discrepancies between the values in the two layers. However, keeping track of only the sum of balances of all participants in the application layer and the funds stored on the smart contract on the protocol layer will not be able to catch reentrancy attacks such as the one discovered by ChainSecurity [5]. This approach was adopted by [32].

Thus, the SmartTool will keep track of each user's funds in both the protocol and application layers instead of keeping track of only the sum of all participants' funds. This will be done by creating s special made wallet for each user that deposits funds to the smart contract. The introduction of such wallets will allow the main contract to get the balance of each user from the protocol layer so that it can compare it to the balance on the application layer.

Moreover, SmartTool can be implemented in three different ways:

1. Implement the logic of the SmartTool directly on the vulnerable smart contract.
2. Implement it as an off-chain tool.
3. Implement it on the blockchain on a different contract.

Workflow of SmartTool

In figure 1, the workflow of the SmartTool is presented. The SmartTool first gets the balance u_i of each $user_i$ from the associated helper wallets (protocol layer) and the balance v_i from the smart contract (application layer) before each operation. Then after the operation is executed, the SmartTool gets again the same balances $\overline{u_i}$ & $\overline{v_i}$ for every $user_i$. Next, the tool performs the integrity checks in two ways:

- Transfer operation from $user_a$ to $user_b$ with value t, the SmartTool will check if:
  1. u_a - $\overline{u_a}$ = v_a - $\overline{v_a}$ = t
  2. u_b - $\overline{u_b}$ = v_b - $\overline{v_b}$ = $-t$
  3. For every other $user_i$: u_i - $\overline{u_i}$ = v_i - $\overline{v_i}$ = 0
- Withdraw operation from $user_a$ with value t, the SmartTool will check if:
  1. u_a - $\overline{u_a}$ = v_a - $\overline{v_a}$ = t
  2. For every other u_i - $\overline{u_i}$ = v_i - $\overline{v_i}$ = 0

If any check from both cases fail, then the SmartTool will flag the operation as malicious and will store the address of the *attacker* to ban it in next operations. Otherwise, the next operation is tackled following the same workflow.

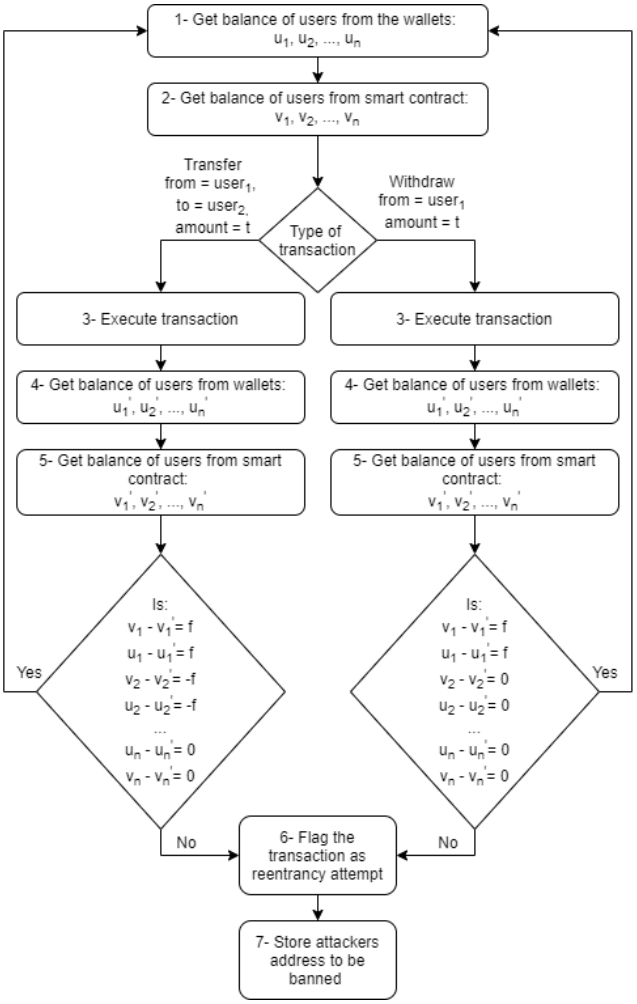

Figure 1: Workflow of *SmartTool*

6 Proof of concept of SmartTool

In this subsection, 3 cases are presented to show that SmartTool can stop different types of reentrancy attempts. The source code of all smart contracts used is in [33]. Besides, sequence diagrams and detailed instructions to perform each test cases are added to the appendix **??**.

