Secure smart contract coding

Michal Ľaš
Brno University of Technology
Faculty of Information Technology
Brno, Czech Republic
xlasmi00@stud.fit.vutbr.cz

***Abstract*—This paper aims to provide an overview of common vulnerabilities in smart contracts and the tools available to improve their security. Smart contracts are a crucial part of decentralized finance (DeFi), but their vulnerabilities can lead to significant financial losses, which makes them unreliable to manage valuable assets. This paper highlights the need to make smart contracts more secure and reliable by providing statistics about crypto crimes, discussing various vulnerabilities, and examining tools to enhance security. By understanding these risks and tools, developers can build safer smart contracts and contribute to a more secure DeFi ecosystem.**

***Index Terms*—smart contracts, decentralized finance, security vulnerabilities, DeFi hacking**

I. Introduction

Recent years have shown the increasing popularity of decentralized cryptocurrencies built upon blockchain technology. This surge has prompted the rapid development of decentralized financial applications, commonly referred to as 'DeFi'. DeFi are open financial applications composed of smart contracts, deployed on publicly accessible, permissionless blockchains [1]. A smart contract, first described by Nick Szabo in 1997, is a set of promises, specified in digital form, including protocols within which the parties fulfill on these promises [2]. In other words, it is executable code that formalizes agreements and their associated conditions [3]. Figure 1 illustrates the interaction between smart contracts and blockchains.

For instance, an Ethereum smart contract consists of executable code, a state consisting of private storage, contract address, and balance (cryptocurrency). A smart contract can be invoked using a contract invoking transactions to its unique address. The actions are then recorded back to the blockchain [4].

Blockchains provide a trusted environment for smart contracts. Blockchain technology became popular mainly because of the popular cryptocurrency Bitcoin, which idea was described by Satoshi Nakamoto in the original paper [5] in 2008. It is a distributed ledger of cryptographically signed transactions grouped in blocks. Communication within the blockchain occurs through a peer-to-peer (P2P) network. Blockchain technology contains its own key features such as security, transparency, decentralization, and immutability [6].

The fact that a smart contract is an executable code also means that there can be vulnerabilities in the code that can be used by potential attackers to steal funds or violate the system's integrity. During the year 2023, $1,1 billion was stolen in DeFi hacks. The year 2022 was even worse with $3,1 billion stolen [7]. These figures underscore the critical need for robust security measures, given that DeFi is intended to be a reliable, trusted, and secure ecosystem. Approximately half of these losses stemmed from vulnerabilities in smart contracts, and the growing value locked in DeFi platforms is likely to attract an increasing number of attackers.

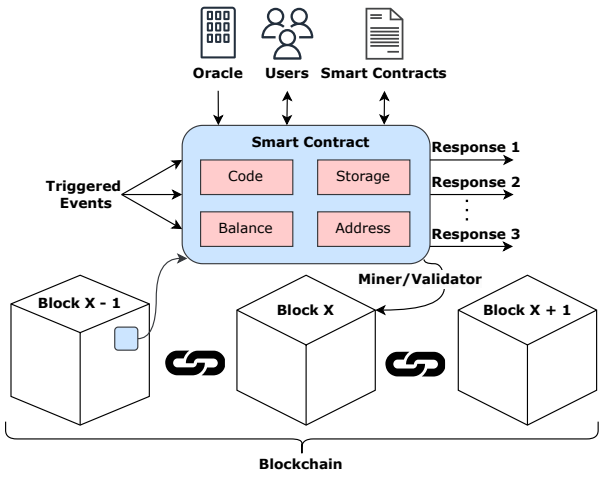


Fig. 1. Example of a smart contract in a blockchain [4].

To sustain the growth and trust in DeFi, it is important to enhance the security of smart contracts. However, this task may be challenging due to the inherent properties of blockchains, such as integrity and transparency. The integrity of blockchains makes it impossible for a smart contract to be changed or deleted. Therefore, security vulnerabilities and best practices have to be considered during the coding of smart contracts [4]. Furthermore, the transparency of blockchains allows everybody to see the code of a smart contract, making it easier for attackers to find a vulnerability [8].

The rest of this paper is organized as follows. Section II provides brief statistics about crypto crimes and an overview of examples of successful attack events. Section III categorizes and discusses the various types of smart contract vulnerabilities. Section IV introduces methods and tools designed to improve smart contract security. Finally, Section V concludes the paper.

