Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts

Péter Hegedűs
MTA-SZTE Research Group on Artificial Intelligence
Szeged, Hungary
hpeter@inf.u-szeged.hu

ABSTRACT

The blockchain based decentralized cryptocurrency platforms are one of the hottest topics in tech at the moment. Though most of the interest is generated by cryptocurrency related activities, it is becoming apparent that a much wider spectrum of applications can leverage the blockchain technology.

The primary concepts enabling such general use of the blockchain are the so-called smart contracts, which are special programs that run on the blockchain. One of the most popular blockchain platforms that supports smart contracts are Ethereum. As smart contracts typically handle money, ensuring their low number of faults and vulnerabilities are essential. To aid smart contract developers and help maturing the technology, we need analysis tools and studies for smart contracts.

As an initiative for this, we propose the adoption of some well-known OO metrics for Solidity smart contracts. Furthermore, we analyze more than 10,000 smart contracts with our prototype tool. The results suggest that smart contract programs are short, not overly complex and either quite well-commented or not commented at all. Moreover, smart contracts could benefit from an external library and dependency management mechanism, as more than 80% of the defined libraries in Solidity files code the same functionalities.

KEYWORDS

static analysis, ethereum, smart contracts, metrics, complexity, blockchain

ACM Reference Format:
Péter Hegedűs. 2018. Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts. In *WETSEB'18: WETSEB'18:IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB 2018), May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3194113.3194119

1 INTRODUCTION

Decentralized cryptocurrencies have gained considerable interest and adoption since Bitcoin was introduced in 2009 [8]. Users in a cryptocurrency network run a consensus protocol to maintain and secure a shared ledger of data (the *blockchain*). This means that cryptocurrencies are administered publicly by users in the network without relying on any trusted third parties. Blockchains were initially introduced for peer-to-peer payments, but since than it became clear that blockchain technology can be used for much more than that. One such new use for blockchains is to enable the so-called smart contracts.

In this paper, we focus on the static analysis of smart contracts on the Ethereum network [1, 2]. Ethereum is much more than a cryptocurrency blockchain and protocol. It defines a Turing-complete programming platform and a run-time environment called EVM (Ethereum Virtual Machine). EVM can run the bytecodes of smart contracts. A smart contract is a program that runs on the blockchain and has its correct execution enforced by the consensus protocol [9]. Smart contracts can be written in several programming languages, but Solidity [5], a contract-oriented language is by far the most popular one.

As smart contracts provide an entirely new platform and paradigm for programmers, new tools helping them in code analysis and validation has already started to roll out. Such static analysis tools are the *Manticore*[1] symbolic EVM byte code execution tool, security checker tools like *Mythril*[2] or *Oyente*[3] [6], the *solidity-coverage* test coverage tool[4], or *Solcheck*[5] and *Solint*[6] linter tools. However, to the best of our knowledge, there are currently no tools that would support calculating various static source code metrics for Solidity based smart contracts.

In the beginning of 2018, more than 10% of the jobs advertised on one of the biggest freelancer site[7] was related to smart contracts and blockchain, thus the growing importance of blockchain related programming is obvious. As metrics for other languages play a very important role in various QA activities, we anticipate that the same would be true for Solidity smart contracts. Given the nature of the blockchain based programs, namely that once deployed they cannot

[1] https://github.com/trailofbits/manticore
[2] https://github.com/ConsenSys/mythril
[3] https://github.com/melonproject/oyente
[4] https://github.com/sc-forks/solidity-coverage
[5] https://github.com/federicobond/solcheck
[6] https://github.com/SilentCicero/solint
[7] https://www.guru.com, searched on 17th January, 2018

WETSEB'18, May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5726-5/18/05. . . $15.00
https://doi.org/10.1145/3194113.3194119

be altered anymore, makes it even more crucial to be able to check and validate the code beforehand. TheDAO smart contract is a classic example of how big damage can be caused by a smart contract with critical vulnerabilities. An exploit of a bug [4] in this contract led to a 60 million US dollar loss in June 2016.

If we think of the advanced usage of source code metrics in bug/vulnerability prediction, code review and refactoring or anti-pattern detection in classic OO languages, it is clear that smart contract programing would also benefit from such easy to calculate static source code metrics. As the structure of the Solidity language is quite similar to that of the OO languages, the classic Chidamber & Kemerer [3] metrics can be defined for smart contracts in a quite straightforward manner.

