

# Solidity Smart Contracts: Language constructs for control/currency exchange and guards

Placeholder 1st Author

*Department 1*

*Affiliation 1*

City, Country

author1@email.com

Placeholder 2nd Author

*Department 2*

*Affiliation 2*

City, Country

author2@email.com

**Abstract**—Ethereum is a blockchain platform which enables the use of smart contracts. Smart contracts will execute a set of instructions without intermediary party when called upon, this process happens automatically. The possibility to make calls to another smart contract within a contract allows for potential exploits to occur. In this paper we will discuss and look at how one of these exploits, called the reentrancy attack, is possible. This attack is most well known for The DAO attack in 2016 where almost 55 million dollar got drained by an attacker who made use of this vulnerability in the smart contract. More specifically we will look at the concept in the Solidity programming language which was made specifically for the Ethereum blockchain. Also an overview of the different advantages and disadvantages of the different functions to exchange Ether, the currency used to execute transactions, will be given, how they work and how the call function might allow for certain exploits. The reentrancy vulnerability is still prevalent in smart contracts nowadays and forms a huge threat to applications and their users due to the huge possible financial losses that can happen. This research also includes an analysis of a verified smart contract database that was collected from Etherscan. This was done to detect potential vulnerabilities by looking for the presence of functions that allow for these attacks. Finally, the results of this analysis are discussed and there will be a brief discussion about prevention measures and methods to avoid a reentrancy attack.

**Index Terms**—Blockchain, Smart Contracts, Solidity, Reentrancy, Guards.

## I. Introduction

A blockchain is an append-only transactional database where the information is structured together in groups, also known as blocks [1, 2]. Each block has certain storage capacities and is chained onto the previous filled block, thus forming a blockchain. Another way to define it is a shared, immutable ledger that records transactions which can be used to track

different assets. Most notable uses of this blockchain technology are the cryptocurrencies Bitcoin[3] and Ether [4, 5] on the Ethereum network. One important difference between these two blockchain platforms is that Ethereum enables the deployment of smart contracts.

A smart contract is a contract which executes automatically when called upon where the terms between the two parties is written in code (on the blockchain). These contracts then run when a function is called and the conditions for that function are met and can be used to automate executions of agreements without any intermediary party [2, 6, 7]. In their simplest form a contract is just a collection of functions. Interesting to note is that all smart contract transactions are traceable, transparent and also irreversible [1, 2].

A common functionality of smart contracts is the possibility to make calls to another contract on the same blockchain platform. This however needs to be done with caution as untrusted contracts can not only introduce errors but also risks as the contract or call may execute malicious code and exploit vulnerabilities. Every call transfers execution control to the called contract.

One of these dangers when calling an external contract is called Reentrancy and is one of the most well known attacks due to the DAO Attack on June 17th 2016 where around 3.6 million Ether was taken which equated to around \$50 million dollar at the time [8]. This exploit is cemented in the history of Ethereum as it resulted into Ethereum being forked into Ethereum Classic and the Ethereum we know today. The original version of this attack involved functions that would be called repeatedly before the first function was finished.

Solidity is one of the major programming languages for smart contracts on Ethereum. To avoid these ex-

exploits there have been introduced some best practices. More specifically the function call() was to be replaced by the more safe functions transfer() and send(). However, recently there has been a switch back to the call() function with the introduction of EIP 1884 [9]. Other precautions instead must be taken to prevent reentrancy attacks, one recommendation is making use of safe code patterns and using guard constructs.

In this paper, we will investigate how these calling functions are used in practice and if they are still commonly used for the current smart contracts being deployed in the Ethereum network. We collected a dataset of 26,800 unique open-source verified smart contracts from Etherscan (from 2012-07-07 to 2022-01-06).

## II. Background

An introduction to some important concepts such as blockchain, the consensus mechanism used in Ethereum, smart contracts and reentrancy attacks is given. The scope of these concepts will be kept to the Ethereum blockchain and one of its programming languages Solidity [10].