II. Crypto crimes statistics

This section demonstrates the impacts of vulnerabilities in DeFi and highlights the importance of smart contract security. It provides statistics on DeFi-related crypto crimes, historical and future trends, and presents brief examples of significant attacks resulting in substantial financial losses. There are outlined more possible attack vectors within the DeFi landscape, but the main focus is on attacks related to smart contracts. The data presented here are sourced primarily from Chainalysis, a blockchain analytics platform specializing in the investigation of blockchain-based criminal activity. Key references include the Chainalysis *The 2024 Crypto Crime Report* [7], and *2024 Crypto Crime Mid-Year Update Part 1* [9]. There is also *2024 Crypto Crime Mid-Year Update Part 2* [10], but it is focused on scams and increasing cases of on-chain activity related to the distribution and consumption of child sexual abuse material. These topics are beyond the scope of this paper.

A. Value stolen in DeFi hacks

The DeFi hacking exploded in 2021 and the trend continues even in 2022. However, 2023 experienced a notable 63,7% reduction in the total value stolen, even though the number of hacks remained relatively consistent. This drop may be caused by multiple factors like better application of security practices, but also because of the overall drop in DeFi activity in 2023 [7]. These data are shown in Figure 2.

The *2024 Mid-Year Update* indicates a rise in attack activity compared to 2023. However, attackers have shifted their focus from DeFi protocols to centralized exchanges, often employing ransomware attacks. This is probably due to the increasing value of Bitcoin as DeFi's are less popular for trading Bitcoin [9]. However, in the last quarter of 2024, the crypto market is rising rapidly including currencies like Solana and other Ethereum virtual Machine-based chains which are the most targeted chains by DeFi attacks due to their popularity and capability to execute smart contracts [7].

B. Attack vectors within the DeFi landscape

DeFi is subject to a wide range of attack vectors, which continue to evolve. From a high level, they can be divided into two types – on-chain and off-chain attack vectors. On-chain attacks stem from vulnerabilities in the components of the DeFi protocol, such as their smart contract, but not from vulnerabilities inherent to blockchains themselves. On the other hand, off-chain attack vectors stem from vulnerabilities outside of blockchain. For example, off-chain storage of private keys, a faulty cloud storage solution, etc [11].

Figure 3 illustrates the proportion of value stolen by various attack vectors in 2023. This graph shows that around 61,3% of value was stolen by on-chain attacks related to smart contract vulnerabilities – **Price manipulation hack** and **Smart contract exploitation**. Further in this paper, the main focus will be specifically on these attacks. For context, brief definitions of all attack vectors are provided below.

Price manipulation hack refers to an attack where an attacker exploits a vulnerability in a smart contract or leverages a flawed oracle, one that fails to provide accurate asset valuations, to manipulate the price of a digital token. **Smart contract exploitation** occurs when an attacker leverages a weakness in the smart contract's code, typically gaining direct access to various protocol control mechanisms and the ability to transfer tokens. **Governance attacks** take place when an adversary influences a blockchain project governed by decentralized structures, securing sufficient voting rights or influence to pass malicious proposals. **Compromised private key** describes an attack in which an adversary gains access to a user's private key, potentially through a leak or an off-chain software failure. **Phishing** attacks involve an adversary deceiving users into granting permissions, often by impersonating a legitimate protocol, thereby enabling unauthorized token spending on the user's behalf. Phishing can also occur when an attacker tricks users into directly transferring funds to malicious smart contracts. **Contagion** refers to an exploit where an attacker compromises a protocol due to vulnerabilities introduced by a hack in another protocol. This term also encompasses attacks closely connected to breaches in other protocols [11].

C. Example of attack events

So far in 2024, the largest hacks have targeted centralized crypto exchanges. On May 31, 2024, DMM Bitcoin, a prominent Japanese cryptocurrency exchange, suffered a significant security breach, resulting in the unauthorized transfer of 4 502.9 Bitcoin. While DMM Bitcoin has not disclosed specific details about the breach, it has been attributed to the Lazarus Group, a well-known hacker organization from North Korea [12].

One of the most infamous attacks in blockchain history remains the Decentralized Autonomous Organization (DAO) attack of 2016. The DAO was an innovative project launched on the Ethereum blockchain, designed to function as a decentralized venture capital fund where participants could invest and vote using smart contracts. However, the project was compromised due to a critical vulnerability in its smart contract code, leading to a significant theft of funds. This incident ultimately resulted in a hard fork of the Ethereum blockchain [8].

