The Journal of Systems & Software 186 (2022) 111193

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Profiling gas consumption in solidity smart contracts☆

Andrea Di Sorbo, Sonia Laudanna, Anna Vacca *, Corrado A. Visaggio, Gerardo Canfora

Department of Engineering, University of Sannio, Italy

ARTICLE INFO

Article history:
Received 2 April 2021
Received in revised form 9 December 2021
Accepted 12 December 2021
Available online 17 December 2021

Keywords:
Software engineering for blockchain technologies
Smart contracts optimization
Code quality
Software metrics
Empirical study

ABSTRACT

Nowadays, more and more applications are developed for running on a distributed ledger technology, namely dApps. The business logic of dApps is usually implemented within smart contracts developed through Solidity, a programming language for writing smart contracts on different blockchain platforms, including the popular Ethereum. In Ethereum, the smart contracts run on the machines of miners and the gas corresponds to the execution fee compensating such computing resources. However, the deployment and execution costs of a smart contract depend on the implementation choices done by developers. Unappropriated design choices could lead to higher gas consumption than necessary. In this paper, we (i) identify a set of 19 Solidity code smells affecting the deployment and transaction costs of a smart contract, and (ii) assess the relevance of such smells through a survey involving 34 participants. On top of these smells, we propose GasMet, a suite of metrics for statically evaluating the code quality of a smart contract from the gas consumption perspective. An experiment involving 2186 smart contracts demonstrates that the proposed metrics have direct associations with deployment costs. The metrics in our suite can be used for more easily identifying source code segments that need optimizations.

1. Introduction

Blockchain emerged as the enabling technology (Nakamoto, 2008) for implementing transactions of electronic cash, namely *cryptocurrency*, without the brokerage of a financial institution. Thanks to the versatility of this technology, it is now increasingly adopted in several contexts, with different purposes far beyond crypto-currencies. Indeed, blockchain is impacting a variety of business sectors; at illustrative aim, some recent applications using blockchain concern[1]: sharing of sensitive data, electronic voting, cross-border payments, personal identity security, goods authenticity and traceability, real estate processing. The global blockchain technology market size is expected to reach USD 57,641.3 million by 2025 (Anonymous, 2019). This forecast entails that in the next years we will witness a significant spreading of decentralized applications (dApps), *i.e.,* computer applications which run on a distributed ledger technology (DLT).

The business logic of dApps is usually implemented within smart contracts, *i.e.,* fully-fledged programs that run on blockchain. Ethereum is one of the most popular blockchain platforms (Chen et al., 2017) and provides an environment to code and run smart contracts (Wohrer and Zdun, 2018). In this environment, smart contracts are typically developed through the Solidity object-oriented language, before being compiled into bytecodes that can be executed by the Ethereum Virtual Machine (EVM).

As each underlying technology imposes its peculiarities and characteristics to applications that run on top of it, also blockchain introduces critical aspects affecting smart contract development (Vacca et al., 2021). For instance, the immutability of blockchain complicates smart contract maintenance activities, as, once deployed, they can hardly be patched (Destefanis et al., 2018). For this reason, it is crucial ensuring that a smart contract is bug-free and well designed *before* deploying it to the blockchain (Chen et al., 2020b). Besides, since smart contracts run on a blockchain infrastructure, a key factor is the cost of execution due to the mining of the blocks participating in the chain. On the Ethereum platform, the gas (in Ether) corresponds to the execution fee compensating for such computing resources. Pragmatically speaking, gas corresponds to real money; unoptimized smart contracts can lead to unnecessary gas leaks and, thus, to money losses (Marchesi et al., 2020). Moreover, when the users produce a high number of transactions, if the dApp is not properly designed, the execution costs could be not sustainable by the business model of the Decentralized Application Organization (DAO) which manages it.

The choices done by developers, such as the types of data structures used, the number of cycles, the kind of instructions,

☆ Editor: Burak Turhan.

* Corresponding author.
E-mail addresses: disorbo@unisannio.it (A. Di Sorbo), slaudanna@unisannio.it (S. Laudanna), avacca@unisannio.it (A. Vacca), visaggio@unisannio.it (C.A. Visaggio), canfora@unisannio.it (G. Canfora).

[1] See https://builtin.com/blockchain/blockchain-applications

https://doi.org/10.1016/j.jss.2021.111193
0164-1212/

the types of variables used, where and how they are initialized or valued, may affect the gas consumption of a smart contract. Although recent research (Marchesi et al., 2020) outlined design patterns and guidelines for the development of more optimized smart contracts from the gas consumption perspective, programming resources and development environments that can help developers more easily identifying code units that need to be optimized are still lacking. This is also confirmed by the survey conducted by Zou et al. (2019), in which the majority of interviewed smart contract developers feel that optimizing gas is painful, especially in complex applications, and it would be important to have tools that allow optimizing smart contract source code rather than bytecode. Indeed, to estimate gas consumption, a typical workflow consists of deploying the smart contract within a simulated DLT running on a private workstation or local servers and obtaining an estimation of the cost.[2] If the cost is too high, the smart contract is modified, deployed again on the local simulated DLT, and the new cost is evaluated. This process is repeated till the result is satisfying. Recently, Ajienka et al. (2020) studied the correlation between object-oriented metrics and the resources required for smart contracts deployment. The results of this prior work demonstrate that while inheritance-based metrics represent good indicators of the gas used for smart contracts deployment, coupling-related metrics do not. This achievement partially confirms the belief that traditional software metrics do not capture all the specific aspects of smart contracts and the need of metrics specifically designed for smart contracts (Ducasse et al., 2019).

To fill this gap, the goal of our work is to provide a tool for helping developers more easily identifying code units that may be optimized for achieving gas savings. In particular, developer communities have identified Solidity *cost smells*, which are coding practices that can negatively affect deployment and transaction costs of a smart contract. Such cost smells are not gathered into a unique catalog, but they are dispersed within different books, reports, and web forums. In this paper, we collect a set of 19 cost smells and, through a survey involving 34 among smart contract developers and researchers, assess the perceived relevance of the collected cost smells. The surveyed developers discussed the extent to which they agree that the identified smells are actual problems and that the proposed solutions are potential candidates to fix those problems, achieving an average agreement of about 70% over the smells and solutions presented in the catalog. On top of these smells, we define a suite of code metrics, namely GasMet (standing for *Gas Metrics*), for statically evaluating the code quality of smart contracts, from the gas consumption perspective. Through an experiment involving 2186 real-world smart contracts, we obtain empirical evidence about the relationships existing between the defined metrics and gas consumption required by the deployment of a smart contract. The advantage brought by GasMet is that the developers can more easily localize the segments of their code that need optimization, *while* coding, without deploying or running it on a DLT, simulated or real. Thus, the metrics in our suite can help developers implementing more optimized smart contracts requiring lower resource consumption. In particular, the collection of GasMet metrics can give developers general indications of gas consumption inefficiencies occurring in the code, fostering the application of gas-saving patterns (Marchesi et al., 2020). Besides, blockchain researchers could leverage the proposed metrics to develop linters for accurately capturing the occurrences of cost smells in source code.

The original contribution of this paper includes:

- *a collection of cost smells, i.e., coding practices that entail a higher cost of smart contract deployment and execution;*
- *the results of a survey involving real smart contract developers on the perceived relevance of each defined cost smell;*
- *GasMet, a suite of metrics for identifying the occurrences of the defined cost smells; and*
- *an empirical study for identifying the GasMet metrics that most correlate with deployment costs.*

Previous research proposed tools based on symbolic execution for (i) automatically inferring gas upper bounds of smart contracts' public functions to prevent out-of-gas vulnerabilities (Albert et al., 2019), or (ii) analyzing the number and types of bytecode instructions executed to detect under-optimized storage patterns (Albert et al., 2020), whereas static analysis techniques turned out to be effective for automatically identifying gas-related vulnerabilities (Grech et al., 2018). Our suite of metrics (i) provides information that is complementary to the one provided by the aforementioned tools, and (ii) aims at supporting developers in identifying potential inefficiencies in the smart contracts/functions implementations that could lead to higher gas consumption when deploying smart contracts. In particular, our metrics are related to a variety of gas-inefficient practices (not only to storage usage) and they are computed directly on the Solidity source code. In addition, our metrics do not take into account function inputs, as previous experiments demonstrated that only a small percentage of smart contracts (*i.e.,* about 10%) do not follow the Ethereum safety recommendations[3] and their gas consumption depends on the size of data stored, the size of functions inputs, or the blockchain state (Albert et al., 2019).

