Exploring Vulnerabilities in Solidity Smart Contract

Phitchayaphong Tantikul and Sudsanguan Ngamsuriyaroj[a]
Faculty of Information and Communication Technology, Mahidol University, Nakhon Pathom, Thailand

Keywords: Smart Contract, Solidity, Security, Vulnerability.

Abstract: A smart contract is a decentralized program executed automatically, reliably, and transparently on a blockchain. It is now commonly used in financial-related applications, which require heavily secure operations and transactions. However, like other programs, smart contracts might contain some flaws. Thus, developers are encouraged to write secure smart contracts, and some approaches are proposed to detect vulnerabilities of smart contracts before deployment. Due to the immutability property of a blockchain, developers cannot modify the smart contract even though there is a vulnerability which may cause financial losses. In this paper, we propose the comparison of vulnerability detection tools to deployed smart contracts on the Ethereum blockchain. We also present the analysis of the state of vulnerabilities in smart contracts as well as their characteristics.

1 INTRODUCTION

A smart contract is a small piece of codes operating on a blockchain system and now has various usages, for example, as a currency token, as an escrow, as a market exchange or as a game. Due to the characteristics of the blockchain, the smart contract's code is stored immutably inside a block. In addition, when the code starts to be executed, the result will be stored in the form of a transaction in the blockchain. Since anyone can read the code and see the results, the transparency property is given to the smart contracts and their execution process. The code can also be executed by every responsible machine to ensure the consistency of the execution results. Although the storage of most blockchain systems and the smart contract's code are immutable, the state of variables inside the smart contract's code is not. Specifically, such variables could be changed by the execution of the programming logic of the code. If the code has some vulnerabilities, anyone with malicious intentions can manipulate the code for his gain or for disrupting the functions of the smart contract. Thus, it is crucial to identify the vulnerabilities of a smart contract so that the code can be secured properly.

There are many cases caused by programming flaws of the smart contracts. For example, in the DAO case (Siegel, 2016), attackers stole about \$150 million worth of Ethers (Ethereum's currency unit) from a crowd-funding smart contract by triggering a vulnerable function to send all Ethers inside the contract to themselves. In the case of Parity's wallet bug (Parity Technologies, 2017), an attacker triggered a flaw to set an uninitialized variable of a smart contract used as the main library of smart contract wallets of other Parity customers. The damage caused all Ethers inside those wallets to be frozen, and there is no way to withdraw any Ether from them.

Several efforts had gathered and created a list of vulnerabilities of smart contracts. For instance, Solidity, the most popular language for smart contract development, has listed vulnerabilities on its official document (Ethereum, 2019). Moreover, Consensys, one of the smart contract audit firms, collected Solidity's coding flaws into Solidity Best Practice (Consensys, 2019), and Mythril team has created Solidity Weakness Repository (Smart Contract Security, 2019). Such lists could help smart contract developers in avoiding repetitions of similar mistakes. Another approach is to compose a secure library for common coding patterns. Zeppelin, a smart contract audit firm, created an open-source project named OpenZeppelin (OpenZeppelin, 2019) to provide a library of audited smart contracts for developers to use. By extending the audited smart contract library, potential bugs could be minimized, and the time to code a smart contract could be reduced as well. However, a developer could still make the same mistakes while writing the code. Therefore, approaches to analyze smart contracts to detect their vulnerabilities using au-

[a] https://orcid.org/0000-0002-7079-2408

Tantikul, P. and Ngamsuriyaroj, S.
Exploring Vulnerabilities in Solidity Smart Contract.
DOI: 10.5220/0008909803170324
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 317-324
ISBN: 978-989-758-399-5; ISSN: 2184-4356

tomation methods are proposed. Oyente (Luu et al., 2016), the first tool in detecting vulnerabilities in Solidity, uses symbolic execution to test for vulnerabilities. SmartCheck (Tikhomirov et al., 2018) uses static analysis, which parses the source code of the smart contract to find out whether the code contains common vulnerability patterns.

