Potential Attacks and Possible Preventions on Blockchain Infrastructure with respect to Cryptocurrency

[1]Umang Vasistha, [2]Dr. Mary Jacintha M, [3]Vivek Arya, [4]Dr. Debjit Kar, [5]Sh. Notan Roy

[1]Software Developer, Education and Training Department, CDAC Noida, Noida, India
[2]Associate Director, Education and Training Department, CDAC Noida, Noida, India
[3]Project Engineer, Education and Training Department, CDAC Noida, Noida, India
[4]Scientist C, Cyber Laws and Data Governance Division, MeitY, Delhi, India
[5]Scientist E, Cyber Laws and Data Governance Division, MeitY, Delhi, India

Abstract - The world is rapidly transitioning from web2.0 to web3.0, along with it the meaning of currency has also changed. Blockchain is a decentralized and distributed ledger that ensures the integrity of data through cryptographic hashing and consensus mechanisms. With the advent of new generation blockchain technology, money is no longer limited to its traditional form and has taken the shape of crypto currency, which is significantly more safe and dependable. This research paper has identified the potential attacks on blockchain infrastructure with respect to crypto currency. Also, this research paper offers an extensive summary of the current risk environment surrounding blockchain infrastructure in cryptocurrencies and discussing various tool to identify vulnerabilities and possible preventions.

Keywords - Blockchain, Web3.0, Redundancy, Bitcoin, Ethereum, Hashrate, Cryptocurrency.

I. INTRODUCTION

Blockchain [1] technology removes the requirement for a centralized authority or middleman by enabling the creation and verification of digital documents. Smart contracts, asset registries, digital payments, and redundancy are just a few of the other uses for blockchain technology that have been suggested. However, there are a number of issues and restrictions with this technology including security, irreversibility, scale, and regulatory compliance. Blockchain technology is best understood as a means of producing digital currency rather than as a versatile invention with applications across multiple industries.

Digital [2] currency is a form of electronic money that can be used to make payments on the Internet. A digital payment system that aims to replicate the properties of physical cash. Unlike traditional currencies, digital currency does not rely on physical tokens or intermediaries, but on cryptographic techniques and distributed networks. One of the main advantages of digital currency is that it can provide anonymity, security, and efficiency for online transactions. However, there are also challenges and risks associated with digital currency, such as double spending, fraud, regulation, and scalability.

The [3] ease with which money can be transferred between two parties in a transaction lends credence to the positive outlook surrounding the use of cryptocurrencies. For security reasons one uses public and private keys to enable these transactions. Since these financial transfers don't come with any processing expenses, Clients can avoid having to pay substantial fees that are assessed by most banks. Cryptocurrencies like Ethereum, Bitcoin, and others are totally decentralized, implying that nobody truly controls the network. While Bitcoin is a fully digital money intended for use as a store of value or a method of exchange, Ethereum adopts a more all-encompassing approach. Ethereum serves as a platform where users can construct and run applications and more importantly smart deals using ether tokens. Code-written contracts that are added to the ethereum block by their inventor are known as smart contracts. Every node in the network executes each of those contracts, setting it to blockchain.

As the adoption of cryptocurrencies accelerates, the need to identify and address vulnerabilities is blockchain infrastructure becomes important. This research paper provides exploration of the potential attacks that loom over the blockchain infrastructure with respect to cryptocurrencies. In order to transform traditional finance, cryptocurrencies are best known for it. But there are some bad actors looking to take advantage of weakness in blockchain network.

Malicious actors continuously seek to exploit weakness within the architecture, posing threats that range from traditional cyber-attacks to sophisticated exploits targeting consensus mechanisms and smart contracts. This research paper highlights few exploits and there preventions performed on blockchain infrastructure. For early detection of vulnerabilities we have identified a tool named “Slither”. It is developed to analyze smart contracts written in Solidity. It aims to identify security vulnerabilities, potential flaws and coding best practice violations within smart contracts.

II. INTRODUCTION ABOUT ATTACKS

Blockchain infrastructure is often conceptualized and implemented using a layered architecture where different layers serve specific purposes and functionalities. Hardware layer is where the blockchain resides and where all the transactions take place. The nodes on this layer are responsible for validating the transactions and appending them to the chain. Data layer stores all the actual data associated with a blockchain. The data itself is stored in blocks that link together by a cryptographic hash. Network layer facilitates inter-node communication. This layer takes charge of node discovery, block creation, and block addition, ensuring the blockchain network functions in a legitimate state. Consensus layer is responsible for ensuring that all transactions added to the chain are valid and in compliance with rules set out by the network. Application layer consists of the execution layer and the application layer protocols, which include smart contracts, scripts, and frameworks.