Paper structure. The paper proceeds as follows. Section 2 highlights the novelty of our findings with respect to the existing literature. Section 3 details the identified *cost smells*, while Section 4 illustrates our study on the relevance of the identified cost smells and discusses the related results. Section 5 introduces the GasMet suite and tool, while Section 6 deals with the evaluation of the suite and the results of our correlation analysis with gas consumption. Threats to our study's validity are commented in Section 7 and, finally, Section 8 concludes the paper.

2. Related work

In this section, we discuss the related literature that has been mainly devoted to studying (i) software metrics for Blockchain-Oriented Software (BOS in the remainder of the paper), (ii) issues related to weaknesses in smart contracts source code, and (iii) evaluation of gas consumption.

2.1. BOS software metrics

Ortu et al. (2019) provided a statistical characterization of BOS. The authors inspected and compared 5 C++ open source Blockchain-Oriented and 5 Traditional Java software systems, to discover strength differences between the two categories of projects, and particularly in the statistical distribution of 10 software metrics. Even if there are similarities between the statistical distributions for Traditional software and Blockchain software, the distribution of Average Cyclomatic and Ration Comment To Code metrics detect significant differences, whereas the Number of Statements metric reveals meaningful differences on the double Pareto distribution.

Hegedűs (2019) used Object-Oriented (OO) metrics for estimating properties of the smart contracts written in Solidity. Based on the results, the author found that smart contract programs are short, not excessively complex and with few or no comments.

[2] https://ethereum.stackexchange.com/questions/27452/how-to-estimate-gas-cost

[3] https://github.com/ethereum/wiki/wiki/Safety

Furthermore, it would be useful for smart contracts to have an external library and dependency management mechanisms because many libraries have similar functionalities.

Tonelli et al. (2018) investigated the differences between the software metrics measured on Smart contracts (SC) and metrics extracted from traditional software systems. The authors built their dataset from the Etherscan collection,[4] taking the Smart Contracts bytecode, the Application Binary Interface (ABI), and the Smart Contract source code (written in Solidity) for each sample. The authors implemented a code parser to extract the software metrics for each smart contract considered. They have computed: the total lines of code to a specific blockchain address, the number of smart contracts inside a single address code, blank lines, the comment lines, the number of static calls to events, the number of modifiers, the number of functions, the number of payable functions, the cyclomatic complexity as the simplest McCabe definition, the number of mappings to addresses for the dataset considered. Based on the results, smart contract lines of code metric is the metric that is closest to the statistical distribution of the corresponding metric in a traditional software system.

Gencer et al. (2018) proposed a measurement framework for two of the dominant cryptocurrencies, Bitcoin and Ethereum. They estimated the network resources of nodes and the interconnection among them, the protocol requirements influencing the operation of nodes, and the robustness of the two systems against attacks to analyze the depth of decentralization. They found that neither Bitcoin nor Ethereum has rigorously better properties than the other.

Other papers have computed more specific metrics tailored for specific application domains, such as healthcare (Zhang et al., 2017), and model the real-time predictive delivery performance in supply chain management (Meng and Qian, 2018).

With respect to these metrics, the ones of the suite proposed in this paper aim at evaluating Solidity code's patterns that deteriorate the performance of the smart contract's deployment.

2.2. Weaknesses on smart contract's source code

A part of the literature on BOS concerns the analysis and detection of different types of weaknesses affecting the code of a smart contract.

Ye et al. (2019) realized a comparison of the state-of-art bug detection tools and executed experiments to find their advantages and disadvantages.

By analyzing the literature, Demir et al. (2019) identified the vulnerabilities that developers must fix when writing smart contracts. In addition, they analyzed applications that seek these vulnerabilities and provided an overview of how they are used and which they cover. In their analysis, they identified issues related to smart contracts and provided a discussion about the problems, the challenges and the techniques of the available technology in this area.

Peng and Rajan (2019) proposed SIF, a framework useful for Solidity contract monitoring, instrumenting, and code generation. SIF is able to detect bugs, analyze, optimize and generate code to support developers and testers. This framework has been tested on real smart contracts deployed on the Ethereum platform.

Tikhomirov et al. (2018) propose a classification of problems that may occur in smart contract code. Subsequently, they implemented a static analysis tool that can identify them, called Smartcheck. The tool converts Solidity source code into an XML-based intermediate representation and compares it with XPath models. The tool has been tested on a large set of real smart contracts.

Dhawan (2017) presented ZEUS, a framework to check the correctness and confirm the fairness of smart contracts. By correctness, they mean using secure programming practices, instead, fairness indicates compliance with high-level business logic. ZEUS simultaneously uses abstract interpretation, the symbolic model checking and horn clauses to speedily check the security of smart contracts. Zeus has been tested on over 22,000 smart contracts and it has shown that around 94.6% of them are vulnerable.

In our study, we focus on the *cost smells* which, at the best knowledge of the authors, is a novel area of investigation in the literature regarding BOS.

2.3. Gas cost evaluation

Literature studying the relationship between the smart contract's code and the cost of its deployment and execution has yet a small number of contributions.

Baird et al. (2019) explore the economics of smart contracts. There is a disparity that continues to increase between the actual costs of executing smart contracts and the computational costs. This occurs because the gas cost model of the Ethereum Virtual Machine (EVM) instruction-set is poorly implemented. To resolve this problem momentarily, the Ethereum community increases the cost of gas periodically. In their study, the authors showed a new gas cost model that fixes the principal irregularity of the current Ethereum gas cost model. Their new cost model blocks the ongoing inflation of execution time per unit of gas.

Chen et al. (2017) proposed analysis to show that many smart contracts have dependencies on the cost of gas and could be replaced by more efficient bytecodes to save gas. For this goal, they implement GASPER to automatically discover gas-costly programming patterns from the bytecode of smart contracts. Their analysis focuses on dead codes, opaque predicates, expensive operations in a loop in relation to the gas cost using bytecode; our metrics extract information from a smart contract but at a higher level than bytecode, i.e. at the source code level.

More recently, Albert et al. (2019, 2020) proposed methods and tools for automatically inferring gas upper bounds for smart contract functions, while Grech et al. (2018) presented a static analysis tool for detecting gas-focused vulnerabilities in smart contracts. Instead, Marchesi et al. (2020) have identified a set of 24 design patterns that influence gas consumption in the execution of Ethereum smart contract. After classifying these design patterns into 5 categories (external transaction, storage, saving space, operations and miscellaneous), for each pattern, they describe the problem and a possible solution. We can observe similarities between the cost smells and patterns identified by Marchesi et al. (2020). In particular, based on the descriptions provided, 14 cost smells identified in our study are similar to patterns presented in prior work. In contrast, our study reports five cost smells, two related to the number of loops present in the contract and the number of variables declared in them (*i.e.,* CS11 and CS1, respectively), two concerning the use of memory type arrays and public members (*i.e.,* CS7 and CS2, respectively), and one pertaining to the number of indexed parameters within events (*i.e.,* CS13), that were not discussed by Marchesi et al. (2020). In addition, differently from this previous study, we (i) assessed the relevance of the identified cost smells through a survey involving real smart contract developers, and (ii) defined a suite of metrics for more easily identifying the code smells that can negatively impact gas consumption. On static profiling and optimization of Ethereum smart contracts, Correas et al. (2021) presented a novel static profiling technique based on static resource analysis to generate upper-bound expressions on a variety

[4] https://etherscan.io/

of resources (*e.g.,* the number of storage instructions, gas consumed by some EVM operations, total ether sent by an external call, etc.) that can be used for optimizing the gas consumption of smart contracts. Moreover, through the use of an automatic optimization of Solidity programs, they propose to reduce gas consumption by replacing the accesses to state variables by gas-efficient accesses to local variables. Brandstätter et al. (2020) presented a python solidity-optimizer based on classical code efficiency optimization strategies in the context of smart contracts, postulated in early work by Bentley (1982) and grouped into six categories (time-for-space rules, space-for-time rules, loop rules, logic rules, procedure rules, and expression rules). Chen et al. (2020a) proposed ten gas-inefficient programming patterns belonging to four categories. The first category regards useless code (i.e., opaque predicates and dead code). The second category encompasses the expensive operations in loops (such as the cost smell CS1 identified in our study), the fusible loops, and the repeated computations in loops. The third category deals with the wasted disk space (*e.g.,* storage that is never used after definition). Finally, the fourth category comprises gas-inefficient operation sequences (*i.e.,* consecutive EVM operation sequences that can be replaced with gas-efficient operation sequences). The authors also presented GasChecker, a tool able to detect the defined gas-inefficient patterns in the bytecode of Ethereum smart contracts. Unlike the GasMet tool that collects metrics at the source code level, GasChecker is based on symbolic execution and works at the bytecode level.