In this paper we propose some well-known static source code metrics for measuring smart contracts' size and complexity attributes. We implemented a prototype tool called SolMet[8] that is able to parse Solidity smart contracts and calculate these metrics on them. To analyze the typical metric landscape of the smart contracts deployed on the Ethereum network, we collected 10,206 smart contracts with validated Solidity source code.

According to the metric results of our prototype tool, we can note some interesting characteristics of smart contract programs. It seems that smart contracts are very short and flat programs without real complexity in terms of McCabe's cyclomatic complexity [7] or nesting level of control structures. They contain few functions on average and are either quite well-commented or not commented at all. Moreover, smart contracts could benefit from an external library and dependency management mechanism, as more than 80% of the defined libraries in Solidity files code the same functionalities (i.e. some kind of safe mathematic operations).

2 SMART CONTRACT ANALYSIS APPROACH AND STUDY

2.1 The Solidity Language

Solidity is a contract-oriented, high-level language for implementing smart contracts. It was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM). Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. Using Solidity, it is possible to create contracts for voting, crowdfunding, blind auctions, multi-signature wallets and more.

2.2 Calculating Metrics for Solidity Programs

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain. Contracts also support constructors, special functions that are run during creation of the contract and cannot be called afterwards. Given the obvious similarities in structure, it is easy to map contracts to classes, states to attributes, and functions to member operations in the OO world and interpret classic OO metrics for Solidity based smart contracts.

In this paper, we propose the following source code metrics for smart contracts:

- SLOC – number of source code lines of the contract, library or interface
- LLOC – number of logical code lines (empty and comment lines are ignored) of the contract, library or interface
- CLOC – number of comment lines of the contract, library or interface
- NF – number of functions in the contract, library or interface
- WMC – the weighted complexity of functions in a contract, library, interface, which is the sum of McCabe's cyclomatic complexities [7] (the number of branching statements + 1) of the functions
- NL – the sum of the deepest nesting level of the control structures within the functions of a contract, library or interface

To calculate these metrics, we implemented a prototype tool called SolMet in Java. For parsing the Solidity source code, we used a generated parser based on a slightly modified version of an existing antlr4 grammar.[9] We made only few adjustments in the antlr grammar to be able to parse older and newer versions of the Solidity language as well. The calculation of metrics is performed by various visitors on the parser built AST. The source code and usage instructions of the SolMet tool is available on GitHub.[8]

2.3 Collecting Ethereum Smart Contracts

To utilize SolMet, we collected 10,206 Solidity smart contracts. We chose not to mine GitHub for smart contract codes, as we wanted to get a picture of smart contracts already deployed on the Ethereum network. For this, we downloaded the validated source code of deployed smart contracts monitored by the Etherscan[10] Ethereum blockchain explorer site. Validated source code means that the functionality of the Solidity source code and the deployed EVM bytecode (the Ethereum network stores only the latter one) is manually compared and validated. Thus, we can be sure that the code we analyze is actually the same as the contract being deployed on the network. We ran SolMet on each Solidity file and collected the calculated source code metrics into a comma-separated file. All the analyzed contract source code (.sol files) as well as the metric results are publicly available on GitHub.[11]

3 ANALYSIS RESULTS

Table 1 summarizes the number of analyzed Ethereum smart contracts. In 10,206 Solidity source code files, we analyzed nearly 45,000 contracts. The average number of contracts, libraries, and interfaces are shown in the second column.

[8] https://github.com/chicxurug/SolMet-Solidity-parser

[9] https://github.com/solidityj/solidity-antlr4

[10] https://etherscan.io/

[11] https://github.com/chicxurug/wetseb-2018-data

Table 1: Statistics of the analyzed contracts

	Total	Avg./sol file
Contract	44,893	4.40
Library	4,260	0.42
Interface	662	0.06

On average, each Solidity source file contains 4.4 contracts and nearly every second file defines a library. An interesting note is that above 80% of these libraries are connected to the same functionality, namely they define safe mathematic operations (e.g. zero division and overflow checks). Interfaces are very rare; roughly, every 20th Solidity source code contains one.

Table 2 displays the descriptive statistics of the calculated source code metrics for all the contracts. Although the standard deviations are quite significant, we can draw some general conclusions.