Attacks as per blockchain infrastructure layers: -

1. *Hardware Layer: -*

Virtual [4] machines are connected to one another peer-to-peer at the hardware layer. It organizes transactions into blocks for management. Attacks such as man-in-the-middle, network disintegration, eclipse, Sybil, and DDoS are occurring in this layer. A Distributed Denial of Service (DDoS) attack on a blockchain refers to a malicious endeavour to overwhelm and disrupt the regular operation of a blockchain network by overloading it with traffic. In a DDoS attack, a lot of requests are made using several malicious laptops or devices or transactions, overwhelming the blockchain network's capacity to process them.

2. *Data Layer: -*

The [5] signature in Bitcoin uses SSL (Secure Sockets Layer) technology, which allows the signature to remain valid even if an attacker modifies a few bytes. Attackers in this scenario track transactions on the bitcoin network and alter the transactions' signatures while maintaining the transactions' validity. A distinct transaction identity is generated following a transaction's signature change. The two transactions were then broadcast to the bitcoin network by the attackers, creating the double spending effect.

3. *Consensus Layer: -*

The [6] 51% attack is a well-known blockchian attack technique that relies on the theory that a subset of miner's controls over 50% of the network's mining hashrate, or processing power. By preventing confirmations for new transactions from being sent the attackers would have the ability to halt customer transactions and merchants. Attackers are able to finish proof-of-work faster than trustworthy miners. The increased hashrate for mining attacker possess, the quicker blockchain attacks take place. When attackers take over more than 50% hashrate of mining on the network, They may make use of a 51% attack to reverse transactions and spend the same money repeatedly.

4. *Application Layer: -*

A re-entrancy attack is a type in which an exploiter contract uses the victim contract's weakness to keep withdrawing money from it until the victim contract fails. This attack is performed on application layer of blockchain. In this type of attack a fallback function which was developed by the programmer was used.

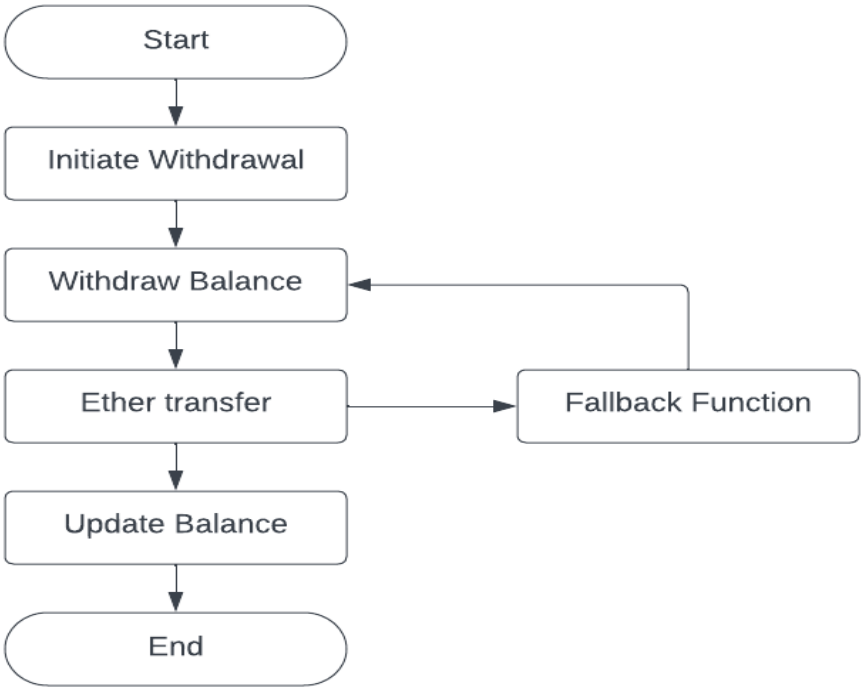

Fig.1 Re-entrancy attack

Taking reference from flowchart in Fig.1, in this attack function triggers another withdraw before updating previous balance. When attacker initiated a transaction smart contracts transfer ether from the smart contract wallet. After that instead of updating smart contract balance a fallback function is called which initiates another withdraw. Attacker keep on withdrawing until victim's wallet balance gets null.