### A. Ethereum & Smart contracts

Ethereum differs from Bitcoin in that it enables the deployment of smart contracts and decentralized applications also known as dApps with built-in economic functions. While bitcoin its primary focus is to be a digital currency payment network, Ethereum is designed to be a general-purpose programmable blockchain that runs a virtual machine capable of executing code. Ether is the currency used to complete transactions on the network and is used as a way to meter and constrain execution resource costs. In comparison to bitcoin, Ether is designed to be a utility currency which is used to call transactions on the Ethereum platform as a sort of fee [11]. Smart contracts can be used to create a range of dApps. An important feature of smart contracts is that when a function is called and the conditions for that function are met there is an automatic execution of the set agreements without any intermediary party [4, 5].

### B. Solidity

Solidity [10] is one of the main languages to code smart contracts in the Ethereum platform. Solidity is an object-oriented, high-level language which has syntax comparable to C++.

1) *Solidity Ether Exchange*: We like to highlight the language constructs use to exchange cryptocurrency among contracts: call, send, and transfer (Table I).

The call function is a low-level interface for sending a message to a contract and it is also a way to send Ether

TABLE I  
Solidity Functions to Exchange Ether

Function	Gas Limit	Error Handling
call.value	Custom	Returns false on failure
transfer	2300	Throws exception on failure
send	2300	Returns false on failure

to another address. The call function transfers the execution control to the called contract and the caller can forward any amount of gas. Therefore, the call function has the potential to introduce vulnerabilities, most notably reentrancy.

The transfer method was first introduced in the version 0.4.10 (May 2017) of the Solidity language. It provides a safe-by-design method to transfer cryptocurrency. Even though, this method also transfers the execution control to the caller, it has a gas limit which prevents abuse. If the transfers fails, an exception is raised, which also adds to the security of this method as the exception reverts the transaction. Due to automatically reverting in case of errors, the transfer function is recommended in most cases.

The send function can be seem as a lower level implementation of transfer. Similar to transfer, it provides a safe-by-design function to transfer cryptocurrency, with a gas limit to prevent exploits. The major difference between send and transfer, is that send returns false if it fails, delegating the error handling to the developer.

2) *Solidity Guards*: Guards are language constructs to prevent access or revert a transaction. In Solidity, different guards have been introduced to the language over different versions such as Require, and Assert.

### C. Reentrancy attack

The call function has some vulnerabilities. Every call to another contract transfers execution control to the called contract. Untrusted contracts may introduce and execute malicious code or exploit vulnerabilities. One of these major vulnerabilities is called the reentrancy attack, which takes advantage of the transfer of execution control by making recursive calls back to the original contract and repeating executions and creating new transactions.

The two main types of reentrancy attacks are single function and cross-function: Single function, and Cross-function.

1) *Single function reentrancy attack*: This version repeatedly calls the involved function before the first invocation of the function is finished. Listing 1 shows a code snippet with this exploit.

```
1 mapping (address => uint) private userBalances;
```

```
2
```

```

3 function withdrawBalance() public {
4     uint amountToWithdraw = userBalances[msg.sender];
5     (bool succes, ) = msg.sender.call.value(
6         amountToWithdraw)("");
7     require(succes);
8     userBalances[msg.sender] = 0;
9 }
10 // Fallback function which gets executed
11 function () public payable {
12     withdrawBalance()
13 }

```

Listing 1. Single function reentrancy attack

In this example an attacker can recursively call the `withdrawBalance()` function and drain the whole contract as the user's balance is only set to 0 at the very end of the function.

2) *Cross-function reentrancy attack*: When a function shares a state with another function there is a possibility of a cross-function reentrancy attack. Listing 2 shows a code snippet with a cross-function reentrancy vulnerability.

```

1 mapping (address => uint) private userBalances;
2
3 function transfer(address to, uint amount) {
4     if (userBalances[msg.sender] >= amount) {
5         userBalances[to] += amount;
6         userBalances[msg.sender] -= amount;
7     }
8 }
9 function withdrawBalance() public {
10     uint amountToWithdraw = userBalances[msg.sender];
11     (bool succes, ) = msg.sender.call.value(
12         amountToWithdraw)("");
13     require(succes);
14     userBalances[msg.sender] = 0;
15 }

```

Listing 2. Cross-function reentrancy attack

Here the attacker will call the `transfer` function when the code is executed on an external call in `withdrawBalance()`, again the user's balance is not yet set to 0 and thus they will be able to transfer tokens again. A simple solution to both these types of attacks is updating the balance before transferring control to another function or contract. Another simple solution would be to use `transfer` or `send` (the safer-by-design constructs) instead of `call`.