Although many works are invented to detect vulnerabilities of smart contracts, there is still no research in investigating the current state of vulnerable smart contracts. Our research aims to discover common occurrences and trends of vulnerabilities in smart contracts as well as identify common characteristics of vulnerable smart contracts. We have collected the source code of 38,982 smart contracts from Etherscan.com. We analyze smart contracts written in Solidity in the Ethereum blockchain as it is widely well-adopted. From those smart contracts, we have found common occurrences and trends of vulnerabilities in already deployed smart contracts. In addition, we have suggestions for developers when developing a smart contract.

The remainder of this paper is organized as follows. Section 2 gives the background of smart contract vulnerabilities. Section 3 explains the proposed work of this research. Section 4 shows the results and discussions. Finally, we conclude our work in Section 5.

2 LITERATURE REVIEW

2.1 Ethereum Smart Contract

Unlike Bitcoin, the most successful implementation of a blockchain system, that only uses the blockchain to store currency-transferring transactions, and the accumulation of those transactions becomes a ledger where the balance of every account on the system is kept. The goal of Ethereum(Wood, 2014) is to use a blockchain to implement distributed applications. In Ethereum, an account's balance is a state that changes values by transactions. In other words, a transaction is a state transition operator. The global state is where Ethereum uses to store its own currency, Ether, on each account. Moreover, Ethereum allows each account to store code and has inner state variables, and that becomes a smart contract. The inner state variables are changed by processing a transaction that contains a function call, probably with arguments, against the account's code as shown in Figure 1. All Ethereum's transactions, including smart contracts, are operated by Ethereum virtual machine (EVM).

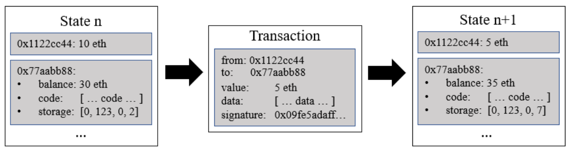


Figure 1: Ethereum State Transition.

EVM operates on the machine-level code, called bytecode, which is difficult for a human to write. Therefore, Ethereum invented several new programming languages for developers to use and later compiled to bytecode for deployment. These languages are LLL, Serpent, Tiger, Solidity and Vyper, and Solidity which is currently the most popular language.

Calling to a function in a smart contract is a form of a transaction. The caller can create a transaction with data part containing function signature and arguments. The caller can choose to send some Ether along with the transaction. When a transaction is committed to a block and distributed to nodes in the network, each EVM will extract calling function and arguments, and then execute the callee function with the extracted information. The Ether sent along with the transaction will be added to the smart contract's account balance.

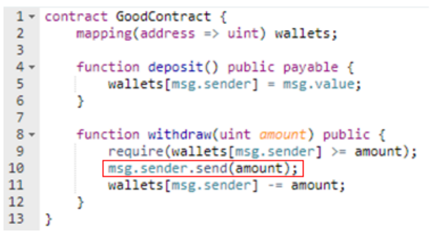
2.2 Smart Contract's Vulnerabilities Analysis

Information on vulnerabilities of Solidity Smart Contracts has been collected ((Ethereum, 2019), (Consensys, 2019), (Atzei et al., 2017), (Luu et al., 2016), (Tikhomirov et al., 2018)). Most studies focus on the analysis of previous weaknesses found in various incidents such as the DAO case (Siegel, 2016), King of Ether case (King of Ether Throne, 2016), and Parity case (Parity Technologies, 2017). The followings explain the most important vulnerabilities.

2.2.1 Re-entrancy

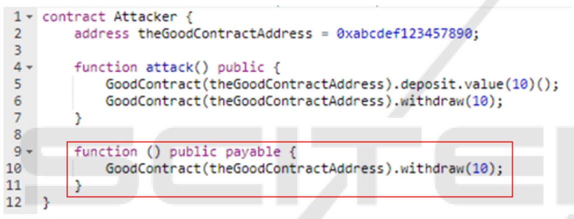
In Solidity, there are three functions used to transfer some currency to an external address; they are `send`, `transfer`, and `call`. However, in a case that the destination address is a smart contract, these functions also act as a function call to "fallback function" in the destination smart contract. A malicious contract might use this fact to create a "crafted fallback function" to execute something back in the original contract.

