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 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 26,799 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).

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.

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 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.

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

```

```
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

A. Overall Analysis

First, we show some general characteristics on our dataset. Table II 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

[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%

two lines, a pragma definition for the solidity version, 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.

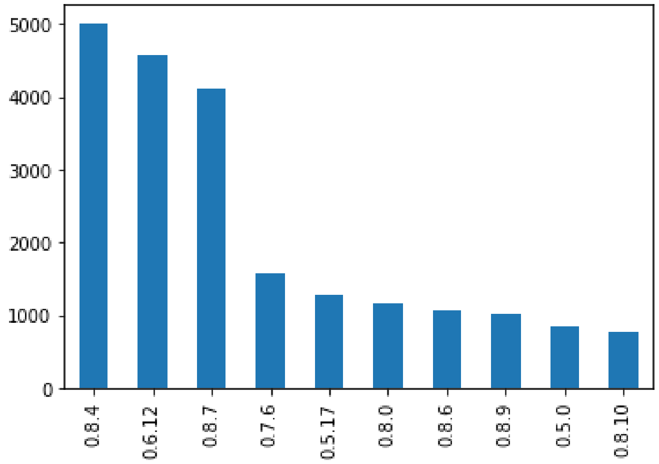

Fig. 1. Top-10 Solidity versions in the database.

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 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, send which is a safer method and it is the least used (2%). Trasnfer is the safest method, and it is used by roughly one-third of the contracts.

TABLE IV
Ether Exchange and Guards Usage

Method	Contracts	Count	Average
Call	13,443 (50%)	32,236	2.3
Transfer	9,176 (34%)	17,814	1.9
Send	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

On the guard methods shown in Table IV, require is 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.

B. *Contracts using Call*

We focus only on the contracts that contain a call function. Since call is a very unsafe function, there is the possiblitiy for the contracts using it to suffer from a reentrancy vulnerability. For this reason, such contracts may have different characteristics. Table V shows the lines of code considering only the contracts that contain a call function. The average and median LoC values are higher than the ones for all contracts. Therefore, the contract with call in our dataset usually have more lines of code. The call contract with the least lines of code is called *FlashBotLowGas*.[7]

Figure 2 shows the Solidity versions for contracts with call. The version with most contracts using call is 0.6.12. The top-3 versions for all contracts are also the top-3 for contracts with call but in different positions.

Table VI show the Ether exchange and guard methods considering only the 13,443 contracts that contains at least one call method. We can see that approximately one-third of the contracts using call also use the send function. We also like to highlight that there is a

TABLE V
Contracts using Call - Lines of Code

Min	Median	Average	Std. Dev.	Max
6	530	511	367	5,572

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

[7] https://etherscan.io/address/0x90ab9a926a1593992547e0f9a0df6401f10421cd#code

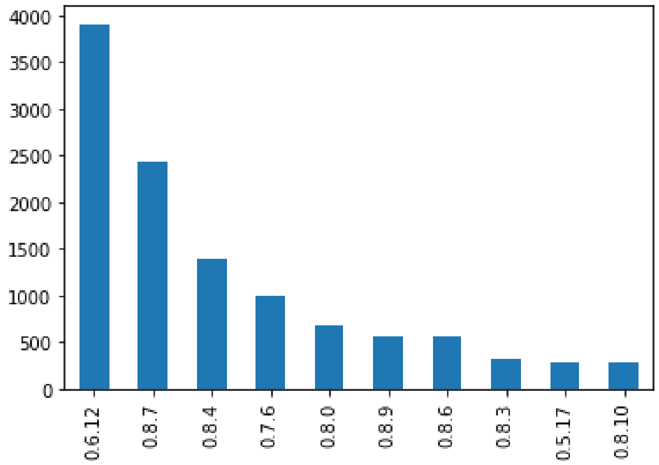

Fig. 2. Top-10 Solidity versions on Contracts using Call.

TABLE VI
Solidity Methods of Contracts using Call

Method	Contracts	Count	Average
Transfer	4,514 (33%)	8,789	0.65
Send	582 (04%)	920	0.07
Require	13,392 (99%)	456,461	33.96
Assert	5,348 (39%)	6,701	0.49
Revert	11,976 (89%)	38,273	2.84

greater number of contracts with call using guards. For isntance, 99% of the call contracts used Require compared to 97% of all contracts; 39% of the call contracts used Assert compared to 27% of all contracts; and 89% of the call contracts used Revert compared to 51% of all contracts. The higher usage of guards is probably to counter the vulnerabilities of call. This may be an indication that Solidity developers are concerned about the security of their contracts specially when using unsafe methods such as call.