### III. Study Design

#### A. Dataset

A database of verified smart contracts was collected from Etherscan which is a block explorer and analytics platform for Ethereum. Verifying a contract is a trivial task on etherscan but it allows the public to audit and read the contract as it has to be made publicly available to be granted the verified status. Etherscan does not give access to a complete dataset of verified smart

contracts but rather has an open source database of the latest 10 000 smart contracts that were verified. This open source database was collected at different times during a period of 7 months to create the database that was used for the analysis starting at the 7th of July 2021 and ending January 6th of 2022. The first step we will do is remove all duplicate contracts and as a final pre-processing step which will only be possible after collecting the compiler version, all vyper contracts will be removed as the focus of this paper is on the Solidity language only. After this step we are left with a total of 28,209 contracts. The next step we do is removing all contracts that have 0 LOC as these contracts will not provide any value to our analysis. After removing those contracts we are left with a total of 26,800 available unique verified smart contracts. A more specific overview of the amount of contracts per month can be found in the chart 1. This chart gives an overview of all the unique contracts that were collected per month.

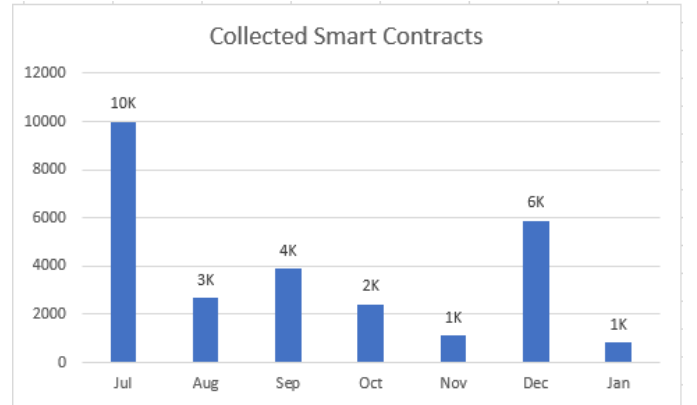


Fig. 1. An overview of the amount of contracts collected each month. The collection started in July which is the reason for the initial 10,000 contracts. Interesting to mention is that starting mid-september, Etherscan has reduced the available verified smart contracts to download to only 5000. The small collection of January is due to only retrieving the database once at the start of the month.

#### B. Method

Using the available API it is now possible to retrieve all source codes for each contract. To retrieve information from the Etherscan API an API key is needed which can be made on Etherscan, after that we can use following API endpoint [3] for each contract address.

```

1 https://api.etherscan.io/api
2 ?module=contract
3 &action=getsourcecode
4 &address=0xaddress // The contract address
5 &apikey=YourApiKeyToken // Your API key

```

Listing 3. Api call

Calling this will give a json response with following information from which we collect all source codes,

contract names and the compiler version. For a more indepth explanation of how the API works we refer to the documentation made available on Etherscan [22].

```

1 {
2   "status": "1",
3   "message": "OK",
4   "result": [
5     {
6       "SourceCode": "Source code"
7       "ABI": "[The ABI Interface]"
8       "ContractName": "The Contract
9       Name",
10      "CompilerVersion": "The Compiler
11      Version",
12      "OptimizationUsed": "1",
13      "Runs": "200",
14      "ConstructorArguments": "
15      Constructor arguments",
16      "EVMVersion": "Default",
17      "Library": "",
18      "LicenseType": "",
19      "Proxy": "0",
20      "Implementation": "",
21      "SwarmSource": ""
22    }
23  ]
24 }

```

#### IV. Analysis and Results

The aim of these experiments will be to have a preliminary analysis of the contracts and to detect potential reentrancy threats where the call function is present. We start with a total of 26,800 unique verified smart contracts.

The first experiment is to take a look at the lines of code of these verified smart contracts and gather some statistics.

After an initial calculation we get the following values for the amount of lines of code, we notice their is a huge standard deviation in comparison with the median.