reentrancy > Contract.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Contract {
    mapping(address => uint) public balance;


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

    function Withdraw() public {
        uint balances = balance[msg.sender];
        require(balances>0);

        (bool sent, ) = msg.sender.call{value: balances}("");
        require(sent, "Failed");
        balance[msg.sender] = 0;
    }

    function getBalances() external view returns (uint256) {
        return address(this).balance;
    }

}
```

Fig.2 Contract 1

Developer [7] carelessness is another factor that leads to re-entrancy issues. We describe the operation of the re-entrancy vulnerability by taking example of contract 1 in Fig. 2. To retrieve all of the ethereum's kept in the smart contract, the user may invoke the Contract.Withdraw() function. All of the contract's external calls might be argued to be vulnerable, and there might be re-entrancy issues. One of the feature of ethereum smart contracts is that the contracts can call one another through external calls.

Now let us examine this contract's withdraw function. There is an external call in this function's transfer operation that is (msg.sender.call{value: bal}), hence it seems that there might be a re-entrancy vulnerability in contract 1 because balance is updating after external call and attacker can make external call repeatedly without updating balance of smart contract.

Attack.sol 1 | Contract.sol

reentrancy > Attack.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;
import "./Contract.sol";

contract Attack {
    Contract public Contracts;

    constructor(address _ContractsAddress) {
        Contracts = Contract(_ContractsAddress);
    }

    // Fallback is called when DepositFunds sends Ether to this contract.
    fallback() external payable {
        if (address(Contracts).balance >= 1 ether) {
            Contracts.Withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        Contracts.Deposit{value: 1 ether}();
        Contracts.Withdraw();
    }

    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }

}
```

Fig.3 Contract 2

Attacker deploys Contract 2 mentioned in Fig. 3. Who then uses the Contract.Deposit function to transmit 1 ethereum to contract 1. Once any mount of ethereum is sent to the contract, an anonymous fallback function will be triggered immediately.

Then the Attack.attack calls the Contract.Withdraw function to extract 1 ethereum that was deposited by the attacker. The Attack.fallback function will be activated when Attack.attack uses Contract.Withdraw to withdraw ethereum from the prior deposit.

The Contract.Withdraw method will be triggered to transfer ethereum from the contract 1 to the contract 2, provided that the ethereum in contract 1 is higher than or equal to 1.

Due to the fallback function in contract 2 the program can successfully call the withdraw function again which creates an infinite loop of re-entrancy. It will keep doing this until the contract 1 remaining balance is less than 1. By doing so, the attacker can carry on in this manner until all of the contract 1 ethereum is extracted.

III. TOOL IDENTIFIED FOR AUDITING

A vulnerable, smart contract have tested using Slither auditing tool. It helps to find vulnerabilities and improves code understanding. Slither provides better accuracy and lowest false positive rate of 10.9%. Slither's closest rival is "Securify" which has false positive rate of 25%. Slither has a higher robustness rating because it only fails in 0.1% of contracts as compared to other tools "Securify" and "SmartCheck" which have rating of failed analysis of more than 10%. In Slither, less than one second is the average execution time for each contract which is very low. Slither may identify code patterns that result in expensive code deployment and execution.

Slither [8] is a framework for static analysis created to give detailed insights into the code of smart contracts while maintaining the flexibility required to accommodate a wide range of use cases. Smart contract defects come in a wide variety, and they can be found without user assistance. It is possible to identify a wide range of smart contract problems without the need for user involvement. Code optimizations that the compiler overlooks are found by Slither. Printers help in codebase analysis by summarizing and displaying contract information. Slither computes a series of pre-specified analyses that give the other modules more detailed information.

1. *Integrated Code Analysis:* -

Slither shows which variables are being read and written. It is possible to extract the read or written variables for each contract, function, or node of the control flow graph and filter them according to their kind (local or state). For instance, it is feasible to identify which functions write to a certain variable or to learn which state variables are written from a particular function. This data serves as the basis for a number of detectors, including re-entrancy and uninitialized variable detectors.

The use of ownership to restrict access to functions is a common characteristic in the design of smart contracts. The idea behind this design is that privileged activities can be carried out by a certain user known as the owner. Reducing the amount of false positives is achieved by modeling the protection of functions. Slither looks for situations in which the function is not the constructor in order to identify unprotected functions the address of the caller which is "msg.sender".