Another notable attack is the Fomo3D exploit. Fomo3D was an Ethereum-based game in which participants purchased keys using Ether and could earn additional Ether by referring others to the game. The objective was to be the last participant to purchase a key before the timer expired, winning the entire pot of Ether. The attacker exploited this by purchasing a ticket and sending multiple high-gas-price transactions in rapid succession. This strategy consumed a significant portion of the block's gas limit, delaying or indefinitely stalling other transactions related to Fomo3D, including key purchases from other participants. As a result, the timer ran out, which secured the prize for the attacker [8].

These are just three examples of the many possible attack strategies. For more recent examples, refer to the article [12] or the Halborn blog [11].

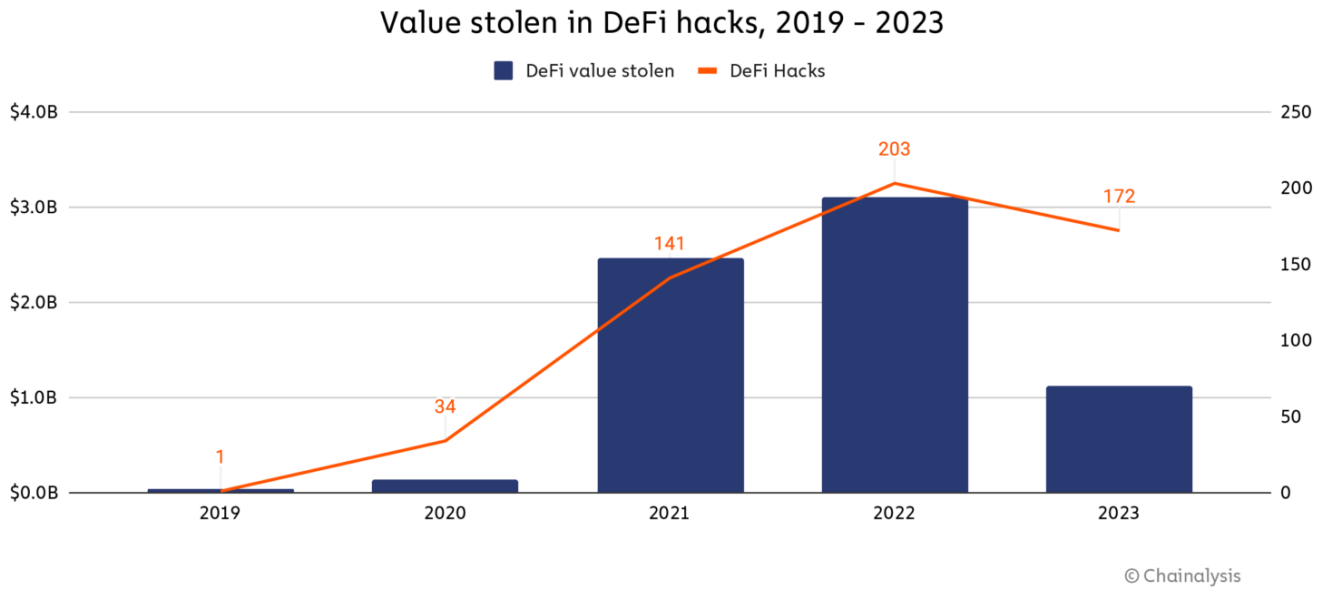
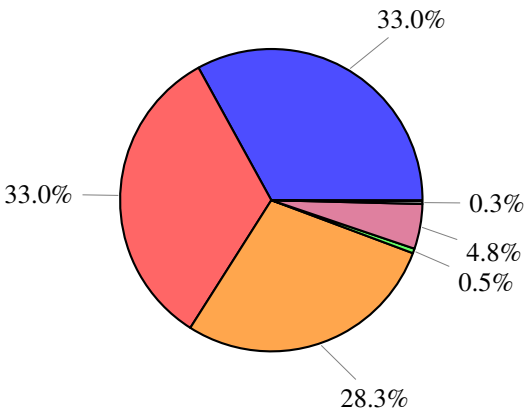


Fig. 2. Value stolen in DeFi hacks [7].