In a different effort, Chen et al. (2020b) defined 20 types of *contract defects*, by analyzing posts on Ethereum StackExchange.[5] Among the defects identified, there are functions and data types that can increase gas consumption. As in our study, to validate the elicited *contract defects*, they conduct a survey with practitioners and evaluate the diffusion of such defects in 587 real world smart contracts. However, while previous work mainly focuses on characterizing different types of defects and only suggests the implementation of practical tools for practitioners, our study has a special focus on bad coding practices that can negatively affect gas consumption, also providing developers with a concrete suite of metrics for more easily evaluating smart contract code quality from the gas consumption perspective.

3. Elicitation of cost smells

We identified a set of *cost smells*, by inspecting specialized forums and books dealing with the development of Solidity smart contracts. Although exhaustively identifying any possible cost-related defect in smart contracts is not an objective of this paper, we relied on multiple sources to spot out some common bad practices in smart contract development with Solidity that can negatively impact gas consumption. In particular, we (i) leveraged some textbooks (Badr et al., 2018), highlighting good practices in smart contracts and decentralized applications development, and (ii) consulted technical blog posts dealing with Solidity and gas optimization tips. Specifically, to mine potential smell-related posts, the keywords (i) "Solidity gas consumption", (ii) "Solidity gas usage", (iii) "Solidity gas optimization", and (iv) "Solidity gas saving" have been used to perform the Google search. By inspecting the ten top-ranked retrieved documents for each search query, we were able to discover ten blog posts specifically targeting Solidity coding practices for optimizing gas usage (Chen, 2018b,a; Gupta, 2018, 2019; Szego, 2018; Bhushan, 2018; Jelski, 2019; Cipher, 2018; Škvorc, 2018; Blitz, 2018). Relying on the aforementioned information sources, we found that, in general, gas consumption could be negatively affected by:

- *inefficient use of data storage;*
- *inefficient implementation of functions.*

In particular, we identified 19 *cost smells* (*i.e.,* coding practices entailing higher gas consumption). Table 1 reports the identified *cost smells*, along with a brief description of each smell and a reference to the webpage from which it has been deduced. In particular, the cost smells reported in Table 1 could be related to either inefficient use of data storage (*i.e., CS1, CS4, CS7, CS8, CS9, CS10, CS11, CS12, CS13, CS14, CS15, CS17,* and *CS18*) or inefficient implementation of functions (*i.e., CS2, CS3, CS5, CS6, CS16,* and *CS19*).

4. Study on the relevance of cost smells

The *goal* of this study is to investigate whether the identified cost smells (see Section 3) are considered as relevant by smart contract developers. To pursue this goal, we pose our first research question:

RQ_1: To what extent are the identified cost smells relevant for smart contract developers?

4.1. Research method

To address RQ_1, similar to previous work on code smells assessment (Borrelli et al., 2020; Tufano et al., 2016; Aniche et al., 2018), we conduct a survey targeting experts in the development of Solidity smart contracts and blockchain applications. Prior research (Mäntylä et al., 2004; Palomba et al., 2014) found that developers may have divergences of opinions about the incidence and the perceived relevance of code smells. Thus, to assess the extent to which the identified cost smells (reported in Table 1) are issues that might actually affect gas consumption, we decided to survey real smart contract developers and blockchain experts and ask them their opinions about the relevance of all the aforementioned smells. In particular, the survey with developers represents an important part of our study, as its results may help to avoid considering eventual cost smells for which developers report a low perceived relevance. Relevant guidelines on how to design and carry out survey studies in software engineering are provided in previous work (Pfleeger and Kitchenham, 2001; Kitchenham and Pfleeger, 2002d,c,b,a, 2003). On the one hand, according to such guidelines, we (i) *design specific and measurable goals*, (ii) *target subjects able to answer the questions posed*, (iii) *group questions into topics*, and (iv) when possible, *use standardized response formats*. On the other hand, we cannot estimate whether the set of subjects involved in our survey is a representative subset of the target population, as we have no prior knowledge of such a population. Specifically, the survey is structured in three sections:

- The first section aims at collecting demographic information about respondents. In particular, this section comprises questions about: (i) the highest education qualification; (ii) the domain in which respondents work (*e.g.,* industry, academia, etc.); (iii) their role in the organization (*e.g.,* developer, project manager); (iv) the years of experience in software development; (v) more specifically, the years of experience in smart contract development, and (vi) the languages used to develop smart contracts.
- The second section gathers the developers' perceived relevance about the identified Solidity code smells that may affect gas consumption. Each smell is presented to respondents through a brief description of the problem, outlining its effects on gas usage and the optimizations to mitigate or avoid such effects. For each smell, we ask to provide a relevance score on a 5-level Likert scale (Oppenheim, 2000) (in which 1 corresponds to *strongly disagree* and 5 corresponds to *strongly agree*). Finally, for each question, the respondents could provide an optional open comment.

[5] https://ethereum.stackexchange.com/

Table 1
The list of the identified Cost Smells along with the rationale explaining the relationship with the gas consumption (ordered by type).

Id	Type	Cost Smell	Description	Ref.
CS1	storage	Duplicate writes	Modifying a variable's value several times could require much gas. To save gas, developers should overwrite variables outside cycles as much as possible.	Chen (2018b)
CS4	storage	Inefficient initialization of variables	An uninitialized variable is automatically set with its default value (*e.g.,* a `uint256` variable, when not initialized, it is assumed to have the default value 0). When declaring a variable, explicitly setting it with its default value is useless and wastes gas.	Gupta (2019)
CS7	storage	Inefficient use of memory arrays	Whenever a developer has to make some internal computation in a Solidity function with the help of an array, it may be good to avoid using storage, by employing `memory` arrays. If the size of the array is exactly known, fixed size `memory` arrays can be used to save gas.	Szego (2018)
CS8	storage	Inefficient use of strings	Using `bytes32` is cheaper than using the `string` type. If the length of the string can be limited to a certain number of bytes, bytes1 to bytes32 data types are preferable wherever possible.	Bhushan (2018)
CS9	storage	Inefficient use of return values	A simple optimization in Solidity consists of naming the return value of a function. It is not needed to create a local variable then.	Jelski (2019)
CS10	storage	Inefficient use of global variables	Storing global variables in memory is expensive in terms of gas. Number and size of global variables should be minimized.	Cipher (2018)
CS11	storage	Unbounded loops	In general, loops should be avoided. If avoiding loops is not possible, it could be beneficial to try to avoid unbounded loops, *i.e.,* loops where the upper limit of iterations is not defined.	Škvorc (2018)
CS12	storage	Inefficient use of data types	Use `bytes32` whenever possible, because it is the most optimized storage type. For example, storing a small number in a `uint8` variable is not cheaper than storing it into a `uint256` variable, as, for storing, any smaller data is padded with zeros to fill the 32 bytes, requiring additional operations from the EVM and additional gas.	Škvorc (2018)
CS13	storage	Inefficient use of indexed parameters	The indexed parameters in events have additional gas costs. It is preferable to only use the indexed qualifier for event parameters that should be searchable.	Badr et al. (2018)
CS14	storage	Inefficient use of structs	Since many DApps use storage, it would be useful to reduce archiving costs in order to optimize gas costs. In particular, instance or struct variables can be packed together to reduce storage costs, while mappings cannot. Thus, using a high number of mappings could result in higher storage costs than using variables that can be packed into single storage slots.	Blitz (2018)
CS15	storage	Inefficient use of mappings	As arrays are not stored sequentially in memory and each access to array elements requires a key–value lookup, in Solidity, arrays are more expensive versions of mappings with added features making them array-like (*e.g.,* length, bound checking, sophisticated packing behaviors, automatic zeroing out of unused storage slots, special optimizations). To save gas, mappings are preferable to arrays.	Gupta (2018)
CS17	storage	Inefficient use of booleans	Booleans (bool) are uint8 which means they use 8 bits of storage even if they can have only two values: True or False. When EVM packs the bools normally it can store only 32 bools in one memory slot. Otherwise, a set of 256 different booleans could be more efficiently packed in a single word by not declaring them as bool but uint256, using one bit for each boolean value.	Gupta (2019)
CS18	storage	Inefficient use of events	It is cheaper to store data that is not required on-chain in events rather than variables.	Gupta (2019)
CS2	function	Abundance of public members	The order of the functions influences the gas consumption. Since the order of the functions is based on the method ID, this implies that the subsequent ordering can consume additional gas. Depending on the VM transaction, each position will have an additional gas fee. Since all `public` members participate in the sorting, reducing public members could save gas.	Chen (2018a)

(*continued on next page*)

- The final section asks questions about the general perceived usefulness of a suite of metrics for more easily identifying the identified cost smells while coding, *i.e.,* (i) whether respondents perceive the availability of such a suite of metrics useful, and (ii) whether they would be willing to adopt it during smart contract development. Finally, we ask to

Table 1 (*continued*).