First, the dependencies are examined in relation to every function. Subsequently, a fix point is calculated encompassing every function within the contract to ascertain the existence of a reliance in a context with multiple transactions. Slither labels certain variables as contaminated. This means that the variable can be influenced by the user and depends on a variable that the user can control.

2. *Automated Vulnerability Detection:* -

Slither's open-source version comes with over ninety bug detectors. Solidity makes it possible to shadow the majority of the smart contract's components, example local and state variables, functions, or events. In programming languages, uninitialized state or local variables are frequently the cause of failures. Other well-known security flaws include arbitrary ether sending, locked ether, and suicide contracts. Slither may identify code patterns that result in expensive code deployment and execution.

3. *Automated Optimization Detection:* -

Slither may identify coding patterns that lead to the deployment and execution of costly code. The compiler is able to optimize the code by detecting variables that can be declared as constants and functions that can be declared as externals. These declarations do not require additional space. When feasible, use constant variables to minimize use costs and code size, which lowers the cost of contract deployment.

4. *Code Understanding and code review:* -

Slither has printers that help users rapidly comprehend the purpose and format of contracts. An easily understood synopsis of the contracts that includes the quantity of bugs discovered and details regarding the quality of the code. An overview of the variables that the smart contract's owner can alter and the authorization accesses. The Slither Python API allows users to create tools and scripts for third parties. A custom script might address needs related to a given smart contract. It supports developer toolboxes and continuous integration. It simply requires a modern version of the solidity compiler and has few dependencies.

```
INFO:Detectors:
Reentrancy in Contract.Withdraw() (Contract.sol#20-27):
        External calls:
        - (sent) = msg.sender.call{value: balances}() (Contract.sol#24)
        State variables written after the call(s):
        - balance[msg.sender] = 0 (Contract.sol#26)
        Contract.balance (Contract.sol#5) can be used in cross function reentrancies:
        - Contract.Deposit() (Contract.sol#16-18)
        - Contract.Withdraw() (Contract.sol#20-27)
        - Contract.balance (Contract.sol#5)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
Pragma version^0.8.13 (Contract.sol#2) allows old versions
solc-0.8.13 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Contract.Withdraw() (Contract.sol#20-27):
        - (sent) = msg.sender.call{value: balances}() (Contract.sol#24)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Function Contract.Deposit() (Contract.sol#16-18) is not in mixedCase
Function Contract.Withdraw() (Contract.sol#20-27) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Contract.locked (Contract.sol#6) is never used in Contract (Contract.sol#4-34)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable
INFO:Detectors:
Contract.locked (Contract.sol#6) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Slither:. analyzed (3 contracts with 93 detectors), 20 result(s) found
```

Fig.4 Result of smart contract auditing

After auditing contract 1 from Fig. 2 and contract 2 from Fig. 3 against ninety plus detectors of Slither results are displayed in Fig. 4. Contracts contained re-entrancy vulnerability which was showed in results of smart contract auditing. Slither color detectors are in red, yellow and green on the basis of impact of that detector. It detects re-entrancy vulnerability in Contract.withdraw(). External call is made in this function by msg.sender.call. State variable is written after the call which is balance[msg.sender]=0. Due to these two mistakes contract fall under re-entrancy vulnerabilities and Contract.Withdraw() keep on executing because of fallback function in Contract 2. At the end of the result a reference link is given by Slither which redirects to Slither's Github page where we can understand configuration of vulnerabilities, description, exploit scenario and recommendation of each vulnerabilities.

IV. POSSIBLE PREVENTIONS AND RESULT

It has been ensured that the contract has the updated balance when the attacker calls withdraw again by changing the state prior to performing external calls.

reentrancy > Contract.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Contract {
    mapping(address => uint) public balance;
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }


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

    function Withdraw() public noReentrant(){
        uint balances = balance[msg.sender];
        require(balances>0);
        balance[msg.sender] = 0;

        (bool sent, ) = msg.sender.call{value: balances}("");
        require(sent, "Failed");
    }

    function getBalances() external view returns (uint256) {
        return address(this).balance;
    }