- Average: 739
- Median: 547
- Standard deviation: 678
- Min LOC: 1
- Max LOC: 9452

Another piece of data we will take a look at are the different versions of Solidity used in each contract. We plot the 30 most used versions in the following bar chart Figure 2 and also plot these versions by only looking at the major version 3. We can notice that three versions are used in more than half of the database

namely v0.6.12, v0.8.4 and v0.8.7. The specific amount of contracts for each version and the percentage of the database that have that contract version can be found in Table III

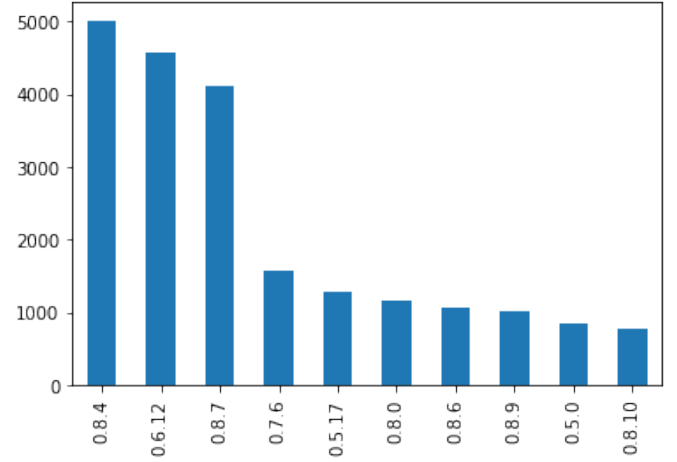


Fig. 2. An overview of the top 10 solidity compiler versions used and the amount of contracts in the database that use that specific version.

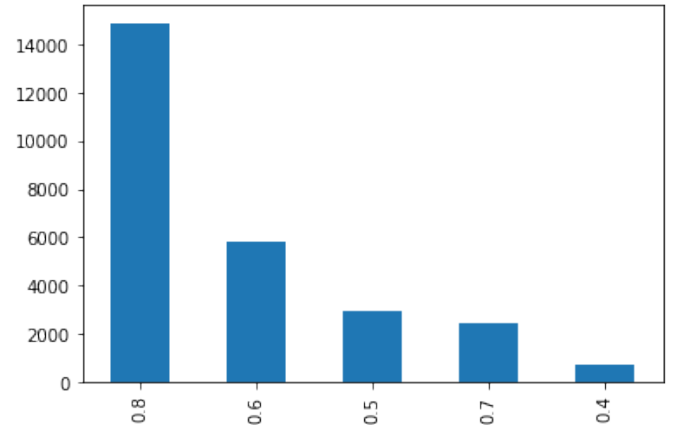


Fig. 3. An overview of the top solidity compiler versions used and the amount of contracts in the database that use that version, where we go into a higher level and only look at the major version.

We will now dive deeper into the source contracts themselves and have a look at the different functionality, more specifically we will count the amount of times we find a call, send, and transfer function which as mentioned earlier are the main methods to exchange Ether in a Solidity smart contract. We will also calculate the amount of times a require, assert and throw function has been called in order to see how many of this contracts make use of guards. Table IV shows us the different statistics for the collected database. We notice that almost all contracts have some form of a guard pattern with a require function. Only 270

TABLE II  
Compiler Versions

Solidity Version	# Contracts	% Percentage
0.8.4	5015	18.71%
0.6.12	4587	17.12%
0.8.7	4103	15.3%
0.7.6	1564	5.84%
0.5.17	1278	4.77%
0.8.0	1166	4.35%
0.8.6	1060	3.96%
0.8.9	1008	3.76%
0.5.0	857	3.20%
0.8.10	768	2.87%
Other	5394	20.13%

TABLE III  
Compiler Versions (Major)

Solidity Version	# Contracts	% Percentage
0.8	14869	55.48%
0.6	5839	21.79%
0.5	2954	11.02%
0.7	2454	9.16%
0.4	684	2.55%

contracts actually have a `call.value` function which shows us that they may have a potential reentrancy attack.