Id	Type	Cost Smell	Description	Ref.
CS3	function	Scarcity of external functions	Storing the input parameters in memory costs gas. For all public functions, the input parameters are copied to memory automatically. If a function is only called externally, it should be explicitly marked as `external`, in a way that these parameters are not stored into memory but are read from call data directly. This can save gas when the function input parameters are huge.	Gupta (2018)
CS5	function	Inefficient use of libraries	The bytecode of library functions is not made part of a deployed client smart contract. Thus, smart contract developers could use software libraries to implement complex logic. Library imports help to keep the size of the client smart contract small, consequently reducing the gas required for deploying it.	Gupta (2019)
CS6	function	Inefficient use of internal functions	From inside a smart contract, calls to internal functions are cheaper than calls to public functions. A call to a `public` function implies that all the parameters are copied into memory and passed to that function. Conversely, a call to an internal function does not entail copying such parameters into memory again. For this reason, the use of `internal` functions is preferable whenever possible, especially when the parameters are big.	Gupta (2019)
CS16	function	Inefficient use of external calls	Every call to an external contract costs a decent amount of gas. For optimizing gas usage, it is better to call one function and have it return all the needed data rather than calling a separate function for every piece of data.	Gupta (2018)
CS19	function	Inefficient use of functions	In Solidity, it could be preferable to use fewer larger functions, rather than implementing multiple functions, each performing a single small task. Indeed, multiple smaller functions cost more gas and require more bytecode.	Gupta (2019)

provide free comments about possible Solidity code smells that were not considered in our study.

To recruit participants, we posted a link to the questionnaire on Reddit channels related to Ethereum and Solidity development, namely `solidity`, `dapps`, `cryptodevs`, `ethdev`, and `ethereum`. Along with the link to the online survey, we added a short message explaining its purpose, the estimated duration (20 min), and our will to only use the collected data in aggregated, anonymized form. Besides, we sent the invitation for the questionnaire completion to our contacts who are working or studying blockchain-related topics in companies or academic institutions. Furthermore, we invited practitioners who regularly contribute to smart contract-related projects on GitHub.

We summarize the responses obtained in our survey in the form of diverging stacked bar charts, also discussing the open comments reported by the respondents.

4.2. Results

Our survey on cost smells relevance has been kept open for one month. During this period we were able to collect responses from 34 different respondents. Considering that (i) the community of smart contract developers is not so big (*e.g.,* blockchain technology is still in its infancy (Grover et al., 2019)), (ii) our survey asked to assess a quite high number of different cost smells (*i.e.,* it required a moderate amount of time for completion), and (iii) the exploratory nature of our survey, we believe that 34 responses are adequate for our purposes. Out of the 34 respondents, 12 own a Bachelor's degree, 12 a Master's degree, and 6 a Ph.D. 14 respondents have less than 5 years of development experience, 10 between 5 and 10 years, and 10 more than 10 years. Concerning smart contract development, 12 respondents have less than a year of experience, 16 between 1 and 3 years, and 6 more than 3 years. Considering that Ethereum and Solidity were first released in late 2015 and that about 65% of the surveyed practitioners declare more than one year of experience in smart contract development, we believe that the respondents have adequate expertise for judging the relevance of identified cost smells (Chen et al., 2020b). Most of the participants (33) in our survey declare to develop smart contracts using Solidity, while just one respondent develops smart contracts with Kotlin. About 80% of survey respondents (27) are smart contract developers, besides 3 blockchain researchers, 2 teachers at blockchain-related courses, one participant involved in smart contract testing, and one (is a) product owner. It is worth pointing out that while the majority of participants in our survey were already aware of most of the 19 smells, by looking at the survey responses, we observed that, for some smells (*i.e., CS2, CS9, CS12, CS13,* and *CS18*), one or two participants declared to be unfamiliar with the specific bad practices.

In Fig. 1 the diverging stacked bar charts summarize the perceived relevance of the 19 identified cost smells. The three percentages reported in the graph indicate the proportion of disagreements, neutral responses, and agreements, respectively. As illustrated in Fig. 1, the majority of interviewed developers agree or strongly agree on most of the identified *cost smells*. In particular, about four-fifths of the participants in our study agree or strongly agree on nine smells (*i.e., CS3, CS6, CS7, CS8, CS10, CS11, CS13, CS14,* and *CS16*), about three-fifths of them agree or strongly agree on five smells (*i.e., CS1, CS9, CS12, CS15,* and *CS18*), and for only two smells (*i.e., CS2* and *CS17*) we observe that less than 50% of respondents agreed. However, in these latter two cases (*i.e., CS2* and *CS17*), developers mainly tend to assume a neutral position (35% and 44% of respondents, respectively), rather than disagreeing. Indeed, concerning *CS2*, surveyed developers acknowledge that public members influence gas consumption, *"though this is not the location where devs should worry too much about optimization"*. According to their opinions (i) the cost/benefit of reducing public members could be not so high (*e.g., "The effects on gas costs due to function selector I would imagine are secondary to other savings I'd imagine"*), and (ii) the optimization should be performed at Ethereum or compiler level (*e.g., "This should be fixed at the Ethereum or Solidity compiler level".*). Similarly, for *CS17*, although *"bit-fiddling-concepts are an amazing source for gas-saving tactics"*,

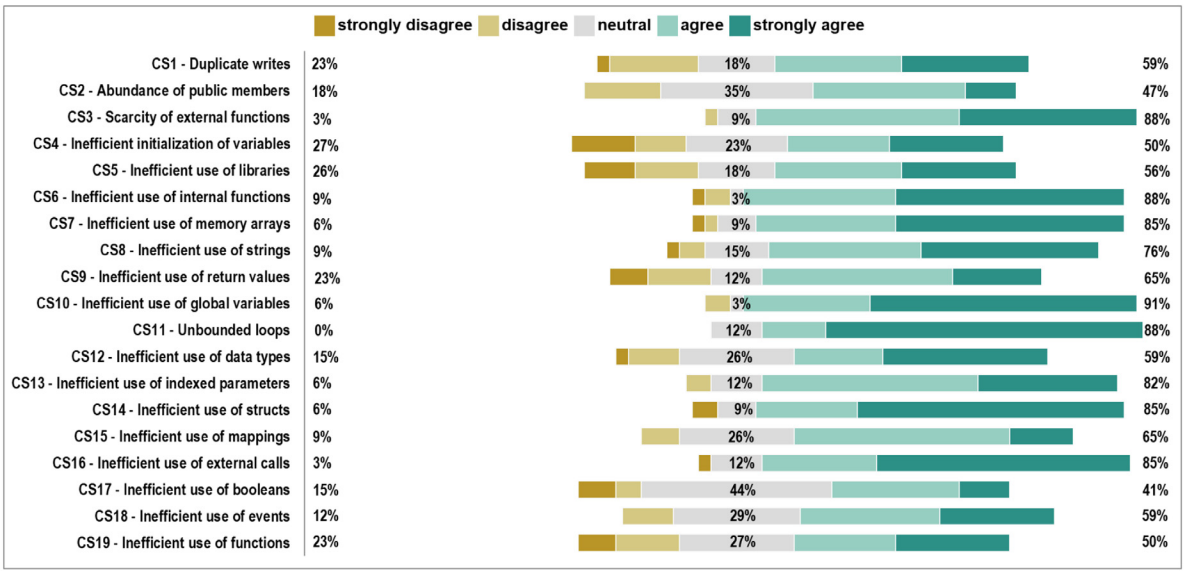


Fig. 1. Perceived relevance of the 19 cost smells.

survey respondents feel that these tactics significantly hamper code readability (*e.g., "I don't think it is worth saving gas if it impacts that much the code readability"*) and *"the benefit in terms of gas consumption should be carefully evaluated"*. However, for ensuring both code readability and gas savings, they recommend that this kind of optimization should be handled at compiler level rather than source code level (*e.g., "If this is of concern, the compiler should handle it"*).