Figure 2 shows an example of the re-entrancy attack on a withdrawing function, and it is similar to the cause of the incident happened in the DAO case (Siegel, 2016). The attacker starts by creating his/her own malicious smart contract (2b) and calls to the withdraw function of the victim's smart contract. After passing some validation on the first line, the contract sends Ether to the attacker's smart contract. Since the destination of the transfer is a smart contract, the fallback function is executed, which in turn, calling to the withdraw function again. Because the balance of the attacker is not deducted yet (in Line 11 of Figure 2a), the victim's smart contract will send out Ether again. The execution loop continues until either the balance of the victim's smart contract is zero or the transaction gas is depleted.



```
1  contract GoodContract {
2      mapping(address => uint) wallets;
3
4      function deposit() public payable {
5          wallets[msg.sender] = msg.value;
6      }
7
8      function withdraw(uint amount) public {
9          require(wallets[msg.sender] >= amount);
10         msg.sender.send(amount);
11         wallets[msg.sender] -= amount;
12     }
13 }
```

(a) Victim's smart contract.



```
1  contract Attacker {
2      address theGoodContractAddress = 0xabcdef123457890;
3
4      function attack() public {
5          GoodContract(theGoodContractAddress).deposit.value(10)();
6          GoodContract(theGoodContractAddress).withdraw(10);
7      }
8
9      function () public payable {
10         GoodContract(theGoodContractAddress).withdraw(10);
11     }
12 }
```

(b) Attacker's smart contract.

Figure 2: An example of re-entrancy attack.

2.2.2 Integer Overflow and Underflow

Integer overflow and underflow are common problems also found in other programming languages. Integer overflow occurs when increasing a value in a limited-size variable till it is over the maximum capacity. The value of the variable then goes to the lowest value as illustrated in the following code snippet:

```
uint8 num_users = 255;
num_users += 1;
```

This code creates a variable `num_users` as an 8-bit unsigned integer having the range of possible values from 0 to 255, and initiating its value to 255. When the variable is increased by 1 and cannot store the value of 256, the value gets truncated to only 8 bits and zeroes are stored instead. The same principle is applied for integer underflow.

2.2.3 Timestamp Dependency

Since a smart contract operates on an Ethereum Virtual Machine (EVM) that only provides information regarding the smart contract itself, i.e. its transactions and blocks. It does not provide information about environment, such as its host operating system, IP address, or even time. A smart contract developer would seek to find the information from the timestamp field in the block's metadata. Unfortunately, the block's timestamp field is arbitrary and the block's miner can write any timestamp he wants without any verification from other nodes in the network. If a smart contract relies on such timestamp information, it could be tricked by a malicious miner.

2.2.4 Transaction Ordering Dependency

Each operation in a smart contract is a transaction and even if multiple transactions are not operated in a parallel fashion, the order of those operations might yield different results. Consider an example case: assuming a smart contract has a record that Alice initially has 0 token in her account, and she issues two consecutive operations: `withdraw(1000)` and `deposit(1000)`. If the transactions are ordered as `withdraw(1000); deposit(1000);`, Alice will not have enough balance to withdraw and the first operation will fail while the second one will pass, and Alice has 1000 tokens remaining. If the order is reversed, both transactions would be completed, and Alice will have 0 remaining tokens in her account. Miners are responsible in collecting transactions and creating a block. It could be possible for a malicious miner to re-arrange depending transactions in such a way that the result would benefit him.

2.2.5 Using Send

`send` is a Solidity's function for transferring Ether from a smart contract to an external address. If the external address is a smart contract, it would perform an additional function by executing code in the fallback function in the external address. Therefore, using the send function in a smart contract could lead to a vulnerability, and the developer could use the `transfer` function instead if they want to transfer Ether and prohibit the code execution.