Table 2: Descriptive statistics of the calculated metrics

	Min	Max	Avg.	Median	Std.dev.
SLOC	1	1,250	60.30	29	100.57
LLOC	1	843	38.28	18	66.07
CLOC	0	481	13.68	2	30.69
NF	0	93	5.11	3	6.43
WMC	0	522	7.58	4	14.10
NL	0	125	1.59	0	4.68
Avg. McCC	1	40.15	1.25	1	0.65
Avg. NL	0	17.86	0.17	0	0.34

On average, contracts are short (not considering empty and comment lines). They are actually either quite well-commented or not commented at all. Comment lines to logical lines ratio is high on average, but we can see that the median value of comment lines is only 2, thus there are many contracts with less than or equal to 2 comment lines. On average, each contract defines 5 functions, but the average weighted complexity is only 7.58. This low complexity is observable on the average McCabe's cyclomatic complexity values over all the functions, which is 1.25. It means that most of the functions use sequential control flows without too much complexity. This finding is strengthened by the nesting level (NL) values, as on average the deepest nesting in control structures is only 0.17. There are very few deeply nested control structures in smart contracts.

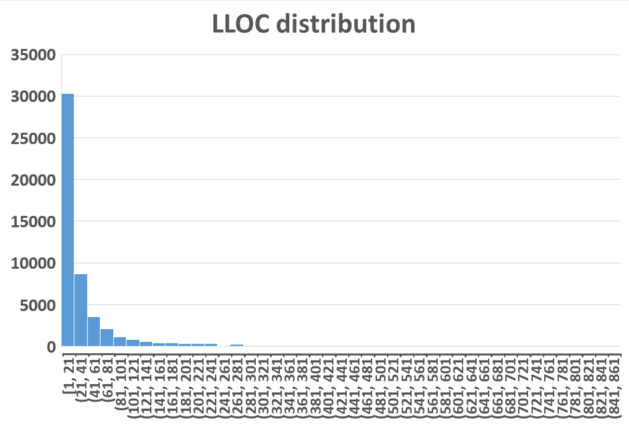

Figure 1: Distribution of the logical code lines metric values across contracts

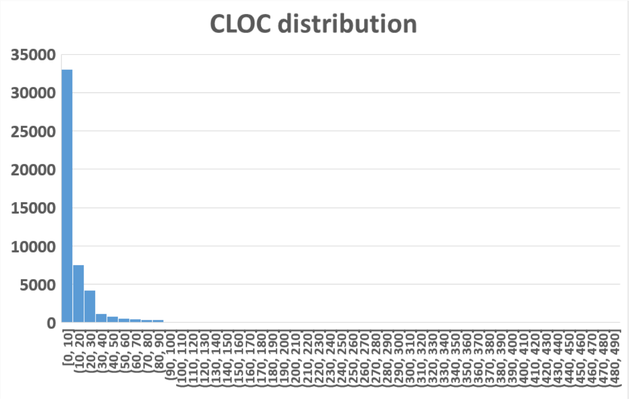

Figure 2: Distribution of the comment lines metric values across contracts

3.1 Metrics Distributions

We plotted also the distributions of the metric values using histograms. Figure 1 shows the logical code lines metric distribution. Although the maximum value is 843, the 25%, 50% and 75% quartiles are only 10, 18, and 37, respectively. The maximum and quartile values for the comment lines metrics (Figure 2) are 481, 0, 2, and 13, respectively. Regarding the number of functions (Figure 3), the maximum is 93; the quartiles are only 2, 3, and 6.

As for the complexity measures, Figure 4 shows the weighted complexity of the functions and Figure 5 displays the nesting level histogram. The maximum of WMC is 522, but quartiles are 2, 4, and 7, respectively. The sum of the deepest nesting levels in a contract is 125, but quartiles are 0, 0, and 1. NL maximum is a very outlying value, thus we manually checked what caused this extreme. Our current implementation counts each *else-if* statement as an additional depth and the contract in question[12] contains a function with a huge *if-else-if* structure. In the forthcoming versions of SolMet we will add the NLE metric, which is similar to NL, but without counting the *else-if* statements.

[12] Contract named WinMatrix with the address 0xDA16251B2977F86cB8d4C3318e9c6F92D7fC1A8f