For all identified smells, less than one-third of the participants disagree or strongly disagree. In the cases in which we observe higher percentages of disagreements, survey participants are mostly concerned about code readability and believe that eventual optimization should be performed at the compiler level, rather than directly in the source code. Code readability is an important aspect to consider as developers spend a lot of time reading and inspecting code. Thus, code readability is particularly important during smart contracts implementation and maintenance activities (Canfora et al., 2021). Previous research (Chen et al., 2021) also demonstrated that smart contract developers often reuse code blocks implemented in other smart contracts. Thus, lower readability would make the smart contract harder to understand, consequently hampering code reuse (Chen et al., 2020b). As reported in previous work (Zou et al., 2019), it is often hard to optimize gas without reducing code readability, as more efficient code basically requires fewer instructions. This is the case of *CS4*, in which, even if the developers are aware that avoiding initializing a variable to its default value would be an easy strategy for gas optimization, they also argue about the negative effects on code readability (*e.g., "It saves gas but also degrades readability"*). Indeed, according to some of the participants in our survey, explicitly initializing variables is always a good practice, otherwise *"it is difficult during a review to know if the intent for the variable was to be initialized to the default value, or if the variable is missing the initialization"*. Other participants suggest *"adding a small comment to document the default value"* initialization, to save both gas and code readability. For the same reason, some of the developers involved in our study believe that default value variable initialization could be optimized at a level lower than source code (*e.g., "Statically initializing a variable to its default value should be caught by the compiler and reduced to a no-op"*).

Comments of a similar sort are received for *CS9*, where respondents recommend balancing *"the readability and the gas consumption"*, as the *"benefit is minimal compared to readability"*. Indeed, they believe that the few *"cost saved does not justify the decrease in code readability"*, as named return values *"are frequently the source of vulnerabilities"*. Thus, since named return values help in *"reducing the size of a contract, although of a small factor"*, their usage is *"an optimization the compiler should do"*. This depends also on the programming style chosen (*e.g., "…it is a matter of taste whether an explicit return is better readable than naming the return variable"*).

Beyond code readability, developers argue that some optimizations could increase error-proneness. This is what happens for *CS1*. More specifically, the survey respondents think that *"it's a waste of gas to increment a storage variable in a loop"*. However, they report that the optimization reduces code quality as it is *"more error-prone"* (*e.g., "…If the dev changes later the number of loop iteration, he might forget to change the increase of count, which can lead to a bug/vulnerability"*), and they also highlight that the situation where the *"variable won't be used"* inside the loop *"is not common in developing's logic"*.

Instead, according to the interviewed developers, code readability is just one of the concerns related to *CS5* and deriving from the adoption of libraries (*e.g., "This has its place but can also reduce readability"*; *"what is not in the contract code is not transparent on the blockchain"*). Indeed, libraries can *"cause all sorts of issues"* and they are rarely used *"because of the difficulty of managing their deployment"*. In addition, survey participants also highlight that the use of libraries for saving gas mainly depends on the application's context. Indeed, while using libraries could be a good strategy to reduce the deployment cost, it can also increase the execution cost (*e.g., "With a library, you reduce the gas cost of deployment, but you increase the gas cost of executing the contract"*; *" Calls to an external library (paid with each call) may become more expensive than the deployment costs (paid once)"*). Finally, the comments about *CS19* indicate a general reluctance of developers to adopt coding practices that would reduce modularization (*"This discourages good coding practices of modularization and small, well-contained functions"*). In particular, the subjects in our study consider security and readability *"more important than gas usage in most cases"* and keeping the business logic clear is important *"for transparency and verification reasons"* (*e.g., "Devs should aim to create small functions with a clear purpose, rather than complex functions difficult to review and to test. This is another case where the gas-saving does not justify the decrease in code readability imho"*).

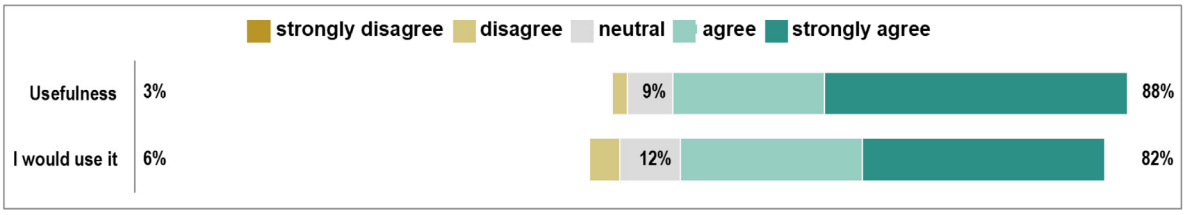


Fig. 2. Survey responses on perceived usefulness of a suite of metrics for identifying cost smells.

Fig. 2 reports, again in the form of diverging stacked bar charts, the results about (i) the perceived usefulness of a suite of metrics for more easily identifying the enumerated smells, and (ii) the willingness of respondents to use it if available. 30 out of 34 respondents (88%) agreed or strongly agreed about the usefulness of a suite of metrics capturing gas consumption information, and 28 participants (82%) indicated that they would use it in the smart contract development process. However, as highlighted by one of the survey participants the *"problem of metrics is that they may become quickly outdated when the compiler changes the way it generates bytecode"*, and optimizing code for gas consumption at the bytecode level would be of broader applicability. However, since smart contract developers desire to optimize source code rather than bytecode (Zou et al., 2019), software metrics can allow them to directly identify the portions of source code that would have room for efficiency improvement.

RQ_1 Summary: *Smart contract developers generally agree or strongly agree about the identified cost smells. However, they point out that some optimizations may negatively impact code readability. The vast majority of respondents are in favor of adopting a suite of source code metrics for more easily identifying cost smells.*

5. The GasMet suite and tool

Relying on the cost smells reported in Table 1, we define a suite of metrics (detailed in Table 2), called GasMet metrics, able to capture information related to each smell. Table 2 reports, for each defined metric, its name, its acronym, its description, the cost smell to which it is related, and the expected direction of the correlation between the metric and the gas consumption. An expected positive (P) correlation between the metric and the gas consumption means that higher values of the metric should correspond to higher gas consumption. On the contrary, an expected negative (N) correlation between the metric and the gas consumption means that higher values of the metric should correspond to lower gas consumption.

It is worth pointing out that some metrics (*e.g.,* DF) consider the lines of code of the smart contract, while some others (*e.g.,* PM) do not. This choice relies on the expected impact on gas consumption. For example, according to CS2, ten public members would have the same impact on the gas consumption independently of the smart contract's lines of code. Contrarily, DF is a proxy to measure code modularization, and, according to CS19, ten defined functions would have different gas consumption impacts in smart contracts with different sizes.

The defined metrics could help in statically measuring the code quality of smart contracts, from the gas consumption perspective.

They could represent a guide for developers for improving this facet of smart contract's quality, as very localized changes might be required to improve the values of the specific metrics. For instance, to achieve lower values of the *ACI* metric (and, consequently, save gas), it is sufficient to reduce the number of variable updates occurring within cycles. More specifically, the defined metrics can help developers detecting (i) gas-inefficient or redundant operations (*e.g.,* ACI, AZ, NLF, RLV), (ii) function visibility (and responsibility) inefficiencies (*e.g.,* PM, EF, IFF), (iii) inefficient usages of data types and structures (*e.g.,* UMA, SB, GV, NU, IP, NM, MA, BV, NE), and (iv) inefficiencies in code modularity (*e.g.,* LI, EC, DF). For example, the function visibility-related metrics can help developers identifying likely opportunities for optimizations (*e.g.,* they could change the modifier of functions that are only used internally from `public` to `internal`, or update the modifier of functions that are not used internally from `public` to `external`). Similarly, as events consume less gas than Solidity variables, the number of events (in combination with the number of variables) represents a good indicator for developers interested in applying optimizations. Indeed, by applying updates to the smart contract's logic, they could identify data (not required on-chain) to store in events and save gas.