The next experiment will focus more on those contracts that contain a `call.value` function, all previous statistics will be recalculated for only the smart contracts containing a `call.value` function to see if there are any similarities but first we will take a look at which version is used most in these contracts. Figure 4. The most used version is v0.5.17 which was the 5th most used version in all our contracts. A more interesting finding is that there are only 4 contracts above v0.6 that contain a call function of which 3 are from the compiler version that was most used by the complete database namely v0.8.4. For the line of code statistics we get the following values:

- Average: 1050

TABLE IV  
Statistics

Function	# Contracts	# Functions
Call	270	384
Send	9176	17,814
Transfer	647	1059
Require	26,191	622,701
Assert	7279	10,507
Throws	4174	10,590
Function	Avg/contract	
Call	0.014	
Send	0.665	
Transfer	0.0396	
Require	23.235	
Assert	0.392	
Throws	0.395	

TABLE V  
Statistics of Contracts containing call function

Function	# Contracts	# Functions
Send	153	495
Transfer	18	51
Require	270	11932
Assert	192	333
Throws	163	324
Function	Avg/contract	
Send	1.83	
Transfer	0.19	
Require	44.19	
Assert	1.23	
Throws	1.2	

- Median: 761
- Standard deviation: 1079
- Min: 31
- Max: 6535

We notice that these smart contracts on average are of a relatively bigger size in comparison with a regular contract. Table V will now show us different statistics for this specific selection of contracts that only contain a `call.value` function. In this sub selection of contracts a call function is called about 1.4 times on average in a contract and a total of 384 call functions were called in 270 contracts with the highest amount of calls in a single contract being 14. An interesting thing to note is that the average amount of guards is higher for these contracts, this shows us that a guard pattern has been potentially used in cases where a `call.value` function has been used and are signs that most of these contracts are probably correctly implemented.

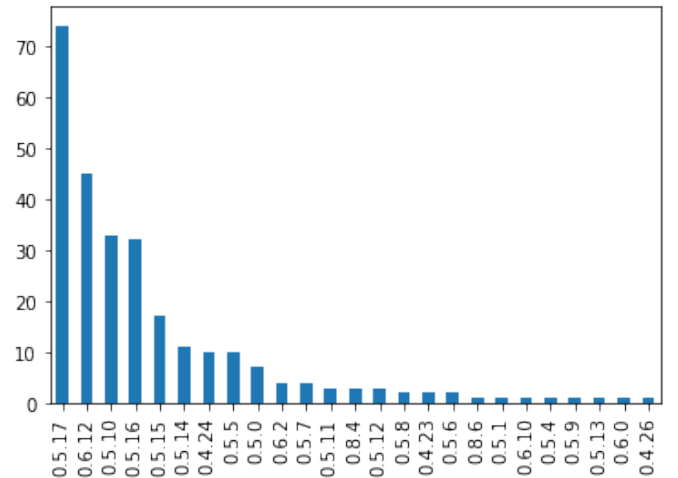


Fig. 4. An overview of all the specific compiler versions used in contracts that contain a `call.value` function.

We will do a similar analysis on both the `send` and `transfer` function. Looking at the versions in Figure 5 we can see that for the `send` function the top 5 versions

are completely similar to the top 5 versions of the whole contract database. This is partly due to the fact that 66% of the contracts of our whole database have a send function. If we look at the versions of transfer in Figure 6 we notice that both v0.8.4 and 0.6.12 are not even in the top 10 versions which is interesting. A version that also showed up in the top 10 of call versions namely v0.4.24 shows up here. This signifies that both call and transfer functions were used more often in v0.4 and that since then there was a shift to the send function.

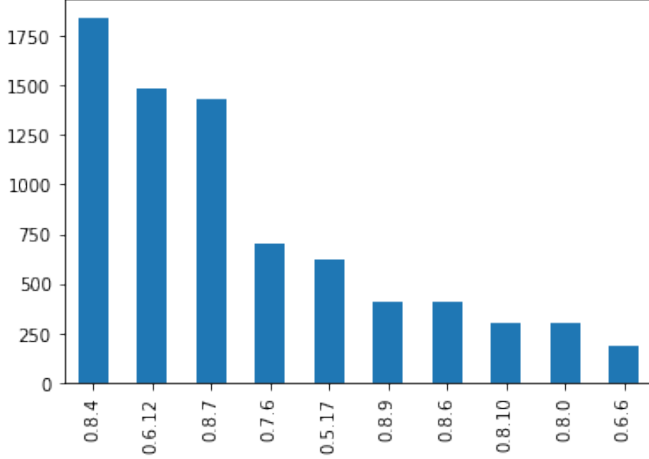


Fig. 5. An overview of all the specific compiler versions used in contracts that contain a send function.