2.2.6 Unchecked Calls

Executing a code in an external smart contract could be performed by `send`, `call` and `delegatecall` functions. However, such functions do not stop the execution or throw any error if the callee contract has a runtime error. Instead, the functions would return false and continue running on the origin contract. Thus, it could lead to an erroneous execution flow. For example, consider the following code for withdrawing from a token contract:

```
msg.sender.send(amount);
wallet[msg.sender] -= amount;
```

If the `send` in the first line fails, the Ether would not be properly sent to the `msg.sender` address, and it would be successfully deducted from the wallet's balance in the second line. The developer should have put an "if condition" surrounding the `send/call/delegatecall` function to handle the case when it fails.

2.2.7 Denial of Service with Throw

A smart contract relying on the result from an external contract could be vulnerable from this vulnerability. An example case is the King of Ether Throne game (King of Ether Throne, 2016). This game requires a new king to bid higher amount than the current king. If it succeeds, the contract sends the amount to the current king's address, some fee to developers, and finally set the new king. However, if the current king's address is a smart contract, and its fallback function issues the `throw` command to stop the execution flow which means the refusal of receiving any Ether. The payment to the current king will never succeed, and no one can overthrow the current king.

2.2.8 Gas Limits and Loops

Ethereum has a great concern about computation and storage needed to perform each operation. So, it requires a user calling a smart contract to pay some amount of Ether as "gas" to help fund the operations. The higher the computation or storage required for each operation, the higher the gas needed. An unaware developer might write a smart contract function containing high gas-cost operations, which will fail to call if the caller does not provide enough gas. Consider an example below.

```
address[] accounts;
uint public fixedInterest = 0.0001 ether;
function distributeInterest() public {
    require(msg.sender == owner);
    for (uint i=0; i<accounts.length; i++) {
        accounts[i].transfer(fixedInterest);
    }
}
```

In this example, a developer wants to distribute the interest to every address stored in the `accounts` variable. Calling to this function might fail when the number of addresses is growing larger and the cost of looping over the `transfer` function, a high-cost function, becomes too high.

3 PROPOSED METHODOLOGY

Figure 3 shows the system diagram of our proposed work. The first step is to collect the smart contracts deployed on Ethereum's blockchain, and put into a repository. The second step is to perform vulnerability analysis using Oyente and SmartCheck tools. In the final step, we do the correlation analysis on the vulnerability results to get some insights of the vulnerabilities states in the smart contracts.

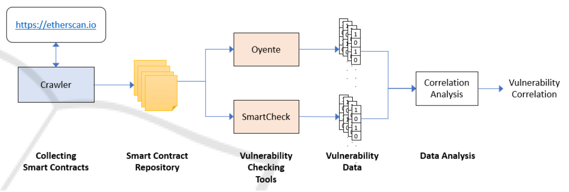

https://etherscan.io
Crawler
Oyente
SmartCheck
Correlation Analysis
Vulnerability Correlation
Collecting Smart Contracts
Smart Contract Repository
Vulnerability Checking Tools
Vulnerability Data
Data Analysis

Figure 3: System diagram.

3.1 Collecting Smart Contracts

Ethereum's blockchain is a good source for collecting smart contracts. However, they are stored in the bytecode format, and it would be difficult to be analyzed for vulnerabilities. In this work, we collected the source code of smart contracts from EtherScan.io - a website providing information on Ethereum's blockchain data and smart contracts. The website allows developers to publish their source codes and verifies them if the source code corresponding with the bytecodes is deployed on the blockchain, so that people wanting to interact with the smart contract can view the source code and the logic inside the contract and be able to trust the contracts easier. Although EtherScan.io does not have a collection of every smart contract deployed on Ethereum's blockchain, it contains a large collection of smart contracts in the form of source codes to be analyzed. In addition, we wrote a crawler program to collect the information on verified smart contracts, metadata, application binary interface (ABI), source code and bytecode. We successfully collected 38,982 contracts by August 15, 2018. Metadata of the contracts are displayed in Table 1.