6.1 The testing setup

To test SmartTool, several smart contracts were developed in Solidity and tested using Remix [34] to simulate their behavior on the Ethereum blockchain. Three types of attacks are studied: *(1) Single function reentrancy, (2) Cross function reentrancy and (3) Constantinople reentrancy*. For the first two cases, the *FundsManagerWithSmartTool* and *FundsManager* contracts are the same but they are attacked by two different smart contracts. The *Attacker* smart contract performs a single reentrancy attack and both the *Attacker1* and *Attacker2* perform a Cross-Function reentrancy in the second case. In the third case, the same smart contracts *PaymentSharer* & *Attacker* presented by ChainSecurity in [35] are used. To prove that SmartTool can stop this type of attack, the *PaymentSharerWithSmartTool* has been created as well.

In each case, two test cases are tested. The first one is to attack without SmartTool and the second one is to attack with SmartTool to see if the solution is able to prevent the reentrancy attempt. The detailed sequence of actions and UML sequence diagrams of each test case are described in the appendix **??**.

In the following subsections, an overview of the balance of each smart contract before and after performing the attack is shown in tables. Then an explanation on how the code can manage to detect reentrancy attacks. In these tables, the stages are numbered and mean the following:

1. Stage 1 is the balance of the smart contracts after deployment on the blockchain.
2. Stage 2 is the balance of the smart contracts before the attack.
3. Stage 3 is the balance of the smart contracts after the attack.

6.2 Single Function reentrancy case

First test case: The *Attacker* smart contract performs single function reentrancy on the *FundsManager* which contains a vulnerability in line 22. The attack is performed in three stages and the balances of each smart contract is presented in table1.

Stage 1:
- Deploy the *Attacker* smart contract with 1 wei.
- Deploy the *FundsManager* smart contract with 2 wei.

Stage 2:
- Deposit 1 wei from the *Attacker* smart contract to *FundsManager*.

Stage 3:
- Call the *withdraw(1)* function from *Attacker* smart contract to perform the Single Function reentrancy attack.
- The *Attacker* receives all the funds of the *FundsManager* because of the recursive calls hidden in the *fallback()* function of *Attacker*.

Stage	Attacker{}	FundsManager{}
1	1	2
2	0	3
3	3	0

Table 1: Balance of smart contracts after all three stages of the first case of Single function reentrancy. All balances are in: wei.

Result: The single function reentrancy attack succeeded in stealing all funds from the *FundsManager* in this test case.

Second test case: The *Attacker* smart contract performs a single function reentrancy on the *FundsManagerWithSmartTool* which contains a vulnerability in line 71. The attack is performed in three stages and the balances of each smart contract is presented in table2.

Stage 1:
- Deploy the *Attacker* smart contract with 1 wei.
- Deploy the *FundsManagerWithSmartTool* smart contract with 2 wei.

Stage 2:
- Deposit 1 wei from the *Attacker* smart contract to *FundsManagerWithSmartTool*.
- The *FundsManagerWithSmartTool* stores the funds in the *SmartWallet* associated with the *Attacker*.

Stage 3:
- Call the *withdraw(1)* function from *Attacker* smart contract to perform the Single Function reentrancy attack.
- The *Attacker* receives only the funds deposited in the second stage from the *FundsManagerWithSmartTool*.
- The *FundsManagerWithSmartTool* stores the address of the *Attacker1* to be banned.

Stage	Attacker{}	FundsManager WithSmartTool{}	SmartWallet{}
1	1	2	0
2	0	2	1
3	1	2	0

Table 2: Balance of smart contracts after all three stages of the second case of Single function reentrancy. All balances are in: wei.

Result: The single function reentrancy attack failed in the second test case even if *FundsManagerWithSmartTool* contract has a reentrancy vulnerability.

6.3 Cross Function reentrancy case

First test case: The *Attacker1* smart contract performs a cross function reentrancy with the help of the *Attacker2* contract on the *FundsManager* which contains a vulnerability in line 22. The attack is performed in three stages and the balances of each smart contract is presented in table3.

Stage 1:
- Deploy the *Attacker1* smart contract with 1 wei.
- Deploy the *FundsManager* smart contract with 2 wei.
- Deploy the *Attacker2* smart contract with 0 wei.

Stage 2:
- Deposit 1 wei from the *Attacker* smart contract to *FundsManager*.