For helping smart contract developers more easily identifying likely gas-inefficient code portions in smart contracts written in Solidity, we implemented the GasMet tool. GasMet is a prototype Java tool able to parse Solidity smart contracts and automatically compute the metrics in the GasMet suite. In particular, for properly parsing smart contracts, the tool leverages a generated parser based on a modified version of an existing `antlr4` grammar.[6] Specifically, the tool is composed of three main software modules: the `Lexer`, the `Parser`, and the `GasMet Metrics Calculator`. The `Lexer` receives as input the raw text of the Solidity smart contract and provides a stream of tokens, relying on lexical rules. The `Parser` processes this stream and builds an abstract syntax tree. The `GasMet Metrics Calculator` traverses the tree generated by the `Parser` to compute the GasMet metrics. Once processed the smart contract, the `GasMet Metrics Calculator` stores the results in a tabular format. The results can also be exported as a CSV file.

The tool either provides a graphical user interface (GUI) or can be used from the command line. The instructions for running the GUI or using it from the command line are provided in our replication package.[7] In particular, the tool's GUI is built as a web application that can be deployed by using the Wildfly application server. Once selected a Solidity smart contract to analyze, the tool's GUI presents two frames (see Fig. 3). The left frame reports a table containing the results of the GasMet metrics computation on the selected smart contract. More specifically, the aforementioned table encompasses the following information about each metric: acronym, extended name, description, computed value, and lines involved in the computation. In the right frame, the source code of the smart contract under analysis is shown. The lines involved in the computation of the different GasMet metrics are highlighted in red to allow developers and researchers more easily identifying the code portions that could likely require improvements. A developer interested in saving the gas required for the deployment of her smart contract could analyze the smart contract through the GasMet tool and apply improvements to

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

[7] https://github.com/paperSubmition2020/GasmetReplicationPackage/

Table 2
Metrics' descriptions and correlations.

Metric's name	Abbr.	Description	CS	Corr.
Assignments within Cycles	ACI	It computes the assignments and/or variable updates occurring within loops.	CS1	P
Public Members	PM	It enumerates the functions defined as public members.	CS2	P
External Functions	EF	It enumerates the functions defined as external.	CS3	N
Assignments to default values	AZ	It computes the assignments to default values during variable definitions.	CS4	P
Library Imports	LI	It estimates the usage of external libraries. Given the number of *import* statements, S_{import}: $LI = S_{import}$	CS5	N
Internal Functions	IFF	It computes the ratio of *internal* functions on the total number of defined functions. Given the number of *internal* functions, $F_{internal}$, and the number of overall functions, F_{all}, defined in the contract: $IFF = \frac{F_{internal}}{F_{all}}$	CS6	N
Uses of Memory Arrays	UMA	It computes the ratio of *memory* arrays on the total number of defined arrays within the contracts. Given the number of arrays defined as *memory*, A_{memory}, and the total number of defined arrays, A_{all}: $UMA = \frac{A_{memory}}{A_{all}}$	CS7	N
Strings and Bytes	SB	It computes the occurrences of *string* variables with respect to the occurrences of *bytes*. Given the number of *string* variables, V_{string}, and the number of *bytes* variables, V_{bytes}: $SB = \frac{V_{string}}{(V_{bytes} + V_{string})}$	CS8	P
Functions Returning Local Variables	RLV	It computes the ratio of functions returning local variables on the total number of functions. Given the number of functions returning local variables, F_{local}, and the overall number of functions, F_{all}, defined in the contract: $RLV = \frac{F_{local}}{F_{all}}$	CS9	P
Global Variables	GV	It computes the number of global variables.	CS10	P
Number of Loops	NLF	It computes the number of loops inside the contract. In the case of a code block with two (or more) nested loops, the NLF metric will count two (or more) loops.	CS11	P
Number of non-32-bytes variables	NU	It computes the ratio of non-32-bytes type variables on the total number of variables.	CS12	P
Indexed Parameters	IP	It computes the number of parameters declared as *indexed* within events.	CS13	P
Number of Mappings	NM	It computes the occurrences of *mapping* with respect to occurrences of instance variables.	CS14	P
Mappings and Arrays	MA	It computes the ratio of *mappings* with respect to the sum of mappings and *arrays* defined in the contract. Given the occurrences of mappings, $N_{mappings}$ and the occurrences of arrays, N_{arrays}: $MA = \frac{N_{mappings}}{(N_{mappings} + N_{arrays})}$	CS15	N
External Calls	EC	It computes the ratio of calls to external functions (*i.e.,* not defined in the contract) on the total number of function calls. Given the occurrences of calls to external functions, $C_{external}$, and the total number of calls within the contract, C_{all}: $EC = \frac{C_{external}}{C_{all}}$	CS16	P
Boolean Variables	BV	It computes the ratio of *boolean* variables on the number of total variables. Given the number of variables of the boolean type, V_{bool}, the number of overall variables, V_{all}: $BV = \frac{V_{bool}}{V_{all}}$	CS17	N
Number of Events	NE	It computes the number of *events*.	CS18	N
Defined Functions	DF	It computes the number of the functions with respect to the overall lines of code. Given the number of overall defined functions, F_{all}, and the amount of lines of code, LOC: $DF = \frac{F_{all}}{LOC}$	CS19	P

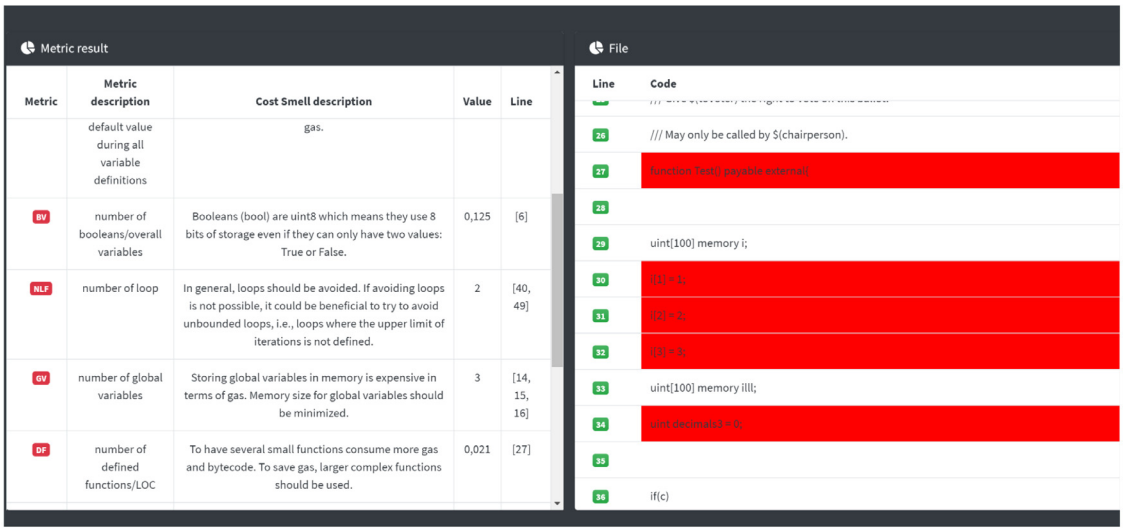


Fig. 3. GasMet tool's usage example.

achieve better values for the computed metrics. There are many different tools available for Ethereum smart contracts. These tools have been gathered from research publications and through Internet searches. To make a comparison with the GasMet tool, we selected the tools that are actively maintained, open-sourced, ready for use, such as Remix-IDE[8] and SmartCheck[9]. Unlike our tool, Remix-IDE is a browser-based IDE for developing Solidity contracts. It can connect to the Ethereum network using Metamask[10] and developers can directly deploy smart contracts from Remix. During compilation it is able to report security issues, indicating where they occurred in the code. It also reports implicit visibility, unchecked return values, implicit typing, deprecated constructs, and address checksums. The static analysis is only lightweight and includes some control flow analysis. Remix-IDE also enables the testing of smart contracts via unit tests written using tape.[11] Remix-IDE, for the Solidity static analysis, is not based on a suite of metrics but computes the gas consumption associated with each Ethereum Assembly instruction listed in the Ethereum Yellow-Paper/AppendixG[12]. SmartCheck is a static analysis tool for smart contracts written in Solidity and Vyper. It is developed by SmartDec and the University of Luxembourg. Like other static analysis tools, it works at the source code level. In particular, it transforms the source code into an XML-based intermediate representation. This representation is then checked against XPath patterns to highlight potential vulnerabilities in the code.

6. Study on the relations between GasMet metrics and deployment costs

The *goal* of this second study is to more-in-depth investigate the relationships between the individual metrics in the GasMet suite and smart contracts' deployment costs. More specifically, this investigation aims at assessing whether our suite can capture gas-related information by analyzing smart contracts' source code. To pursue this goal, we pose our second research question:

RQ_2: To which extent does the GasMet suite correlate with gas consumption?