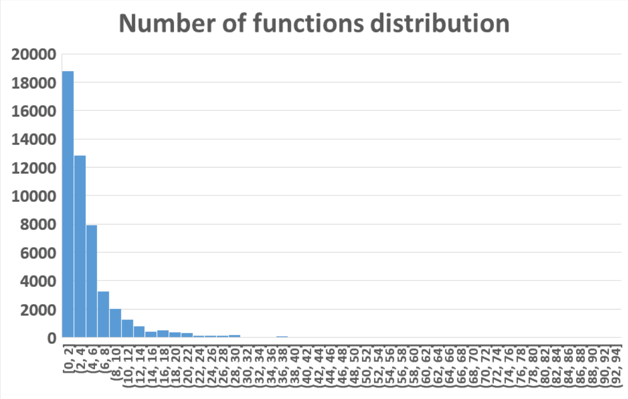

Figure 3: Distribution of the number of functions metric values across contracts

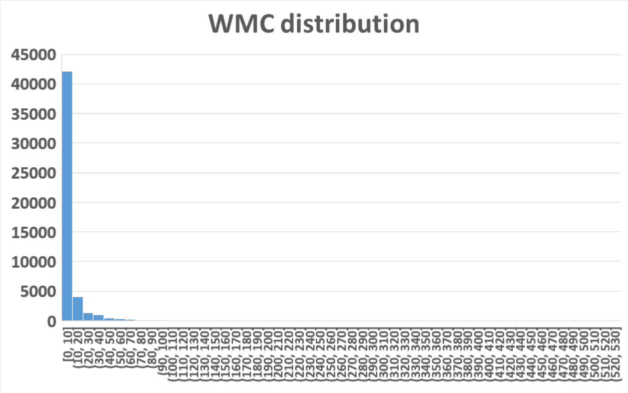

Figure 4: Distribution of the WMC metric values across contracts

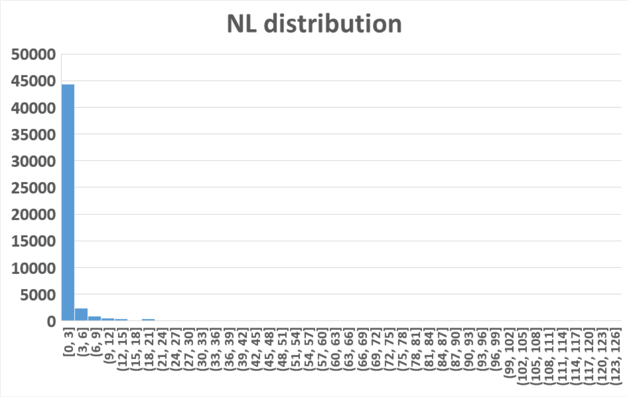

Figure 5: Distribution of the NL metric values across contracts

All the detailed numbers and charts of the presented results are available online as well.[13]

3.2 Discussion

What we can observe is that except several extreme high values for each metrics, most of the smart contracts share some common properties. Most of them are short, defines only a few functions with sequential control structures. There is no common trend in using comments; the contracts are either well-commented or not commented at all. The fact that above 80% of the defined libraries implements similar safe mathematical operations cries for a better solution at the level of Solidity language. Some kind of common dependencies management system would be desirable.

4 CONCLUSION

In this paper, we proposed the usage of well-known static OO metrics to the smart contracts written in the Solidity contract-oriented language. To the best of our knowledge, there are no tools for calculating such metrics. Given the fact that these metrics developed together with the programming languages themselves and many papers showed their efficient applications in QA activities, we believe the new era of blockchain programming could benefit from them as well.

We implemented several size and complexity metrics in our SolMet tool and calculated them for more than 45,000 contracts from 10,206 Solidity smart contract source code files. We were able to get a quick overview about the typical structure of smart contracts in terms of their sizes and complexity. However, our tool is very immature in its current state, thus all the presented results should be handled with appropriate care.

We continuously add new metrics to SolMet and expect that inheritance and coupling style metrics will provide even more thoughtful insights of smart contract structures. Once a proper metric suite is defined and implemented, we can start utilizing them in various forms that worked very well for other languages, for example to build bug or vulnerability prediction models, to aid code review and guide refactoring activities or detect common anti-patterns in the code before being deployed and becoming permanent.

ACKNOWLEDGMENTS

This research was supported by the UNKP-17-4 New National Excellence Program of the Ministry of Human Capacities, Hungary.

REFERENCES

[1] Vitalik Buterin et al. 2013. Ethereum white paper. *GitHub repository* (2013).
[2] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).
[3] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.
[4] Phil Daian. 2016. Analysis of the DAO exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/. (2016).
[5] Chris Dannen. 2017. Solidity Programming. In *Introducing Ethereum and Solidity*. Springer, 69–88.

[13] https://github.com/chicxurug/wetseb-2018-data/tree/master/study-results

[6] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269.

[7] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[8] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[9] N Szabo. 1997. Nick Szabo – the idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials. http://szabo. best. vwh. net/smart_contracts_idea. html* (1997).