Table 1: Metadata of the collected smart contracts from EtherScan.io.

Field	Description
address	Address of the deployed smart contract
name	Name of the smart contract (name of the main class)
compiler_version	Version of the compiler
compiler_version_has_bug	Whether version of the compiler is outdated (checked by EtherScan.io)
balance	Current balance of the smart contract (in Ether)
tx_count	Number of transactions related to the smart contract
optimization_enabled	Whether the smart contract is compiled with optimization
date_verified	Date of the smart contract source code is verified by EtherScan.io

3.2 Vulnerability Data Collection

The source code of the smart contracts was analyzed by two analysis tools – Oyente (Luu et al., 2016) and SmartCheck (Tikhomirov et al., 2018).

3.2.1 Oyente

Oyente (Luu et al., 2016) is a tool used to perform the smart contract analysis for vulnerabilities. Due to its early stage of development, a small number of issues are listed and only 6 issues can be detected as shown in the top of Table 2 which are re-entrancy, callstack depth, underflow, overflow, transaction ordering, and timestamp dependency. Oyente builds a control flow graph from the compiled smart contract bytecode, and uses the symbolic execution method to detect whether any point in the execution path could be attacked by any issue. We use the version from the author's paper obtaining from the official docker image to perform vulnerability analysis on our smart contract collection.

3.2.2 SmartCheck

SmartCheck (Tikhomirov et al., 2018) is another tool we used for our vulnerability analysis. The tool employs a different approach for detecting vulnerabilities of the actual source code. It parses a smart contract code into an abstract syntax tree, encodes it into XML, and searches for vulnerability patterns using XPath.

SmartCheck could detect all security and coding issues shown in Table 2, except that underflow and overflow were disabled from inside of the tool dues to high false positive.

3.3 Vulnerability Correlation Analysis

We perform the correlation analysis on the detected vulnerabilities to understand which vulnerabilities could be related and what could be the common cause to be avoided when developing a smart contract. The correlation is a statistical analysis that helps determine whether two variables could be related. In this paper, we use Pearson Correlation which can be calculated using the following formula. The correlation value is ranged from -1 to +1 where -1 means negative correlation, 0 means no correlation, and +1 means linear correlation.

$$\rho(X,Y) = \frac{E[(X-\mu_x)(Y-\mu_y)]}{(\sigma_x\sigma_y)} \qquad (1)$$

Where:

- X, Y are two variables
- ρ is the correlation coefficient between X and Y
- E is the expectation value
- μ_X, μ_Y is the mean of X and Y
- σ_X, σ_Y is the standard variation of X and Y

4 RESULTS AND DISCUSSION

We classified our analysis results into two categories: trend of vulnerabilities and their correlations.

4.1 Trends

4.1.1 New Smart Contracts Over Time

The number of new smart contracts had been growing over time. It reached the peak around February thru April 2018 with the average of 3,537 new smart contracts per month. This may result from the Bitcoin price rise to its peak of $18,953 in December 2017 since the number has been later declined as well as the concurrent decrease in the cryptocurrency market.

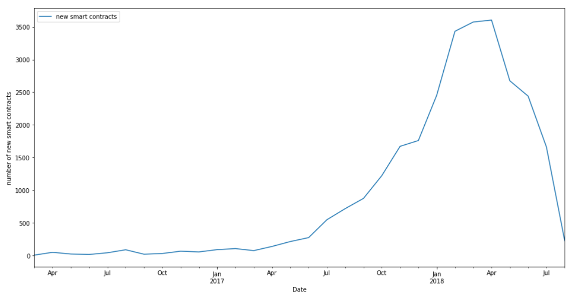


Figure 4: The number of new smart contracts over time.

Table 2: List of security coding issues.