Stage 3:
- Call the *withdraw(1)* function from *Attacker* smart contract to perform the Cross Function reentrancy attack.
- The *Attacker1* 1 wei from the *FundsManager*.
- The *Attacker2* receives the rest of the funds of *FundsManager*

Result: The cross function reentrancy attack succeeded in stealing all funds from the *FundsManager* in this test case.

Stage	Attacker1{}	FundsManager{}	Attacker2{}
1	1	2	0
2	0	3	0
3	1	0	2

Table 3: Balance of smart contracts after all three stages of the first case of Cross-Function reentrancy. All balances are in: wei.

Second test case: The *Attacker1* smart contract performs a cross function reentrancy with the help of the *Attacker2* contract on the *FundsManagerWithSmartTool* which contains a vulnerability in line 71. The attack is performed in three stages and the balances of each smart contract is presented in table4.

Stage 1:
- Deploy the *Attacker1* smart contract with 1 wei.
- Deploy the *FundsManagerWithSmartTool* smart contract with 2 wei.
- Deploy the *Attacker2* smart contract with 0 wei.

Stage 2:
- Deposit 1 wei from the *Attacker* smart contract to *FundsManagerWithSmartTool*.
- The *FundsManagerWithSmartTool* stores the funds in the *SmartWallet* associated with the *Attacker1*.

Stage 3:
- Call the *withdraw(1)* function from *Attacker* smart contract to perform the Cross Function reentrancy attack.
- The *Attacker1* 1 wei from the *FundsManager*.
- The *FundsManagerWithSmartTool* stores the address of the *Attacker1* to be banned.

Stage	Attacker1{}	FundsManager WithSmartTool{}	Attacker2{}	SmartWallet{}
1	1	2	0	0
2	0	2	0	1
3	1	2	0	0

Table 4: Balance of smart contracts after all three stages of the second case of Cross-Function reentrancy. All balances are in: wei.

Result: The cross function reentrancy attack failed in the second test case even if *FundsManagerWithSmartTool* contract has a reentrancy vulnerability.

6.4 Constantinople reentrancy case

To prove that SmartTool can detect this type of reentrancy, the same example used by ChainSecurity to prove that the lowering of gas costs can enable this reentrnacy is used in this proof as well [35]. To simulate the same state of the blockchain at the time of the Constantinople update, we use Ganache [36] at the ”constantinople” hardforfk. This enables us to run the attack with low gas costs.

First test case: The *Attacker3* smart contract performs the attack on the *PaymentSharer* smart contract which is vulnerable to Constantinople reentrancy attacks. The attack is performed in three stages and the balances of each smart contract is presented in table5.

Stage 1:
- Deploy the *Attacker3* smart contract with 1 wei.
- Deploy the *PaymentSharer* smart contract with 1 wei.
- Deploy the *SecondaryAttacker* smart contract with 0 wei.

Stage 2:
- Deposit 1 wei from the *Attacker3* smart contract to *PaymentSharer*.
- The *Attacker3* creates a shared account between itself and *SecondaryAttacker* with 100% of the funds to itself.

Stage 3:
- Call the *attack()* function from *Attacker3* smart contract to perform the Constantinople reentrancy attack.
- The *Attacker1* 1 wei from the *PaymentSharer*.
- The split of the shared account gets changed to 100% to the *SecondaryAttacker*.
- The *Attacker2* receives 1 wei from the *PaymentSharer*.

Stage	Attacker3{}	PaymentSharer{}	SecondaryAttacker{}
1	1	1	0
2	0	2	0
3	1	0	1

Table 5: Balance of smart contracts after all three stages of the first case of Constantinople reentrancy. All balances are in: wei.

Result: The Constantinople reentrancy succeeded in stealing funds because the *Attacker3* contract updated the splits before updating the internal state of *PaymentSharer*.

Second test case: The *Attacker3* smart contract performs the attack on the *PaymentSharerWithSolution* smart contract which is vulnerable to Constantinople reentrancy attacks. The attack is performed in three stages and the balances of each smart contract is presented in table6.

Stage 1:
- Deploy the *Attacker3* smart contract with 1 wei.
- Deploy the *PaymentSharerWithSolution* smart contract with 1 wei.
- Deploy the *SecondaryAttacker* smart contract with 0 wei.

Stage 2:
- Deposit 1 wei from the *Attacker3* smart contract to *PaymentSharerWithSolution*.
-
- The *PaymentSharerWithSolution* stores the funds in the *SmartWallet* associated with the *Attacker3*.
- The *Attacker3* creates a shared account between itself and *SecondaryAttacker* with 100% of the funds to itself.