}
```

Fig.5 Contract 3

By applying prevention techniques in contract 3 shown in Fig. 5. Adding a reentrancy guard is must. By locking the contract, this stops several functions from being performed simultaneously. A modifier called reentrancy guard can be applied to Contract.Withdraw(). By

locking the smart contract, multiple functions cannot be performed simultaneously. This ensures that attacker cannot call external multiple functions simultaneously and smart contract has the updated balance before attacker calls withdraw function.

The noReentrant() modifier is applied to functions that need protection against reentrancy. It checks the locked flag before allowing the function to execute. If the flag is already true, signifying a continuing execution, the function reverts, preventing reentrancy. In this example, the withdraw function is protected by the noReentrancy modifier. The reentrancy guard prevents the function from being called again until it has completed its execution, thereby reducing the risk of reentrancy attacks.

It has been completely removed the chance of a recursive call being exploited with this reentrancy guard technique. It's crucial to exercise prevention while creating external contract calls in smart contracts.

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Pragma version^0.8.13 (Attack.sol#2) allows old versions
Pragma version^0.8.13 (Contract.sol#2) allows old versions
solc-0.8.13 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Contract.Withdraw() (Contract.sol#19-26):
        - (sent) = msg.sender.call{value: balances}() (Contract.sol#24)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Variable Attack.Contracts (Attack.sol#6) is not in mixedCase
Function Contract.Deposit() (Contract.sol#15-17) is not in mixedCase
Function Contract.Withdraw() (Contract.sol#19-26) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Attack.Contracts (Attack.sol#6) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Detectors:
Pragma version^0.8.13 (Contract.sol#2) allows old versions
solc-0.8.13 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Contract.Withdraw() (Contract.sol#19-26):
        - (sent) = msg.sender.call{value: balances}() (Contract.sol#24)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Function Contract.Deposit() (Contract.sol#15-17) is not in mixedCase
Function Contract.Withdraw() (Contract.sol#19-26) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:. analyzed (3 contracts with 93 detectors), 14 result(s) found
PS D:\Truffle\reentrancy>
```

Fig.6 Result of auditing after applying locks

The results of Contract 3 shown in Fig. 5 after auditing using Slither is shown in Fig. 6. It shows reentrancy vulnerability that was found in Contract 1's Contract.Withdraw() function is now resolved. We have implemented noReentrant() in Contract.Withdraw() function and external call (msg.sender.call{value: bal}) is executing after balance updation and we are good to go for deployment of smart contract.

V. Conclusion and Further Work

The significance of the smart contract security audit has been emphasized and the smart contract security flaws has been introduced. By performing auditing of vulnerable smart contracts, using Slither auditing tool. Slither's ability to detect bugs was assessed and concluded Slither is showing fastest and accurate results in comparison of other smart contract auditing tools. By implementing preventions, found vulnerability in smart contract is removed and smart contract is safe for deployment.

Further, there are several directions that we could take to improve smart contract and prevent blockchain infrastructure from potential attacks with respect to cryptocurrency. Since the instrument is utilized on a daily basis for smart contract audits the incorporation of new issue detectors is the first enhancement. Future research will focus on creating mitigation and detection mechanisms for the significant security vulnerabilities this research paper highlights.

VI. References

[1] Saifedean Ammous, "Blockchain Technology: What is it Good for?" (August 8, 2016). Available at SSRN: https://ssrn.com/abstract=2832751.

[2] Sai Anand, R Madhavan, C. (2000). An Online, Transferable E-Cash Payment System. In: Roy, B., Okamoto, E.(eds) Progress in Cryptology – INDOCRYPT 2000. INDOCRYPT 2000. Lecture Notes in computer Science, vol 2977.

[3] T. Boshkov, 'Blockchain and Digital Currency in the World of Finance', Blockchain and Cryptocurrencies. IntechOpen, Aug. 28, 2019. doi: 10.5772/intechopen.79456.

[4] U. Javaid, A.K. Siang, M.N. Aman, B. Sikdar, Mitigating loT device based DDoS attacks using blockchain, in: Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems, CryBlock'18, ACM, New York, NY, USA, 2018, pp. 71–76, http://dx.doi.org/10.1145/3211933.3211946, http://doi.acm.org/10.1145/3211933.3211946.

[5] Priya Maidamwar, Nekita Chavhan, "Blockchain Technology: A review on architecture, security issues and challenges" Vol. 4, Issue 12, ISSN No. 2455-2143.

[6] C. Ye. G. Li, H. Cai, Y. Gu and A. Fukuda, "Analysis of Security in Blockchain: Case Study in 51%-Attack Detecting," 2018 5th International Conference on Dependable Systems and Their Applications (DSA), Dalian, China, 2018, pp. 15-24, doi: 10.1109/DSA.2018.00015.