Issue	Description
Re-entrancy	An attacker redirects the control flow back to the victim's smart contract, to create unintended consequence.
Callstack Depth	EVM cannot execute functions higher than 1024 in depth, attacker could make use of this fact and cause some target function to fail.
Underflow	Deduct an integer value to under the minimal value of the data type, `uint8(0) - uint8(1) == 255`.
Overflow	Increase an integer value to over the maximum value of the data type, `uint8(255) + uint8(1) == 0`.
Transaction Ordering	Miner can choose the order of transactions included in a block, thus a set of dependent transactions might yield different results on different orderings.
Timestamp Dependency	Get time from using `block.timestamp` or now might be incorrect because a miner can manipulate this field.
tx.origin	`tx.origin` is a variable referenced to the original caller who initiate the call, not the immediate caller to the function.
Using send	Using `send` instead of `transfer` for payment is insecure.
Unchecked External Calls	Detect using `send`, `delegatecall`, or `call` without under if condition.
DoS by external contract	Detect whether the contract relies on an external call's result. If the external call fails with throw or revert, the caller contract cannot continue, or denial of service (DOS).
Costly loop	Detect a function call within a loop. It might consume too much gas.
Balance Inequality	Prefer `>=` or `<=` , rather than `==`.
Malicious library	Detect if the contract use any library.
Transfer forward all gas	Using `addr.call.value(x)()` could send x Ether and all remaining gas to the `addr` contract.
Integer Division	Solidity does not support floating point data type; integer division will be rounded down.
Locked money	Detect the contract contains none of functions for sending money out of the contract.
Unsafe type inference	Solidity chooses smallest data type that appropriates to the initial data assigned to a variable. If later the variable is used to store larger value, it could overflow.
Byte array	Prefer using `bytes` instead of `byte[]`
Token API violation	API standard for creating digital token (ERC20) does not support throwing exception (e.g. throw, revert, require, assert).
Compiler version not fixed	Source code specified compiler version with operator ˆ allowing it to be compiled with future compiler version which might not have backwards compatible.
Private modifier	Possible misconception of using `private` modifier to hide variable's data.
Redundant fallback function	Detects if source code contains `function () payable throw;` It is already built-in compiler version 0.4.0.

4.1.2 Vulnerabilities of the Smart Contracts (Detected by Oyente) Over Time

As the number of new smart contracts has risen, the vulnerabilities detected in smart contracts also grew. The `overflow` and `underflow` vulnerabilities were found to be the first and second most commonly vulnerabilities detected by Oyente in the smart contract. The remaining vulnerabilities are `compiler_version_has_bug`, `tx_ordering_dep`, `timestamp_dep`, `re-entrancy`, `parity_bug`, in the decreasing order. `compiler_version_has_bug` is the result of EtherScan.io to detect whether a smart contract is compiled using an old version of the solidity compiler which has some known vulnerabilities. The rise of this vulnerability happened before October 2017 was dropped after issuing Solidity Compiler (solc) version 0.4.18 which fixed 10 bugs.

4.1.3 Vulnerabilities of the Smart Contracts (Detected by SmartCheck) Over Time

Similarly to Oyente's results, SmarCheck also illustrates the growth of detected vulnerabilities. The growth of the number of smart contracts occurred before July 2018 before starting to decline. The top 3 highest vulnerabilities are `pragmas_version`, `visibility`, and `malicious_libraries`. They were rising as number of smart contracts grew which could be resulted from that they detect typical keywords in the smart contract. One major observation is that many vulnerabilities had been slowly dropped due to more uses of libraries (`malicious_libraries` only detects whether the contract uses *library* keyword). In addition, `compiler_version_has_bug` had dropped significantly since October 2017 when the new version of the compiler with fixed bugs was introduced.

