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 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-

ploits 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 26,799 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).

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

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

```
1 mapping (address => uint) private userBalances;
2
```

```
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender
    ];
    (bool succes, ) = msg.sender.call.value(
    amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
    }
// Fallback function which gets executed
function () public payable {
    withdrawBalance()
}
```

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 snipped with a cross-function reentrancy vulnerability.

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

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
        userBalances[to] += amount;
        userBalances[msg.sender] -= amount;
        }
    }
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender
    ];
    (bool succes, ) = msg.sender.call.value(
    amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
}
```

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

We collected verified smart contracts from Etherscan[1] which is a block explorer and analytic platform for Ethereum. Etherscan verified contracts allows the public to audit and read contracts 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 10,000-5,000 smart contracts that were verified. Therefore, we gathered the latest contracts from time to time, over a period of six months (2021-07-07 to 2022-01-06) to build our dataset.

Then, we did the following pre-processing steps in our dataset: (i) remove all duplicated[2] contracts; (ii) remove contracts not written in Solidity; (iii) removed contracts which we could not process using cloc.[3] After removing those contracts, we had a total of 26,799 unique verified solidity smart contracts. This dataset is publicly available.[4]

B. Method

We use the Etherscan API [22] to retrieve the source codes for each contract in our dataset. Listing 3 shows an example of the API call used to acquire the contract source code.