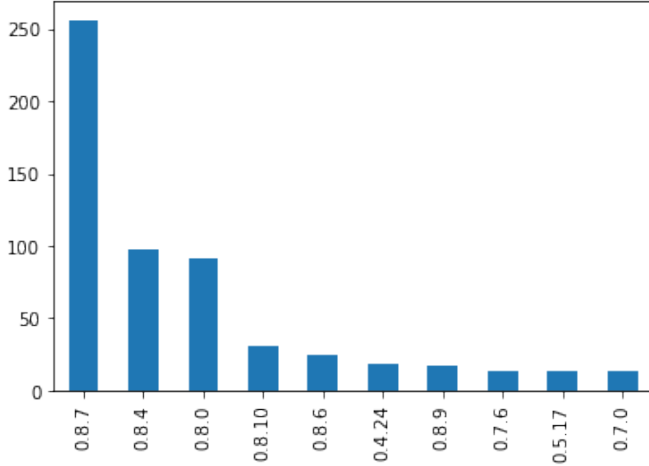


Fig. 6. An overview of all the specific compiler versions used in contracts that contain a transfer function.

If we calculate the LOC statistics for both contracts containing send functions and contracts containing transfer functions we get the statistics in Table VI. Contracts containing a transfer function seem to be a lot longer on average (almost twice as long) in comparison with the average length of a smart contract. A contract

TABLE VI  
LOC Statistics for send and transfer

Statistic	Send	Transfer
Average	832	1453
Mean	553	1378
Std Dev	741.78	841.18
Min	10	20
Max	9452	9452

TABLE VII  
Statistics of Contracts containing send function

Function	# Contracts	# Functions
Call	153	226
Transfer	107	251
Require	9060	256835
Assert	2722	4789
Throws	1507	4338
Function	Avg/contract	
Call	0.0246	
Transfer	0.0274	
Require	27.9898	
Assert	0.5219	
Throws	0.4727	

containing a send function is pretty similar in length in comparison with the average smart contract.

Of all contracts containing a send functions there are almost on average two send functions per contract with the highest amount of send functions used in any contract being 68. This is by far the most used way of transferring Ether in the database that we collected. The transfer function is present in a lot less contracts but when it is used, the average of times it is in a contract is around 1.63 times with the highest amount being 12. We will now look at the different functions present in both cases in a similar way we did for the call function for which the results can be found in Table VII and Table VIII. For the contracts containing a send function we notice that they almost always include atleast one require guard as well. The average of require guards used in a contract containing a transfer function is however higher in comparison with the average of contracts that contain a send function. We can also see that the average of both assert guards and throws are higher for contracts containing a send function than the transfer function. If we now compare some of these statistics to the numbers of the contracts containing a call function we immediately see that those contracts have a lot more guards. These due to the fact that both transfer and send are considered safer functions and thus a call function by design will need to utilize more guards for it to not be exploited.

## V. Related work

Juels, Kosba and Shi[6] investigate the risk of smart contracts fueling new criminal ecosystems . They show

TABLE VIII  
Statistics of Contracts containing transfer function

Function	# Contracts	# Functions
Call	18	20
Send	107	234
Require	647	25058
Assert	63	166
Throws	53	224
Function	Avg/contract	
Call	0.0309	
Send	0.3616	
Require	38.7295	
Assert	0.2566	
Throws	0.3462	

how a Criminal Smart Contract can facilitate leakage of confidential information, theft of cryptographic keys and more, showing the urgency of creating safeguards against these CSCs. They look at questions like how practical these new crimes will be, whether these CSCs enable a wider range of new crimes in comparison to earlier cryptocurrencies such as Bitcoin and what advantages they offer to criminals in comparison with the conventional online systems.

Luu et al. [2] also investigate and introduce several security problems to manipulate smart contracts in an attempt to gain profit and propose ways to enhance the operational semantics of Ethereum. A focus is put on the semantic gap between the assumption contract writers make about the underlying execution semantics and the actual semantics of the contract are made as a reason for these security flaws. A tool OYENTE is also provided to detect bugs which is a symbolic execution tool. The model works directly with Ethereum virtual machine byte code and thus does not have a need for a higher level representation such as Solidity. An evaluation of OYENTE on 19366 smart contracts is given where 8333 contracts were documented as potentially having bugs.