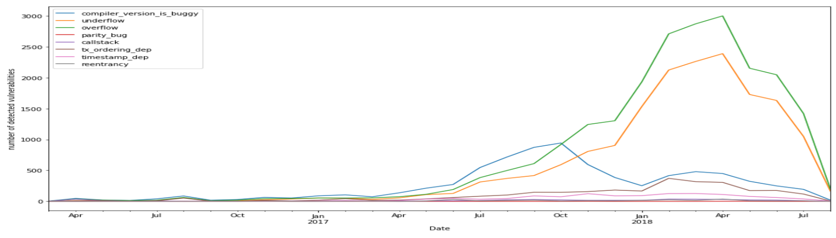


Figure 5: The number of vulnerabilities (detected by Oyente) over time.

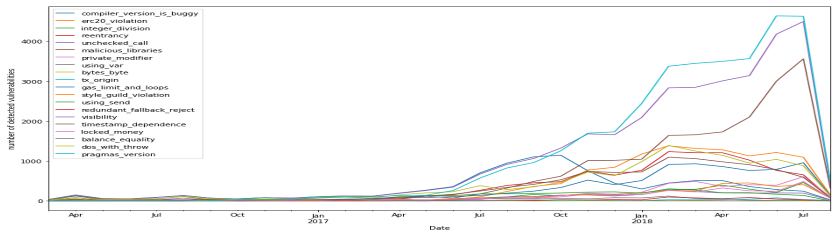


Figure 6: The number of vulnerabilities (detected by SmartCheck) over time.

4.2 Correlation

In this paper, we perform correlation computation via Pearson's correlation in order to understand relations between vulnerabilities. The computed correlation values are used to investigate how often any pair of vulnerabilities found on the same smart contract.

4.2.1 Correlation of Vulnerabilities Detected by Oyente

Figure 7 illustrates the correlation matrix of vulnerability issues detected by Oyente, and it is obvious that the overall correlations between each pair of those issues are very low, ranging from -0.1 to 0.3. In other words, they are highly unrelated. The highest correlation coefficient value is 0.3 and it happens in the pair of "Overflow and Underflow" and "Timestamp Dependency and TX-Ordering". The detailed discussion is given below.

- **Overflow** and **Underflow** pair has the highest correlation of 0.3. Their relation is clear because they are caused by the same issue which is the improper handling of integer value greater than the declared storage size.
- **Timestamp Dependency** and **TX-Ordering** pair also has the highest correlation of 0.3. Their relation might stem from the same root cause that the block can be manipulated by a malicious miner.

4.2.2 Correlation of Vulnerabilities Detected by SmartCheck

Figure 8 illustrates the correlation matrix of vulnerability issues detected by SmartCheck. Obviously, the overall correlation results from SmartCheck results have a wider range than those of Oyente as the range varies from -0.1 to 0.6 with the highest correlation coefficient value at 0.6. The four highest correlated pairs are:

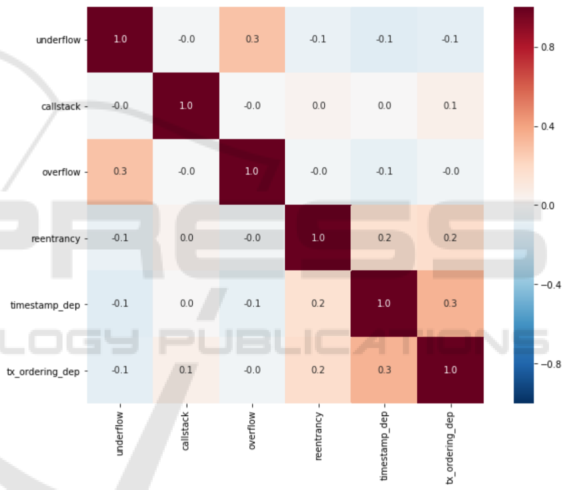


Figure 7: Correlation matrix of vulnerabilities detected by Oyente.