```
https://api.etherscan.io/api?module=contract
   &action=getsourcecode
   &address=0xb4e32b964f6ae78 //The contract address
   &apikey=YourApiKeyToken // Your API key
```

Listing 3. Etherscan API call

Then, we used the cloc tool on the contracts to discover how many lines of code are in each one. We removed from this study the contracts that cloc were not able to process. Moreover, we used a Python script on the contracts to locate specific methods used in the contracts related to Ether exchange functions (call, send, and transfer) and guards (require, assert, revert).

IV. Analysis and Results

First, we show some general characteristics on our dataset. Table VI shows the general statistics considering the lines of code on the contract. We can see that the contracts in our dataset are small in lines of codes, with an average of 256 LoC and a median of 356 LoC. That is expected, as smart contract code tends to be smaller when compared to software code in other domains. The smallest contracts have only 2 LoC. For example, the contract *BlackHole*[5] only has two lines, a pragma definition for the solidity version,

[1] etherscan.io/contractsVerified

[2] We removed contracts with the same address in the Ethereum blockchain. We did not verified whether the contracts with the same name have the same source code. Since these contracts have different addresses, they are considered separate entities in the blockchain platform.

[3] cloc is a tool to count lines of code available at <https://github.com/AlDanial/cloc>.

[4] https://bit.ly/3fyOgBD, the link has been properly anonymized for the reviewers to download the dataset spreadsheet with the name and address of the contracts and not break the double-blind review process.

[5] https://etherscan.io/address/0x727E9A3067DeEaF031916fA0fC53B02cf44F8731#code

TABLE II
Lines of Code (LoC)

Min	Median	Average	Std. Dev.	Max
2	256	359	335	6,461

TABLE III
Solidity Major Versions

Solidity Version	# Contracts	Percentage
0.8.x	14,869	55.48%
0.6.x	5,838	21.78%
0.5.x	2,954	11.02%
0.7.x	2,454	09.16%
0.4.x	684	02.55%

and an empty contract definition. The biggest contract is *RewardControl*[6] with 6,461 LoC.

Table III shows the contracts categorized by their major Solidity version. Most contracts are from the latest Solidity version, 0.8.x. The oldest Solidity version to appear in our dataset is 0.4.x which has the fewer amount of contracts.

Figure 1 shows the top-10 solidity versions (major and minor) in our contracts. The versions with most contracts are 0.8.4 (18.7%), 0.6.12 (17.1%), and 0.8.7 (15.3%). Together these three versions compose over 50% of the contracts in our dataset.

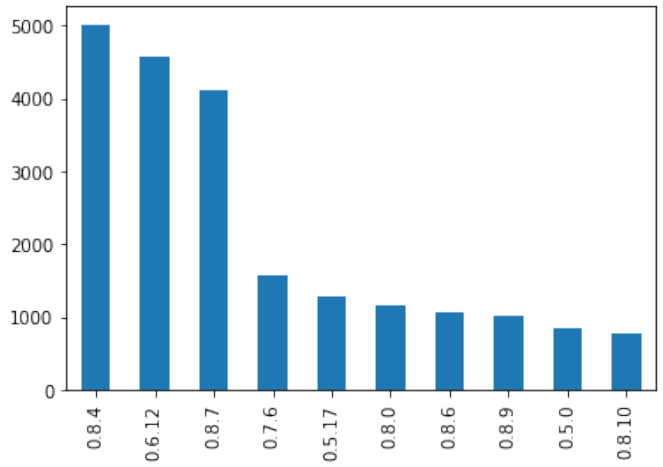

Fig. 1. Top-10 Solidity versions in the database.

Now, we investigate how many Ether exchange methods and guards are being used in the contracts. Table IV shows how many contracts in our dataset has at least one of the methods, the overall count of the method, and the average number in the contracts that have at least one. Even though, call is the unsafest method for Ether exchange, it is used by 50% of the

[6]https://etherscan.io/address/0xcf8Fe5bB819359Ea02DF65E50B6194D12b69aB88#code

TABLE IV
Ether Exchange and Guards Usage

Method	Contracts	Count	Average
Call	13,443 (50%)	32,236	2.3
Send	9,176 (34%)	17,814	1.9
Transfer	647 (02%)	1,059	1.6
Require	26,190 (97%)	622,679	23.7
Assert	7,279 (27%)	10,507	1.4
Revert	13,819 (51%)	41,502	3.0

contracts in the dataset. On average, there are 2.3 call uses on the contracts. Any contract using call have the potential to have a reentrancy vulnerability. On the other hand, transfer is supposed the be the safest method and it is the least used (2%). Send is also a safe-by-design function, and it is used by roughly one-third of the contracts.