Stage 3:
- Call the *attack()* function from *Attacker3* smart contract to perform the Constantinople reentrancy attack.
- The *Attacker3* 1 wei from the *PaymentSharer*.
- The *PaymentSharer* stores the address of the *Attacker3* to be banned.

Stage	Attacker3{}	SmartSharer WithSmartTool{}	SecondaryAttacker{}	SmartWallet{}
1	1	1	0	0
2	0	1	0	1
3	1	1	0	0

Table 6: Balance of smart contracts during the second case of Constantinople reentrancy. All balances are in: wei.

Result: The Constantinople reentrancy was stopped by the *PaymentSharerWithSolution* before stealing funds because the funds of the shared account were all stored in the helper Wallet and it was empty after the first lawful withdrawal by the attacker.

7 Discussion

The test cases conducted in this paper have proven that the *SmartTool* can detect and stop single function, Cross-Function and Constantinople reentrancy attacks. Since the third type of attack was a novel one, most tools that rely on patterns for reentrancy attacks detection didn't have prior knowledge that a reentrancy attack can be performed in a similar way and thus couldn't stop it.

SmartTool can protect smart contracts that are already deployed on the Ethereum smart contract unlike most of the other tools presented in the related work section 4. It can also be implemented in the vulnerable smart contract or in an external one. Moreover, it can be implemented in languages other than Solidity. What is also important to note here is that due to SmartTool's main idea that exploits the fact that the protocol layer balance and the balance of users stored on the application layer are both consistent before and after each transaction, it can potentially detect and stop new types of reentrancy attacks without prior knowledge of how the attack is performed.

Even though, SmartTool can detect reentrancy attacks and stop them during execution. It comes with some limitations as well depending on the implementation approach done by the developers.

- *Implementation on the smart contract itself or on a different smart contract* requires access to the code which is not always possible since the smart contracts can be immutable on the Ethereum blockchain. It also requires a high amount of gas to perform all the necessary actions.
- *Implementation of an off-chain tool* is a relatively centralized approach since a third party will be responsible of checking if the transaction is a reentrancy or not. This goes against the goal of DeFi systems.

A future version of the SmartTool can be focused on limiting the gas cost of running the tool on-chain.

8 Responsible Research

In this section, the ethical implications and reproducibility of this research are discussed.

In this research, two main ethical implications exists. First, the proposed tool can detect reentrancy attacks on smart contract of Decentralized Finance systems on the Ehtereum blockchain. Therefore, all the contracts used to prove the efficacy of this tool were made up and do not have any relations with existing tools on the Ethereum blockchain. The choice to not use real smart contract that are already deployed on the Ethereum blockchain is to avoid putting users' funds put on the smart contracts in case they were found to be vulnerable to reentrancy attack.

The second implication is that the proposed tool can achieve two purposes. Detect possible smart contracts and protect vulnerable smart contracts on the blockchain. Therefore, it can be used by anyone to test if the smart contract of a DeFi system is vulnerable to reentrancy attacks. Thus, it can help an attacker to chose which smart contract to target. However, this tool was created with the purpose to, first, help creators and auditors of the smart contract to protect their users, and to provide the users of Decentralized Finance the ability to test the smart contracts themselves before interacting with them. That is why we ask creators/auditors of DeFi systems to use this tool to help the Ethereum community to prevent future reentrancy attacks.

It is also important to discuss the reproducibility of the results of this paper. The research has presented a new tool to protect and prevent reentrancy attacks on smart contract already deployed on the Ethereum blockchain. A detailed proof of the efficacy of this tool is presented in section 6. The source code of the tool and the contracts used in the proofs are publicly available on GitHub for any researcher to verify the results. Moreover, the workflow of the idea behind the tool is presented in details in section 5. This can help any skilled researcher to build a similar tool and verify if the idea works and is sound.

9 Conclusions and Future Work

This paper has provided a new solution (SmartTool) to detect and stop reentrancy attacks on smart contracts already deployed on the Ethereum blockchain. Its main idea is to check if the difference between the balance of the smart contract on the protocol layer and the balance of participants stored on the application layer should stay the same before and after each transaction. This allows SmartTool to detect reentrancy attacks during the execution of the smart contract without having prior knowledge of the mechanism of the attack. Meaning that it, unlike current tools and methods that use patterns, can stop novel types of reentrancy attacks in the future as well. The correctness of this tool was proven by implementing the tool on vulnerable smart contract and performing 3 different types of reentrancy attacks.