C. Contracts using Transfer

Now, we focus only on the contracts that contain a transfer function. Transfer is much safer alternative than call for Ether exchange. Therefore, we expect the contracts using transfer to have different characteristics than the ones using call.

Table VII shows the lines of code only from contracts with a transfer function. The median, average, and standard deviation are higher than the ones when considering all contracts. However, the same statistics are lower when compared to the contracts with call. This means that contracts using transfer tend to have lower LoC than the contracts using call. The reason for this difference could be that call will need extra code to protect against vulnerabilities. Since transfer is a safe-by-design function, it will not need as much extra code as call for a more secure contract. The smallest LoC contract using transfer is *TransferValueToMinerCoinbase*[8] with 6 LoC.

TABLE VII
Contracts using Transfer - Lines of Code

Min	Median	Average	Std. Dev.	Max
6	345	452	373	6,461

TABLE VIII
Solidity Methods of Contracts using Transfer

Methods	Contracts	Count	Average
Call	4,514 (49%)	11,083	1.20
Send	107 (01%)	251	0.03
Require	9,060 (98%)	256,835	27.99
Assert	2,722 (29%)	4,789	0.52
Revert	4,876 (53%)	14,493	1.57

Figure 3 shows the solidity versions for contracts that contain transfer. The top-5 versions for contracts using send are the same top-5 versions for all contracts in our dataset. This could indicate that the contracts with send may represent a general set similar to all contracts in our dataset.

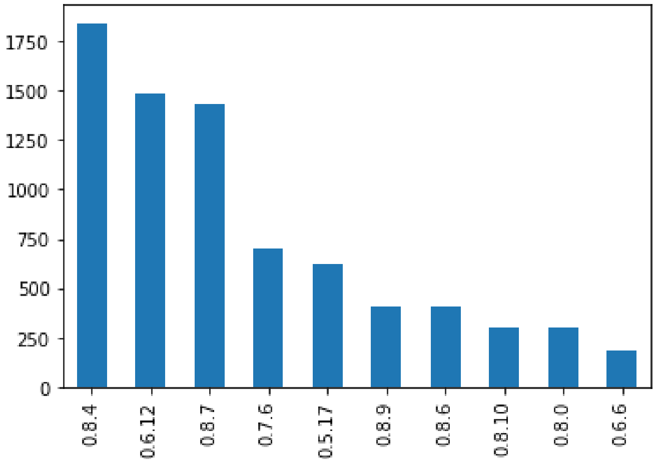

Fig. 3. Top-10 Solidity versions on Contracts using Transfer.

Table VIII show the Ether exchange and guard methods considering only the 9,176 contracts that contains at least one transfer method. The percentage of contracts when looking at only contract with transfer is similar (with a 1-2% difference) from the ones considering all contracts. This is different from the contracts with call where the guards percentage increases by a noticeable amount for Assert and Revert.

D. Contracts with Send

We analyze only the contracts using at least one send method. In our dataset, send is the least used method,

[8] https://etherscan.io/address/0x8512a66d249e3b51000b772047c8545ad010f27c#code

TABLE IX
Contracts using Send - Lines of Code

Min	Median	Average	Std. Dev.	Max
15	575	635	498	6,461

TABLE X
Solidity Methods of Contracts using Send

Method	Contracts	Count	Average
Call	582 (89%)	1,203	1.86
Transfer	107 (16%)	234	0.36
Require	647 (100%)	25,058	38.73
Assert	63 (09%)	166	0.26
Revert	588 (90%)	2304	3.56

being present in only 647 (approximately 2%) of the contracts. Even though there are fewer contracts to analyze, we expect to obverse different characteristics.

Table IX shows lines of code for contracts with send. The median, average, standard deviation are the highest when compared to all contracts, contracts using call, and contracts using transfer. The contract *PaymentManager*[9] has 2 send functions and it is smallest contract with 15 LoC.

The most common Solidity versions for contracts using send were 0.8.7, 0.8.4, 0.8.0, 0.8.10, and 0.8.6. As we can see, the top-5 versions are all 0.8.x.

Table X show the Ether exchange and guard methods considering only the 647 contracts that contains at least one send function. The contracts percentage are different when contrasted with all contracts. For instance, 89% of send contracts have a call function compared to 50% of all contracts; 90% of send contracts have at least one revert compared to 51% of all contracts; and in the oppositive direction, 16% of send contracts have at least one transfer method compared to 34% of all contracts.

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

[9] https://etherscan.io/address/0xaddeb5dbdc1c62c2a2a8e04fddd42e3c3f19587b#code

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.