6.1. Context selection and data extraction

For this study, we collected a dataset containing the source code of 2186 Solidity smart contracts deployed on Ethereum. Actually, there are about 1.5 million smart contracts deployed on Ethereum (Pierro et al., 2020). However, for many of these smart contracts, the source code is not publicly available (Zhou et al., 2018). For this reason, we leveraged Etherscan[13], a popular service for Ethereum blockchain exploration that offers a feature called *"verified contracts"*, through which developers can publish the source code of blockchain smart contracts. The Etherscan API actually allows obtaining the source code of more than 40,000 smart contracts (Oliva et al., 2020). As we needed to compile and deploy the extracted contracts to estimate the deployment costs and this requires a discrete amount of time, for reducing the experimentation time, we decided to conduct our study on a (statistically significant) set of contracts randomly sampled from the initial collection returned by Etherscan. It is worth noticing that the number of instances in our dataset was defined for guaranteeing high representativeness. Indeed, our dataset, beyond being a statistically significant sample of the Etherscan collection (*i.e.,* a confidence level of 99% and a margin of error smaller than 3%), it is also a statistically representative sample of all the smart contracts deployed on Ethereum (*i.e.,* a confidence level of 99% and a margin of error smaller than 3%). It is worth noticing that the margin of error was computed according to the formula for margin of error with finite population correction (Salkind, 2010):

$$Margin\ of\ error = z * \sqrt{\frac{p * (1-p)}{(N-1) * \frac{n}{N-n}}}$$

where z is the z-score associated with the confidence level ($z = 2.576$ in the case of a confidence level of 99%), p is the sample proportion ($p = 0.5$ in the case of random sampling), n is the sample size ($n = 2186$ in our case), and N is the population size.

For each smart contract in our dataset, we extracted the values of the GasMet metrics, by using the parser we developed (see Section 5).

Table 3 groups the smart contracts considered in our dataset according to the number of lines of source code (SLOC). It is worth noticing that about 73% of the analyzed contracts have a size expressed in lines of source code between 50 and 500, while only a limited number of them (i.e., about 6%) exhibit higher values of SLOC.

8 https://remix-ide.readthedocs.io/en/latest/
9 https://github.com/smartdec/smartcheck
10 https://metamask.io/
11 https://www.npmjs.com/package/tape
12 https://ethereum.github.io/yellowpaper/paper.pdf
13 https://etherscan.io/

Table 3
Lines of source code of the analyzed contracts.

# of source code lines	#instance	%instance
SLOC $<$ 50	462	21.1
50 $\leq$ SLOC $<$ 100	800	36.6
100 $\leq$ SLOC $<$ 500	794	36.3
SLOC $\geq$ 500	130	5.9

To collect data on gas consumption, we used the built-in smart contract compilation. The compilation process for smart contracts involves the Truffle suite (Group, 2019b) and the Ethereum client Ganache (Group, 2019a) where it gets deployed. Truffle Suite is a collection of tools for the development and testing of Ethereum blockchain based software. It contains Truffle which is the most popular development and testing framework using the Ethereum Virtual Machine (EVM). The compilation and deployment pipeline of Ethereum smart contract, that we used, is as follows:

(1) *setting up an Ethereum development environment*: we create a Truffle project, read smart contracts from dataset, create a deployment script that deploys and initializes the state of deployed contracts on blockchain specified in the project config file, and
(2) *collect data regarding gas consumption*: Ganache is a local test blockchain included in the above mentioned Truffle Suite. The gas consumption, on local blockchain, is computed by: *gasCost* * *gasPrice*. *gasPrice* represents the price to pay per gas unit to deploy the contracts under Truffle project and was set, in our experimentation, to the value of 1 Wei (1 Ether = 10^{18} Wei). *gasCost* is the maximum number of gas unit the EVM can use to process the contract deployment transaction.

Replication package. All the analyzed contracts source code (`.sol` files), as well as the metrics' results, are made publicly available in our replication package.[14]

6.2. Analysis method

For answering RQ_2, we considered the values of GasMet metrics computed on the smart contracts in our dataset and the related values of gas consumption collected by deploying such smart contracts (see Section 6.1). Similar to previous work (Ajienka et al., 2020), we investigated the correlation between source code metrics computed on several smart contracts and the resources needed for their deployment (i.e., *gasUsed*), as well as the correlations between each pair of metrics. More specifically, to evaluate if significant relationships can be found between (i) each of the GasMet metrics and the gas consumption, and (ii) each pair of GasMet indicators, we used the Spearman rank correlation coefficient (Daniel, 1990), fixing the p-value ≤ 0.05 and adopting the Holm's p-value correction procedure (Holm, 1979) to deal with multiple comparisons. We ran statistical significance tests, with $\alpha = 0.05$, for being sure that there is at maximum a 5% probability that the strength of the relationship found (the ρ coefficient) happened by chance (when p-value ≤ 0.05). In particular, we tested the following null hypothesis:

H_0: *There is no monotonic association between the metrics* m_i *and* m_j

where m_i and $m_j \in$ {*GasCost, ACI, PM, EF, AZ, IFF, UMA, SB, RLV, GV, NLF, NU, IP, NM, MA, EC, BV, NE, DF*} and $i \neq j$. We interpret the strength of the correlation as (i) *small* for $0.10 \leq |\rho| \leq 0.29$, (ii) *medium* for $0.30 \leq |\rho| \leq 0.49$, and (iii) *large* for $|\rho| \geq 0.50$, as recommended by Cohen's standard (Cohen, 1988).

To better understand which of the metrics in our suite might affect gas consumption more, we performed a Random Forest (RF) regression analysis. To perform this analysis, we considered a dataset containing the values of all the GasMet metrics computed on the 2186 smart contracts selected for our study (see Section 6.1) and the corresponding values of gas consumption collected by deploying such smart contracts. In particular, we tried to predict gas consumption (*i.e.,* dependent variable) based on the values of the GasMet metrics (*i.e.,* independent variables). In this analysis, we leveraged the `randomForest` (Liaw and Wiener, 2002) package from RStudio to estimate the performance of the models and the importance of variables. We used the Random Forest regression method as it works well with almost all types of data, generally does not overfit, and it is easy to get the relative importance of the predictor variables from a trained model (Dey and Mockus, 2020). Specifically, we first performed a grid search to find the best model parameters (i.e., `ntree`, the number of trees to grow, and `mtry`, the number of variables randomly sampled as candidates at each split), by using 10-fold cross-validation. The best model (i.e., the one with lowest Mean Square Error) was obtained by setting `ntree` = 500 and `mtry` = 8. Then, the following steps were performed:

1. We split the initial dataset into a training set T_{train} (*i.e.,* 75% of the initial dataset) and the corresponding test set T_{test} (*i.e.,* 25% of the initial dataset) at random.
2. The RF algorithm (with `ntree` = 500 and `mtry` = 8) was trained (on T_{train}) and tested by predicting the samples in T_{test}. Popular metric were used to assess the performance of the model: Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and the Coefficient of determination (R^2) (Sammut and Webb, 2011). In particular, MAE measures the average of the residuals (*i.e.,* the absolute difference between actual and predicted values), RMSE represents the standard deviation of residuals, and R^2 is the proportion of the variance in the dependent variable (*i.e.,* the gas consumption) which is explained by the model.
3. We estimated the importance of the different metrics by considering the samples in T_{train} and the percentage increase in Mean Square Error (*i.e.,* %IncMSE) of each independent predictor (*i.e.,* each GasMet metric). The %IncMSE metric represents the deterioration of the predictive ability of the model when a predictor is randomly permuted and the other predictors remain unchanged (Strobl et al., 2008). In particular, each tree in a random forest has its out-of-bag (OOB) sample of data that was not used during construction. This sample is used to compute the importance of a specific variable. For each tree, the prediction error (MSE) on the out-of-bag portion of the data is recorded. Then the same is done after randomly shuffling the values related to a specific GasMet metric, keeping all other variables the same. The differences between the two MSE values (*i.e.,* MSE on correct data and MSE on permuted data) obtained for each tree are then averaged over all trees and normalized according to the standard deviation of the differences (Liaw and Wiener, 2002). Finally, the percentage increase in MSE (%IncMSE) on the shuffled data is measured. This procedure has been applied for all the 19 GasMet metrics. The higher the %IncMSE is, the higher the importance of the respective predictor in relation to the target variable is (Sabatti and Lalanne, 2009).

To reduce sampling bias and obtain more reliable results, we repeated steps 1, 2, and 3 ten times. Therefore, all the results were averaged over the ten runs.