It is also important to highlight that most existing tools used to protect smart contracts against reentrancy attacks can only detect them before deployment. On the other hand, SmartTool can detect reentrancy attacks after deployment and thus can be used by auditors and members of the DeFi community to test public smart contracts on the Ethereum blockchain.

However, this paper only tackled this problem by implementing the tool on the smart contract directly which will drive the gas costs up. Further research on whether it can be implemented off-chain to limit the gas cost is needed. Another possible future study can focus on whether detection of reentrancy attack can be implemented directly on the protocol layer which will make the miners check the transactions and decide if it is a legitimate one or not instead of relying on developers to protect each smart contract individually.

There exist many other challenges when it comes to the security of smart contracts of DeFi systems. Since any small mistake on the developers side can cause the loss of huge amounts of funds. That is why more research on the security of smart contracts by the industry and academia is needed.

References

[1] "Defi pulse: The defi leaderboard: Stats, charts and guides," https://defipulse.com/, 2020.

[2] D. Siegel, O. Godbole, D. Palmer, D. Nelson, D. Dantes, and C. Kim, "The dao attack: Understanding what happened," https://www.coindesk.com/understanding-dao-hack-journalists, Dec 2020.

[3] "Ethereum white paper," https://ethereum.org/en/, 2019.

[4] C. Ferreira Torres, M. Baden, R. Norvill, B. B. Fiz Pontiveros, H. Jonker, and S. Mauw, "Ægis: Shielding vulnerable smart contracts against attacks," *Proceedings of the 15th ACM Asia Conference on Computer and CommunicationsSecurity*, 2020.

[5] ChainSecurity, "Constantinople enables new reentrancy attack," https://medium.com/chainsecurity/constantinople-enables-new-reentrancy-attack-ace4088297d9, Jan 2019.

[6] M. Coblenz, "Obsidian: A safer blockchain programming language," *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[8] 2021. [Online]. Available: https://defihacks.wiki/

[9] "Solidity documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.8.4/

[10] "What is ether (eth)?" [Online]. Available: https://ethereum.org/en/eth/

[11] "Withdrawal from contracts," 2021. [Online]. Available: https://docs.soliditylang.org/en/develop/common-patterns.html#withdrawal-from-contracts

[12] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[13] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," *Lecture Notes in Computer Science*, p. 164–186, 2017.

[14] W. Shahda, "Protect your solidity smart contracts from reentrancy attacks," Aug 2020. [Online]. Available: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

[15] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[16] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[17] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[18] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[19] "Welcome to mythril's documentation!" 2019. [Online]. Available: https://mythril-classic.readthedocs.io/en/master/

[20] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard," *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018.

[21] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[22] V. Wüstholz and M. Christakis, "Harvey: a greybox fuzzer for smart contracts," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[23] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[24] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck," *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[25] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 1–28, 2018.

[26] ConsenSys, "Known attacks." [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/

[27] OpenZeppelin, "Reentrancyguard," Feb 2021. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol

[28] "Vyper documentation." [Online]. Available: https://vyper.readthedocs.io/en/latest/index.html

[29] M. Coblenz, "Obsidian: A safer blockchain programming language," *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[30] A. Tyurin, I. Tyuluandin, V. Maltsev, I. Kirilenko, and D. Berezun, "Overview of the languages for safe smart contract programming," *Proceedings of the Institute for System Programming of the RAS*, vol. 31, pp. 157–176, 09 2019.

[31] G. Bessler, J. Cordova, S. Cullen-Baratloo, S. Dissem, E. Lu, S. Devin, I. Abugharah, and L. Bang, "Metrinome: Path complexity predicts symbolic execution path explosion," *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021.

[32] A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, "A mechanism to detect and prevent ethereum blockchain smart contract reentrancy attacks," *Frontiers in Computer Science*, vol. 3, p. 1, 2021. [Online]. Available: https://www.frontiersin.org/article/10.3389/fcomp.2021.598780

[33] N. El Coudi El Amrani, "Smarttool: Source code," 2021. [Online]. Available: https://github.com/NafieAmrani/SmartTool

[34] "Remix ide." [Online]. Available: https://remix.ethereum.org/

[35] ChainSecurity, "Chainsecurity/constantinople-reentrancy." [Online]. Available: https://github.com/ChainSecurity/constantinople-reentrancy

[36] T. Suite, "Ganache: Overview: Documentation." [Online]. Available: https://www.trufflesuite.com/docs/ganache/overview