Mense and Flatscher [23] summarize known vulnerabilities found by literature research and analysis such as external calls, gasless sends, mishandled exceptions and reentrancy. They also compare code analysis tools for their ability to identify vulnerabilities in smart contracts based on a taxonomy for vulnerabilities. The results of their paper show that reentrancy ranks the highest among the vulnerabilities that they have discussed and is detected by most of the tools used. They then delve deeper into the DAO hack as well.

Liu et al. [24] present ReGuard which is a fuzzing-based analyzer to automatically detect reentrancy bugs in Ethereum smart contracts. They iteratively generate random (but diverse) transactions, this is called fuzz testing. Then based on the runtime they will identify reentrancy vulnerabilities in a contract. How the archi-

ture works is they parse a smart contracts source or binary code to an intermediate representation which will then be transformed to C++, keeping the original behavior. Together with a runtime library, ReGuard executes the contract and runs an analysis of the operations for any reentrancy attacks.

SmartCheck is an extensible static analysis tool to detect code issues in Solidity by Tikhomirov et al.[25] where they translate Solidity into an XML-based representation and check it against XPath patterns. They also used a real world dataset to evaluate their tool and also make a comparison to the earlier mentioned Oyente.

Samreen and Alalfi [26] explain eight vulnerabilities by looking at past exploitation case scenarios and review some of the available tools and applications to detect these vulnerabilities. For each case they discuss the vulnerability exploited, the tactic used as well as the financial loss what happened. A coverage is given of some preventive techniques as protection against some of these exploits. The tools/frameworks they discuss adopt either a form of static analysis such as symbolic execution and control flow graph construction or dynamic analysis such as the fuzzing testing or tracing the sequence of instructions that are executed at run time.

Tantikul and Ngamsuriyaroj [27] investigate a more recent state of the vulnerabilities of smart contracts. Their research consists of going through a database of verified smart contracts and checking common occurrences as well as trends of vulnerabilities. An analysis is done using both Oyente and Smartcheck and common characteristics of vulnerable smart contracts are identified. A correlation computation is done via Pearson's correlation in order to detect how often any pair of vulnerabilities will be found on the same smart contract. Their results show that overflow and underflow have the highest correlation. Another relation found is the timestamp dependency and transaction ordering which might be caused by malicious miners.

Bragagnolo, Rocha, Denker and Ducasse [28] address the lack of inspectability of a deployed smart contract. They do this by analyzing the state of the contract using different decompilation techniques. Their solution SmartInspect is an inspector based on plug-gable property reflection. Their approach of utilizing mirrors generated from an analysis of Solidity source code allows access to unstructured information from a deployed smart contract in a structured way. This can be done without a need to redeploy or develop additional code for decoding.

Wang et al. [29] evaluate a set of real-world smart contracts with ContractWard which uses machine



learning techniques to detect vulnerabilities in smart contracts. Their idea was proposed due to existing detection methods being mainly based on symbolic execution or analysis which are very time-consuming. The system extracts dimensional bigram features from simplified operation codes to construct a feature space and is able to get a predictive recall and precision of over 96% based on their dataset of 49502 smart contracts on 6 vulnerabilities.

A deep-learning based approach is used by Qian et al. [30]. The aim is to precisely detect reentrancy bugs using a bidirectional long-short term memory with attention mechanism. They also propose using a contract snippet as another way to represent a smart contract only capturing key semantic sentences which contain related and critical information such as control flow and data dependencies. These are then used as input to the sequential models. They show that this deep-learning approach outperforms other state-of-the-art smart contract vulnerability tools.

Slither by Feist et al. [31] is a static analysis framework that converts Solidity smart contracts into an intermediate representation which they call SlithIR. Static Single Assignment forms are used as well as a reduced instruction set for ease of implementation. Their framework has use cases in automated detection of vulnerabilities, detection of code optimization opportunities, improvement of clarity and ease of understanding of the contracts. An evaluation of the proposed frameworks capabilities is done using a set of real-world smart contracts.