- `DoS_by_external_contract` and `costly_loop` ($\rho = 0.6$) pair: Their similarity relation could have a common cause that both of them detect an external call within a loop condition. However, the effects of both vulnerabilities are different; `DoS_by_external_contract` will stop the execution while `costly_loop` focuses on the cost of execution.
- `timestamp_dependency` and `re-entrancy` ($\rho = 0.5$) pair: These two vulnerabilities often occur

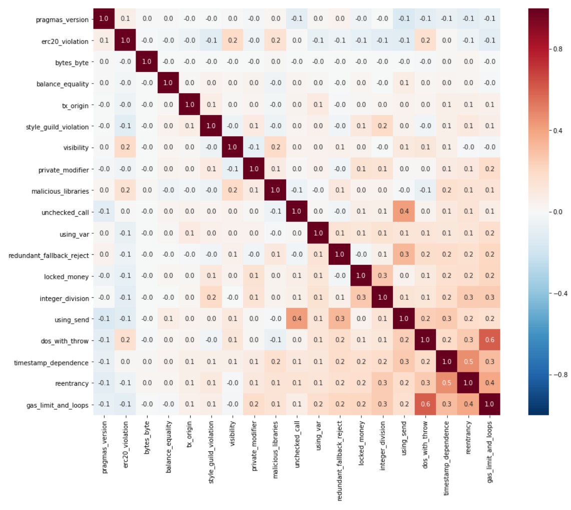


Figure 8: Correlation matrix of vulnerabilities detected by SmartCheck.

together naturally since `timestamp_dependency` is detected by finding *now* keyword, whereas `re-entrancy` checks for external contracts followed by an internal call. Thus, the methods of detection are different.

- `costly_loop` and `re-entrancy` (ρ = 0.4) pair: The common cause between the two relation is the detection of external contract callings.
- `using_send` and `unchecked call` (ρ = 0.4) pair: They are related by the same cause since `using_send` checks that the contract is using the *send* keyword, whereas `unchecked_call` checks for whether the contract calls to an external contract, either by *send*, *call*, or *delegatedCall* keyword.

5 CONCLUSIONS

In this paper, we have collected the source code of 38,982 smart contracts, performed vulnerability analysis using Oyente and SmartCheck which took different approaches to analyze, and calculated correlation coefficient for each pair of the detected vulnerabilities. As a result, we found many highly related pairs as they have some common cause of the vulnerabilities including overflow and underflow, timestamp-dependency and transaction-ordering, using-send and unchecked-external-call. We also found that a group of related vulnerabilities often occurs together since the contract has an external contract calling: DoS by external contract, costly loop, and re-entrancy. Lastly, we found that a pair between timestamp-dependency and re-entrancy is not related by a common cause, but often appears together from the data we collected. From the results we investigate, we would recommend smart contract developers to be cautious when using external calls, when relying on block metadata, and when using math operations.

REFERENCES

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In Maffei, M. and Ryan, M., editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg. Springer Berlin Heidelberg.

Consensys (2019). Ethereum smart contract best practices. Retrieved from https://consensys.github.io/smart-contract-best-practices/.

Ethereum (2019). The solidity contract-oriented programming language documentation. Retrieved from https://solidity.readthedocs.io/.

King of Ether Throne (2016). Kotet - post-mortem investigation. Retrieved from https://www.kingoftheether.com/postmortem.html.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 254–269, New York, New York, USA. ACM Press.

OpenZeppelin (2019). Openzeppelin. Retrieved from https://openzeppelin.com.

Parity Technologies (2017). A postmortem on the parity multi-sig library self-destruct. Retrieved from https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

Siegel, D. (2016). Understanding the dao attack. Retrieved from https://www.coindesk.com/understanding-dao-hack-journalists.

Smart Contract Security (2019). Overview · smart contract weakness classification and test cases. Retrieved from https://SmartContractSecurity.github.io/SWC-registry/index.html.

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. (2018). SmartCheck: Static Analysis of Ethereum Smart Contracts Sergei. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain - WETSEB '18*, pages 9–16, New York, New York, USA. ACM Press.

Wood, G. (2014). Ethereum: a secure decentralised generalised transaction ledger. pages 1–32.