[7] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng and N. Guizani, "Smart Contract Vulnerability Analysis and Security Audit," in IEEE Network, vol. 34, no. 5, pp. 276-282, September/October 2020, doi: 10.1109/MNET.001.1900656.

[8] J. Feist, G. Grieco and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," *2019 IEEE/ACM 2nd I nternational Workshop on Emerging Trends in Software Engineering for Blockchain(WETSEB),* Montreal, QC, Canada, 2019, pp. 8-15, doi: 10.1109/WETSEB.2019.00008.

[9] S. Sayeed, H. Marco-Gisbert and T. Caira, "Smart Contract: Attacks and Protections," in IEEE Access, vol. 8, pp. 24416-24427, 2020, doi: 10.1109/ACCESS.2020.2970495.

[10] A. Kosba, A. Miller, E. Shi, Z. Wen and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts," 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 2016, pp. 839-858, doi: 10.1109/SP.2016.55.

[11] W. Wang, J. Song, G. Xu, Y. Li, H. Wang and C. Su, "ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts," in IEEE Transactions on Network Science and Engineering, vol. 8, no. 2, pp. 1133-1144, 1 April-June 2021, doi: 10.1109/TNSE.2020.2968505.

[12] S. Nakamoto et al., Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[13] How does Bitcoin Work?, Jul. 2009, [online] Available: https://bitcoin.org/en/how-it-works.

[14] F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, et al., "EVM: From offline detection to online reinforcement for Ethereum virtual machine", Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reeng. (SANER), pp. 554-558, Feb. 2019.

[15] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena and A. Hobor, "Finding the greedy prodigal and suicidal contracts at scale", ACSAC, 2018.

[16] Phil Daian, Analysis of the dao exploit, June 2016, [online] Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[17] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli and M. Vechev, "Securify: Practical security analysis of smart contracts", CCS '18, 2018.

[18] N. Atzei, M. Bartoletti and T. Cimoli, "A survey of attacks on ethereum smart contracts", Proc. Int. Conf. Princ. Secur. Trust, pp. 164-186, 2017.

[19] S. Sayeed and H. Marco-Gisbert, "Assessing blockchain consensus and security mechanisms against the 51% attack", Appl. Sci., vol. 9, no. 9, pp. 1788, Apr. 2019.

[20] W. Shahda, Protect Your Solidity Smart Contracts From Reentrancy Attacks, Oct. 2019, [online] Available: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21.

[21] H. Olickel, Why Smart Contracts Fail: Undiscovered Bugs What We Can do About Them, Jul. 2016, [online] Available: https://medium.com/hrishiolickel/why-smart-contracts-fail-undiscovered-bugs-and-what-we-can-do-about-them-119aa2843007.

[22] J. Feist, G. Grieco and A. Groce, "Slither: A static analysis framework for smart contracts", Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB), pp. 8-15, May 2019.

[23] S. Sayeed, H. Marco-Gisbert, I. Ripoll and M. Birch, "Control-flow integrity: Attacks and protections", Appl. Sci., vol. 9, no. 20, pp. 4229, Oct. 2019.

[24] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli and M. T. Vechev, "Securify: Practical security analysis of smart contracts", Proc. ACM SIGSAC Conf. Comput. Commun. Secur., pp. 67-82, Oct. 2018.

[25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static analysis of Ethereum smart contracts", Proc. 1st Int. Workshop Emerging Trends Softw. Eng. Blockchain-WETSEB, pp. 9-16, 2018.

[26] L. Luu, D.-H. Chu, H. Olickel, P. Saxena and A. Hobor, "Making smart contracts smarter", Proc. ACM SIGSAC Conf. Comput. Commun. Secur.-CCS, pp. 254-269, 2016.

[27] B. Jiang, Y. Liu and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection", Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.-ASE, pp. 259-269, 2018.

[28] M. Fu, L. Wu, Z. Hong, F. Zhu, H. Sun and W. Feng, "A critical-path-coverage-based vulnerability detection method for smart contracts", IEEE Access, vol. 7, pp. 147327-147344, 2019.

[29] M. Rodler, W. Li, G. O. Karame and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks", arXiv:1812.05934, 2018, [online] Available: https://arxiv.org/abs/1812.05934.

[30] J. Feist, G. Grieco and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," *2019 IEEE/ACM 2nd I nternational Workshop on Emerging Trends in Software Engineering for Blockchain(WETSEB),* Montreal, QC, Canada, 2019, pp. 8-15, doi: 10.1109/WETSEB.2019.00008.