## References

- [1] S. Bragagnolo, H. S. C. Rocha, M. Denker, and S. Ducasse, "SmartInspect: Solidity Smart Contract Inspector," in *IWBOSE 2018 - 1st International Workshop on Blockchain Oriented Software Engineering*. Campobasso, Italy: IEEE, Mar. 2018. [Online]. Available: <https://hal.inria.fr/hal-01831075>
- [2] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: <https://doi.org/10.1145/2976749.2978309>
- [3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
- [4] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," pp. 1–39, 06 2018.
- [5] E. Foundation, "Ethereum's white paper," 2014. [Online]. Available: <https://ethereum.org/en/whitepaper/>
- [6] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Investigating the future of criminal smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 283–295. [Online]. Available: <https://doi.org/10.1145/2976749.2978362>
- [7] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguélin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–96. [Online]. Available: <https://doi.org/10.1145/2993600.2993611>
- [8] 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.
- [9] M. H. Swende. (2019) Eip-1884: Repricing for trie-size-dependent opcodes. [Online]. Available: <https://eips.ethereum.org/EIPS/eip-1884>
- [10] Ethereum. (2021) Solidity documentation. [Online]. Available: <https://docs.soliditylang.org/en/v0.8.11/#>
- [11] A. Antonopoulos, G. Wood, and G. Wood, *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018. [Online]. Available: <https://books.google.be/books?id=SedSMQAACAAJ>
- [12] N. Szabo. (1994) Smart contracts. [Online]. Available: <https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html>
- [13] ethereum.org. (2021) Ethereum development documentation. [Online]. Available: <https://ethereum.org/en/developers/docs/>
- [14] fravoll. (2018) Checks effects interactions pattern. [Online]. Available: [https://fravoll.github.io/solidity-patterns/checks\\_effects\\_interactions.html](https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html)
- [15] —. (2018) Guard check. [Online]. Available: [https://fravoll.github.io/solidity-patterns/guard\\_check.html](https://fravoll.github.io/solidity-patterns/guard_check.html)
- [16] C. DAO. (2021) Constitution dao. [Online]. Available: <https://www.constitutiondao.com/>



- [17] —. (2021) The dao for decentralized asset ownership. [Online]. Available: <https://www.citydao.io/#Section-1>
- [18] V. Buterin. (2021) Crypto cities. [Online]. Available: <https://vitalik.ca/general/2021/10/31/cities.html>
- [19] K. DAO. (2021) Introducing klimadao. [Online]. Available: <https://docs.klimadao.finance/>
- [20] V. Network. (2020) The reentrancy strikes again - the case of lendf.me. [Online]. Available: <https://valid.network/post/the-reentrancy-strikes-again-the-case-of-lendf-me>
- [21] R. news. (2021) Cream finance - rekt. [Online]. Available: <https://rekt.news/cream-rekt/>
- [22] Etherscan. (2021) Etherscan api knowledge base. [Online]. Available: <https://docs.etherscan.io/api-endpoints/contracts>
- [23] A. Mense and M. Flatscher, "Security vulnerabilities in ethereum smart contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications and Services*, ser. iiWAS2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 375–380. [Online]. Available: <https://doi.org/10.1145/3282373.3282419>
- [24] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 65–68. [Online]. Available: <https://doi.org/10.1145/3183440.3183495>
- [25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 9–16. [Online]. Available: <https://doi.org/10.1145/3194113.3194115>
- [26] N. F. Samreen and M. H. Alalfi, "A survey of security vulnerabilities in ethereum smart contracts," in *Proceedings of the 30th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '20. USA: IBM Corp., 2020, p. 73–82.
- [27] P. Tantikul. and S. Ngamsuriyaroj., "Exploring vulnerabilities in solidity smart contract," in *Proceedings of the 6th International Conference on Information Systems Security and Privacy - ICISSP, INSTICC*. SciTePress, 2020, pp. 317–324.
- [28] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "Smartinspect: solidity smart contract inspector," in *2018 International Workshop on Blockchain Oriented Software Engineering (IW-BOSE)*, 2018, pp. 9–18.
- [29] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2021.
- [30] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang, "Towards automated reentrancy detection for smart contracts based on sequential models," *IEEE Access*, vol. 8, pp. 19 685–19 695, 2020.
- [31] 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.