Color	Attack Vector and Share
	Price manipulation hack (on-chain) - 33.0%
	Compromised private key (off-chain) - 33.0%
	Smart contract exploitation (on-chain) - 28.3%
	Contagion (on-chain) - 4.8%
	Governance attack (on-chain) - 0.5%
	Phishing (off-chain) - 0.3%

Fig. 3. Yearly share of value stolen in DeFi hacks by attack vector, 2023 [7]

III. Smart Contracts Vulnerabilities

This section discusses the types, classifications, and sources of code examples related to vulnerabilities in smart contracts. Despite numerous studies, no unified classification of vulnerabilities exists. Atzei et al. [13] categorized 12 vulnerabilities in three categories – Solidity, Ethereum Virtual Machine (EVM) bytecode, and blockchain. Kushwaha et al. [4] followed up this categorization but defined 23 vulnerabilities. Rameder et al. [14] extracted 54 vulnerabilities and assigned them to 10 categories based on 17 systematically selected surveys. Wei et al. [8] propose a novel classification with four categories, 14 subcategories, and 43 vulnerabilities. This classification enhances the understanding of the underlying causes of vulnerabilities and facilitates more effective categorization.

This categorization is illustrated in Table I. A filled circle (●) indicates the vulnerability has been resolved, an empty circle (○) indicates it remains unresolved but widely discussed, and a half-filled circle (◐) signifies partial resolution through specific tools or approaches. However, detailed explanations focus only on the most common vulnerabilities responsible for the majority of incidents. For a full breakdown of categories, subcategories, and vulnerabilities, refer to the original papers.

A. Most common vulnerabilities

The ten most prevalent vulnerabilities, as identified by Wei et al. [8] are: *reentrancy*, *arithmetic overflow/underflow*, *DoS with block gas limit*, *unsafe suicidal*, *unsafe delegatecall*, *unchecked send*, *transaction order dependency*, *time manipulation*, and *authorization through tx.origin*.

1) Reentrancy attack: It is also known as a recursive call attack. This vulnerability arises when an external callee contract calls back to a function in the caller contract before the caller contract finishes. This creates a loop that runs until the caller contract is drained or the transaction runs out of gas. This vulnerability was behind the famous DAO attack in 2016 [4].

TABLE I
CATEGORIZATION OF SMART CONTRACT VULNERABILITIES [8]

Improper Adherence to Coding Standards	
Syntax Error	
Typographical Error	●
Right-To-Left-Override control character	●
Version Issue	
Floating Pragma	●
Use of Deprecated Functions	●
Incorrect Constructor Name	●
Uninitialized Storage Pointer	●
Irrelevant Code	
Presence of Unused Variables	●
Code with No Effects	●
Shadowing State Variables	●
Visibility	
Function Default Visibility	●
State Variable Default Visibility	●
Insufficient Verification of Data Authenticity	
Signature Issue	
Signature Malleability	○
Missing Protection against Signature Replay Attack	○
Lack of Propper Signature Verification	○
Data Issue	
Unencrypted Private Data On-Chain	○
Fake EOS	◐
Forged Notification, Fake Receipt	◐
Improper Access Control	
Unprotected Low-level Function	
Unsafe Suicide	◐
Authorization through `tx.origin`	◐
Unsafe Delegatecall	◐
Coding Issue	
Unprotected Ether Withdrawal	◐
Write to Arbitrary Storage Location	●
Insufficient Control Flow Management	
Input Issue	
Assert Violation	●
Requirement Violation	●
Wrong Address	◐
Incorrect Calculation	
Arithmetic Overflow/Underflow	◐
Call-Stack Overflow	●
Asset Overflow	◐
Denial of Service	
Dos with Failed Call	◐
Insufficient Gas Griefing	◐
DoS with Block Gas Limit	◐
Use of Low-level Function	
Unchecked Send	◐
Arbitrary Jump with Function Type Variable	◐
Hash Collisions	◐
Message Call with Hardcoded Gas Amount	●
Behavioral Workflow	
Reentrancy	◐
Unexpected Ether Balance	◐
Incorrect Inheritance Order	○
Infinite Loop	◐
Consensus Issue	
Transaction Order Dependence	◐
Time Manipulation	◐
Bad Randomness	◐

An example of a reentrancy attack is in Listing 1. Bob sends Ether and then sets `sent` to `true`. However, before `sent` is updated, Mallory's fallback function can re-enter Bob and call `ping` again, allowing repeated calls to withdraw Ether.