On the guard methods shown in Table IV, require used by the great majority of all contracts (97%). On average, there are 23.7 uses of require in the contracts. The great usage of this guard may be to counteract the vulnerabilities of call. Revert is also commonly used by more then half of the contracts in our dataset. Finaly, Assert is the least used guard in our analysis.

A. Call

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 2. 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: 511
- Median: 530
- Standard deviation: 367
- Min: 6
- Max: 5572

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

TABLE V
Statistics of Contracts containing call function

Function	# Contracts	# Functions
Send	4514	8789
Transfer	582	920
Require	13,392	456,461
Assert	5348	6701
Throws	3602	8377
Function	Avg/contract	
Send	0.65	
Transfer	0.07	
Require	33.96	
Assert	0.50	
Throws	0.62	

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.

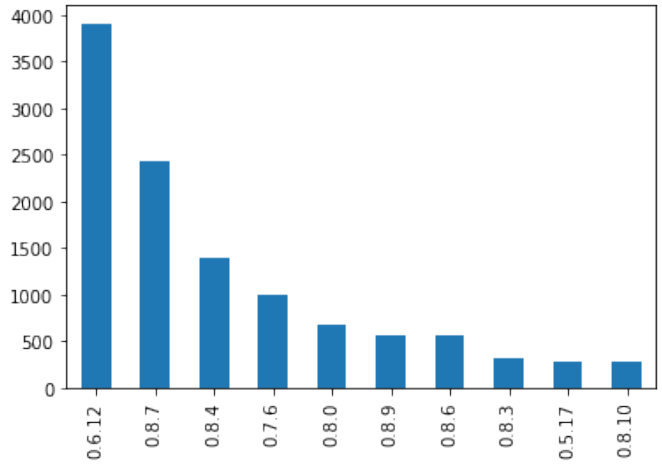

Fig. 2. An overview of all the specific compiler versions used in contracts that contain a call.value function.

We will do a similar analysis on both the send and transfer function. Looking at the versions in Figure 3 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 4 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.

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

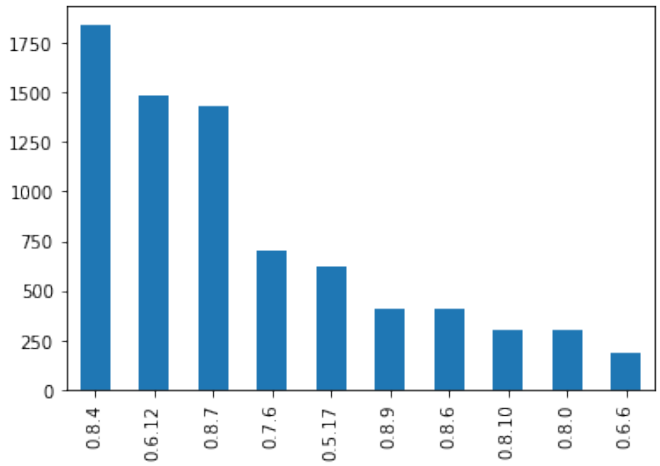

Fig. 3. An overview of all the specific compiler versions used in contracts that contain a send function.

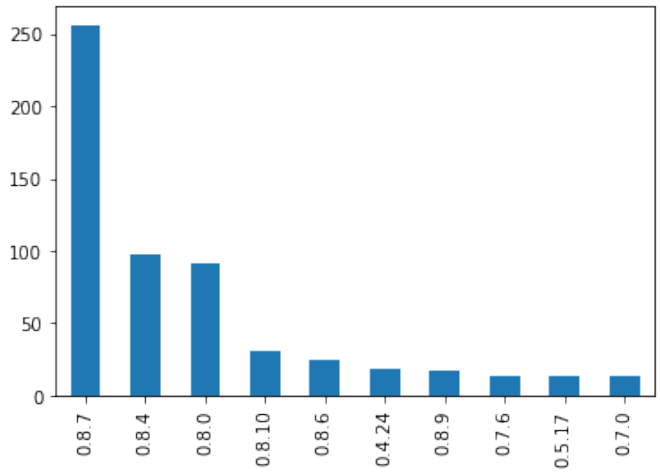

Fig. 4. An overview of all the specific compiler versions used in contracts that contain a transfer function.

longer on average (almost twice as long) in comparison with the average length of a smart contract. A contract 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

TABLE VI
LOC Statistics for send and transfer

Statistic	Send	Transfer
Average	452	635
Mean	345	575
Std Dev	373	498
Min	6	15
Max	6461	6461

TABLE VII
Statistics of Contracts containing send function

Function	# Contracts	# Functions
Call	4514	11083
Transfer	107	251
Require	9060	256835
Assert	2722	4789
Throws	1507	4338
Function	Avg/contract	
Call	1.20	
Transfer	0.03	
Require	27.99	
Assert	0.52	
Throws	0.47	

TABLE VIII
Statistics of Contracts containing transfer function

Function	# Contracts	# Functions
Call	582	1203
Send	107	234
Require	647	25058
Assert	63	166
Throws	53	224
Function	Avg/contract	
Call	1.86	
Send	0.36	
Require	38.73	
Assert	0.26	
Throws	0.35	

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 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 aswell.

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 architecture 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 contact. They do this by analyzing the state of the contract using different decompilation techniques. Their solution SmartInspect is an inspector based on pluggable 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 aswell 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.

VI. Conclusion

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-Béguelin, "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

[15] ——. (2018) Guard check. [Online]. Available: 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 amp; 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 Proceeedings*, 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 (IWBOSE)*, 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.