As a preventive method against a reentrancy attack, it has

```
1  pragma solidity 0.4.16;
2
3  contract Bob {
4      bool sent = false;
5      function ping( address c ) {
6          if (!sent) {
7              // Sends 2 wei to `c` and executes
                   its fallback function
8              c.call.value(2)();
9              // Sets `sent` to true after making
                   the call
10             sent = true;
11 }}}
12
13 // ------------------------------- //
14
15 contract Bob { function ping (); }
16
17 contract Mallory {
18     function() {
19         // Calls `ping` on the sender contract (
               assumed to be `Bob`)
20         Bob (msg.sender).ping(this);
21 }}
```

Listing 1. Example of reentrancy attack [13].

to be assured that state variables are updated **before** calling another contract. There should be a mutex lock on the contract state to ensure that only the lock owner can change the state and limit the currency amount of possible transfer between contracts [15].

2) Arithmetic overflow/underflow: Integer overflow or underflow is a common vulnerability not only in smart contracts. It is a vulnerability when there is a computation operation on an integer variable in a statement or expression and the developer does not pay attention to the bounding value of the variable. For instance, exceeding the maximum value of a 32-bit unsigned integer (2^{32}) causes the value to wrap around to zero. Attackers can exploit this to bypass critical checks, such as balance or transfer limits [16].

A good prevention method is to do additional checks with *assertions* or *require* and use tested libraries for safe arithmetic functions [4].

3) DoS with block gas limit: This attack is also known as *DoS with Unbounded Operations* and it is related to smart contracts that are running on platforms using gas or other type of limited unit of computational effort. This vulnerability occurs when the amount of gas required for executing a contract exceeds the block gas limit. The block gas limit is the maximum amount of gas that can be used in a single block. If the block gas limit is exceeded then other transactions will fail. This can happen if the smart contract contains a cycle through a dynamically sized data structure [15].

An example of a DoS with a block gas limit attack is in Listing 2. The loop iterates `_amount` times. For large `_amount`, the gas cost exceeds the block gas limit, causing the transaction to fail.

One possible solution is to avoid iterating through data structures with a dynamic number of items. If such an iteration is necessary, it is crucial to ensure that some limit is ensured,

```
pragma solidity ^0.8.0;

contract TokenTransfer {
    mapping(address => uint256) public balances;

    function transfer(address _to, uint256
        _amount) public {
        require(balances[msg.sender] >= _amount,
            "Insufficient balance");

        // The loop iterates _amount times, which
             can be very inefficient and can
            potentially exceed the block gas
            limit if _amount is too large.
        for (uint256 i = 0; i < _amount; i++) {
            balances[msg.sender]--;
            balances[_to]++;
}}}
```

Listing 2. Example of DoS with block gas limit attack [17].

that will not be exceeded, and to divide large transactions into smaller ones in several blocks [4].

4) Unsafe suicidal: Blockchains are immutable, but there should be a mechanism that can deactivate a smart contract. Solidity provides a destruction function that requires the user to introduce a suicide function when developing a contract. Calling this function destroys the contract and transfers the rest of the currency allocated by the contract to the specified address. However, the suicidal function has to have permission control so it can not be called by unauthorized users to kill the contract and change the address to return the remaining cryptocurrency [16]. Additionally, if the address for the remaining currency return is on a different smart contract that was already destroyed the remaining currency is lost forever [8].

5) Unsafe delegatecall: The delegation function allows one contract to call another contract, which will be executed, using the original contract's storage. In this way, it is possible to create modular smart contracts. However, a vulnerability may occur if the called contract is unsafe or compromised. It can lead to unexpected execution such as self-destruction or balance loss [4].

To prevent this vulnerability Delegate calls must be used to call trusted contracts only and the return of delegetian has to be always checked for exceptions [4].

6) Unchecked send: This vulnerability is also known as *unhandled exceptions* or *unchecked low-level call* and it is a common vulnerability not only in smart contracts. Typically this vulnerability occurs when a developer forgets to check the return of a function or assumes a function never fails [8]. If an attacker manages to create an unhandled exception the subsequent operation will continue, resulting in a method that is not implemented as expected by the developer. Therefore always check for a possible exception [16].

7) Transaction order dependency: This vulnerability is connected to the way how transactions are selected and ordered in individual blocks. Typically, the selection of transactions in the block is determined by validators/miners. They usually select the transactions with the highest reward for processing the transaction. This system can be abused if a potential attacker is interested in having his transaction processed preferentially, so he can increase the reward for processing his transaction and thus affect the smart contract, which is not protected against such actions [16]. For example[1], in the case of an online auction, an attacker can follow the bids of others and, with his bid shortly before the end of the auction, surpass the bids of others and unfairly win the auction. In this case, the protection could be that the last offers would be hidden (e.g. hashed) and revealed only after a certain time.

8) Time manipulation: Smart contracts can access the block timestamp in which the transaction is placed. However, this can be dangerous because block timestamps can be manipulated by validators/miners. So if an attacker is also a validator/miner he can gain an advantage by choosing a suitable timestamp for a block. The solution is to use the block number rather than the block timestamp because the block number can not be manipulated [13].

9) Authorization through tx.origin: This vulnerability is specific to Solidity smart contracts. `tx.origin` is a global variable and refers to the transaction initiator. The vulnerability occurs when `tx.origin` is used for authorization. An attacker can create a smart contract that calls the vulnerable contract. If the attacker tricks the author of the vulnerable contract into calling the attacker's smart contract the `tx.origin` will still be set to the author of the vulnerable contract and the attacker is authorized.

An example of an authorization through tx.origin attack is in Listing 3. If the owner of `MyContract` calls `Exploiter.exploit()`, the check `tx.origin == owner` passes (as the owner initiated the transaction), allowing unauthorized Ether transfers.

This vulnerability can be prevented by using `msg.sender`, instead of `tx.origin`, for authentication, because `msg.sender` returns the account that incurred the message [15].

B. Vulnerabilities code examples

One of the biggest advantages when it comes to learning about smart contract vulnerabilities is that all code is transparent and accessible. Many projects collect code examples with smart contract vulnerabilities and this material can be used for studying, analysis, testing, and machine learning. For addressing the top vulnerabilities that should be avoided there are the Smart Contract Weakness Classification (SWC) [18], OWASP Smart Contract Top 10 [17], and Decentralized Application Security Project (DASP) Top 10 [19]. SWC and DASP are some older projects, but they are still relevant. A great checklist for smart contract developers, architects, security reviewers, and vendors is provided by D. Rusinek and P. Kuryłowicz [20]. A good starting material is a project by D. Muhs [21] with a guide for smart contract developers and ethical hackers. The notable project is also **DeFiHackLabs**[2]

[1]The example was inspired by https://github.com/obheda12/Solidity-Security-Compendium/blob/main/days/day1.md

[2]https://github.com/SunWeb3Sec/DeFiHackLabs

```
pragma solidity 0.4.24;

contract MyContract {
    address owner;
    function MyContract() public {
        owner = msg.sender;
    }
    function sendTo(address receiver, uint amount
        ) public {
        require(tx.origin == owner);
        receiver.transfer(amount);
}}

// ------------------------------ //

contract Exploiter {
    address payable public owner;
    Wallet wallet;
    constructor(Wallet _wallet) {
        wallet = Wallet(_wallet);
        owner = payable(msg.sender);
    }
    function exploit() public {
        // Exploiter's contract calls MyContract.
        wallet.transfer(owner, address(wallet).
            balance);
}}
```

Listing 3. Example of Authorization through `tx.origin` attack [18].

which currently contains a reproduction of 539 DeFi hacks incidents.

IV. Security analysis and enhancement tools

This Section provides an overview of available tools for smart contracts analysis, security enhancement, and their methodologies. Contract analyzers play a critical role because, as stated in the introduction, once a smart contract is deployed, it cannot be modified. So most of the tools target the contract analysis before deployment, although there are also tools for runtime monitoring. The methodologies and tools examples in this Section follow four surveys on this topic [8], [14], [22], [23]. These surveys declared almost the same common methodologies for smart contract analysis: formal verification, symbolic execution, fuzzing, intermediate representation, and machine learning. However, the number of available tools is too big to explain all of them. Therefore take the examples more like an overview and for better detail refer to the mentioned surveys.

A. Formal verification

Formal verification is a mathematical method used to construct and verify formal proofs, ensuring software adheres to its specifications and requirements. In smart contracts, two main approaches are used: model checking and theorem proving [8].

Model checking uses the property of high-level programming languages for writing smart contracts, like Solidity, and allows formal verification based on modeling the contract as a *finite state mechanism (FSMs)*. FSMs are simple formal objects and can be used to verify properties specified in variants of computation tree logic [14]. This approach is practical for smart contracts because they have a finite number of states and are designed to be deterministic [8]. Some available tools in this category are SIPN [24], FDR [25], or NuSMV [26].

Theorem proving on the other hand can be used for smart contracts with an infinite number of states. This approach leverages mathematical logic to define the desired properties of a system and utilizes a theorem prover to produce proofs that confirm these properties [8]. Available tools in this category are Coq [27] or FEther [28].

B. Symbolic execution

Symbolic execution systematically explores the possible execution paths of a smart contract by attempting all combinations of uncertain variables [22]. This approach identifies vulnerabilities that may arise under specific conditions and provides inputs that trigger these vulnerabilities. However, symbolic execution is not well-suited for large contracts with many execution paths. Oyente [29], the first tool for smart contract verification using control flow graphs, is a notable example of this methodology [8].

C. Fuzzing

Fuzzing is a technique that tests a smart contract with numerous randomized inputs to identify cases that cause exceptions or unexpected behavior [14]. This method is widely used due to its simplicity, effectiveness, and ease of implementation. Prominent tools in this category include ContractFuzzer [30] and ConFuzzius [31].

D. Intermediate representation

Some approaches explore analysis methods when a smart contract is converted into an intermediate representation (IR) with highly semantic information. IR allows easier analysis that can reveal vulnerabilities and security issues. This method is different from formal verification, IR relies on semantic-based transformation. There are two notable challenges semantic heterogeneity and high processing time compared to other methodologies [8]. A notable tool in this category is e.g. Slither [32].

E. Machine learning

Machine learning (ML) gained popularity across many different fields and also in smart contract vulnerability detection tools. Machine learning methods can extract features from smart contracts and train models for classifying smart contracts based on the types of vulnerabilities discovered in them. Most of the existing ML methods of smart contract threat mitigation use supervised learning, but there are also experiments with the utilization of large language models like ChatGPT for smart security enhancement. Under the research are also hybrid solutions that utilize ML and fuzzing. [8], [23].

V. Conclusion

This paper presented an overview of crypto crimes, common vulnerabilities in smart contracts, and available tools to mitigate them. It explained critical issues containing reentrancy attacks, arithmetic overflows/underflows, DoS with block gas

limit, unsafe suicidal, unsafe delegatecall, unchecked send, transaction order dependency, time manipulation, and authorization through tx.origin. It also covers effective tools for pre-deployment analysis and security enhancement like formal verification, symbolic execution, fuzzing, intermediate representation, and machine learning. The key takeaway is that the security of smart contracts is crucial for making DeFi usable and reliable. Using available tools along with following best practices, can help protect funds and ensure trust in decentralized systems.

Currently, the number of successful DeFi attacks is decreasing, and it seems that attackers have shifted their efforts to other areas. This can be caused by the improving security of smart contracts. According to the number of new studies, academic papers, and developed tools, this area is under intensive research and I believe that it brings even better security.

REFERENCES

[1] J. R. Jensen, V. von Wachter, and O. Ross, "An introduction to decentralized finance (defi)," *Complex Systems Informatics and Modeling Quarterly*, no. 26, pp. 46–54, April 2021.

[2] N. Szabo, "Smart contracts: Building blocks for digital markets," 1996, Accessed: 11.12.2024. [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[3] J. Liu and Z. Liu, "A survey on security verification of blockchain smart contracts," *IEEE Access*, vol. 7, pp. 77 894–77 904, 2019.

[4] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," October 2008, Accessed: 11.12.2024. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[6] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: Architecture, applications, and future trends," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 11, pp. 2266–2277, 2019.

[7] Chainalysis Team, "The 2024 crypto crime report," Chainalysis, February 2024, Accessed: 11.12.2024. [Online]. Available: https://go.chainalysis.com/crypto-crime-2024.html

[8] Z. Wei, J. Sun, Z. Zhang, X. Zhang, X. Yang, and L. Zhu, "Survey on quality assurance of smart contracts," *ACM Comput. Surv.*, vol. 57, no. 2, p. 36, October 2024. [Online]. Available: https://doi.org/10.1145/3695864

[9] Chainalysis Team, "2024 crypto crime mid-year update part 1: Cybercrime climbs as exchange thieves and ransomware attackers grow bolder," Chainalysis, August 2024, Accessed: 11.12.2024. [Online]. Available: https://www.chainalysis.com/blog/2024-crypto-crime-mid-year-update-part-1/

[10] ——, "2024 crypto crime mid-year update part 2: China-based csam and cybercrime networks on the rise, pig butchering scams remain lucrative," Chainalysis, August 2024, Accessed: 11.12.2024. [Online]. Available: https://www.chainalysis.com/blog/2024-crypto-crime-mid-year-update-part-2/

[11] Halborn, "Breaking down the top 100 defi hacks," Halborn, 2024, Accessed: 11.12.2024. [Online]. Available: https://www.halborn.com/reports/top-100-defi-hacks

[12] O. Marzouk, "Top 10 crypto losses of 2024: Hacks, frauds, and exploits," Blockchain Intelligence Group, October 2024, Accessed: 11.12.2024. [Online]. Available: https://blockchaingroup.io/compliance-and-regulation/top-10-crypto-losses-of-2024-hacks-frauds-and-exploits/

[13] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186.

[14] H. Rameder, M. Di Angelo, and G. Salzer, "Review of automated vulnerability analysis of smart contracts on ethereum," *Frontiers in Blockchain*, vol. 5, p. 814977, March 2022.

[15] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3391195

[16] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, "A survey on smart contract vulnerabilities: Data sources, detection and repair," *Information and Software Technology*, vol. 159, p. 107221, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584923000757

[17] J. V. Behanan and Shashank, "Smart contract top 10," OWASP, 2023, Accessed: 11.12.2024. [Online]. Available: https://owasp.org/www-project-smart-contract-top-10/

[18] Gerhard Wagner, "Smart contract weakness classification," 2018, Accessed: 13.12.2024. [Online]. Available: https://swcregistry.io/

[19] NCC Group, "Decentralized application security project top 10," 2018, Accessed: 13.12.2024. [Online]. Available: https://dasp.co/

[20] D. Rusinek and P. Kuryłowicz, "Smart contract security verification standard," 2023, Accessed: 11.12.2024. [Online]. Available: https://github.com/ComposableSecurity/SCSVS

[21] D. Muhs, "Smart contract security field guide," 2023, Accessed: 11.12.2024. [Online]. Available: https://scsfg.io/

[22] D. He, R. Wu, X. Li, S. Chan, and M. Guizani, "Detection of vulnerabilities of blockchain smart contracts," *IEEE Internet of Things Journal*, vol. 10, no. 14, pp. 12 178–12 185, 2023.

[23] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, "Security threat mitigation for smart contracts: A comprehensive survey," *ACM Comput. Surv.*, vol. 55, no. 14s, December 2023. [Online]. Available: https://doi.org/10.1145/3593293

[24] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal modeling and verification of smart contracts," in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ser. ICSCA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 322–326. [Online]. Available: https://doi.org/10.1145/3185089.3185138

[25] M. Qu, X. Huang, X. Chen, Y. Wang, X. Ma, and D. Liu, "Formal verification of smart contracts from the perspective of concurrency," in *International Conference on Smart Blockchain*, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:56594850

[26] Z. Nehaï, P.-Y. Piriou, and F. Daumas, "Model-checking of smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018, pp. 980–987.

[27] B. Bernardo, R. Cauderlier, G. Claret, A. Jakobsson, B. Pesin, and J. Tesson, "Making tezos smart contracts more reliable with coq," *CoRR*, vol. abs/2106.12973, Jun 2021. [Online]. Available: https://arxiv.org/abs/2106.12973

[28] Z. Yang and H. Lei, "Fether: An extensible definitional interpreter for smart-contract verifications in coq," *CoRR*, vol. abs/1810.04828, March 2018. [Online]. Available: http://arxiv.org/abs/1810.04828

[29] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," Cryptology ePrint Archive, Paper 2016/633, 2016. [Online]. Available: https://eprint.iacr.org/2016/633

[30] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269.

[31] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Towards smart hybrid fuzzing for smart contracts," *CoRR*, vol. abs/2005.12156, May 2020. [Online]. Available: https://arxiv.org/abs/2005.12156

[32] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019, pp. 8–15.