[14] https://github.com/paperSubmition2020/GasmetReplicationPackage

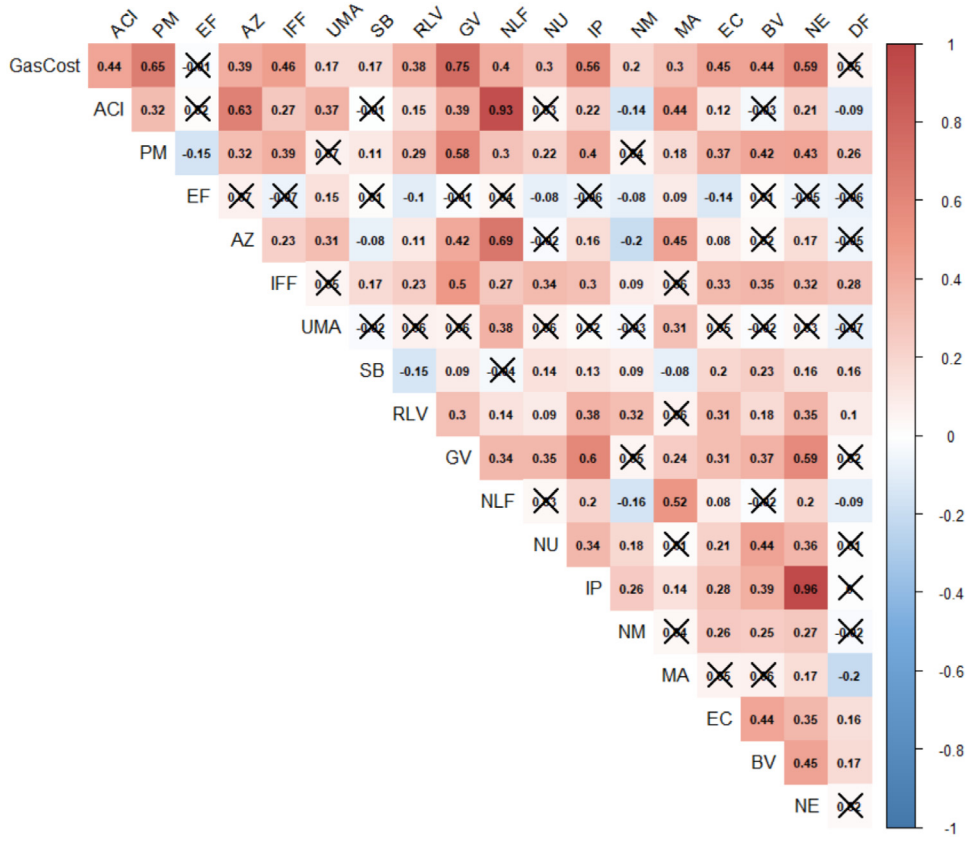


Fig. 4. Spearman correlation results among the metrics belonging to the GasMet suite and the gas cost (values with *x* above is used to indicate the correlations that are not statistically significant).

6.3. Results

Fig. 4 reports the results of the Spearman correlation between each pair of the metrics in the GasMet suite and between the GasMet metrics and gas consumption. As specified in Sections 6.1 and 6.2, for each smart contract in our dataset, we used the GasMet tool to compute the GasMet metrics, while we leveraged Truffle (for compiling and deploying the smart contract) and Ganache (a locally deployed blockchain simulator) to collect data about gas consumption. Once obtained the metrics and gas consumption values for all the smart contracts in our dataset, we implemented a script in the R language (using the `corrplot`[15] library) to compute the correlation values. It is worth noticing that in Fig. 4 we do not consider the *LI* metric, as when computing this metric on the instances in our dataset, all zero values have been obtained.

All the results considered in the following discussion correspond to an adjusted p-value ≤ 0.05.

Four metrics of the GasMet suite are more strongly linked to the gas consumption (in accordance with the Cohen's standard), since their correlations with the gas cost have a *large* effect size, which are: (i) the occurrences of public members (*PM*, with $\rho = 0.65$), (ii) the number of global variables (*GV*, with $\rho = 0.75$), (iii) the number of indexed parameters (*IP*, with $\rho = 0.56$), and (iv) the number of events (*NE*, with $\rho = 0.59$). Higher PM values correspond to a higher gas consumption since public members are hash-sorted, and the sorting algorithm entails a certain fixed quantity of gas (depending on the EVM transactions) for each position in the list. Global variables cause an additional cost due to the storing of information within the smart contract's state on the blockchain, so their usage must be reduced and replaced with local variables which are not stored on the blockchain. Similarly, indexed parameters consume more gas than non-indexed parameters, while, if properly used, events could be adopted to store data that is not required on-chain.

With regards to the metrics that exhibited a *medium* effect size in the correlation with the gas cost, nine should be mentioned: (i) the ratio of internal functions (*IFF*) with $\rho = 0.46$, (ii) the ratio of external calls (*EC*) with $\rho = 0.45$, (iii) the ratio of boolean variables (*BV*) with $\rho = 0.44$, (iv) the assignments within cycles (*ACI*) with $\rho = 0.44$, (v) the number of loops (*NLF*) with $\rho = 0.40$, (vi) the assignments to default values (*AZ*) with $\rho = 0.39$, (vii) the number of functions returning local variables (*RLV*) with $\rho = 0.38$, (viii) the number of non-32-bytes variables (*NU*) with $\rho = 0.30$, and (ix) the ratio of mappings (*MA*) with $\rho = 0.30$.

As reported in Table 4, among the metrics exhibiting *large* relationships with gas consumption, for *GV*, *IP*, and *NE* we observe that the majority of surveyed developers (*i.e.*, $>$ 58%) agree or strongly agree on the relevance of the related smells (*i.e., CS10, CS13,* and *CS18*). Only 47% of them agree or strongly agree on the relevance of the *CS2* smell (related to the *PM* metric). However, as reported in Section 4.2, the developers acknowledge that the *efficient usage of public members can save gas*, but they believe that the *optimizations should be performed at the Ethereum or compiler level*. Similarly, among the metrics exhibiting *medium* relationships with gas consumption, we observe that, for most of them (*i.e., ACI, IFF, RLV, NLF, NU, MA,* and *EC*), the majority of surveyed developers (*i.e.*, $>$ 58%) agree or strongly agree on

15 https://www.rdocumentation.org/packages/corrplot/versions/0.90.

Table 4
Survey responses and correlations between GasMet metrics and gas consumption.

Metric	Correlation with gas consumption (Spearman's ρ)	Cost smell (CS)	% of survey respondents who agreed on the CS relevance
ACI	$\rho = 0.44$	CS1	59%
PM	$\rho = 0.65$	CS2	47%
EF	not significant	CS3	88%
AZ	$\rho = 0.39$	CS4	50%
LI	not significant	CS5	56%
IFF	$\rho = 0.46$	CS6	88%
UMA	$\rho = 0.17$	CS7	85%
SB	$\rho = 0.17$	CS8	76%
RLV	$\rho = 0.38$	CS9	65%
GV	$\rho = 0.75$	CS10	91%
NLF	$\rho = 0.40$	CS11	88%
NU	$\rho = 0.30$	CS12	59%
IP	$\rho = 0.56$	CS13	82%
NM	$\rho = 0.20$	CS14	85%
MA	$\rho = 0.30$	CS15	65%
EC	$\rho = 0.45$	CS16	85%
BV	$\rho = 0.44$	CS17	41%
NE	$\rho = 0.59$	CS18	59%
DF	not significant	CS19	50%

the relevance of the related smells (*i.e., CS1, CS6, CS9, CS11, CS12, CS15,* and *CS16*). In contrast, only 50% of them agree or strongly agree on the relevance of the *CS4* smell (related to the *AZ* metric), and 41% of the participants in our survey agree or strongly agree on the relevance of the *CS17* smell (related to the *BV* metric). As reported in Section 4.2, although developers recognize that *CS4* and *CS17* smells could be sources of gas wasting, they rather worry about the risk of hampering code readability that could derive from the optimizations. Curiously, the majority of surveyed developers agree or strongly agree on the relevance of the cost smells related to metrics exhibiting *small* or not significant relationships with gas consumption (*i.e., EF, UMA, SB, NM,* and *DF*). Thus, further investigation is needed to understand whether it is possible to define better metrics for capturing the occurrences of the corresponding smells (*i.e., CS3, CS7, CS8, CS14,* and *CS19*).

Five correlations with gas consumption have been observed to be weak or not significant (*i.e.,* UMA, SB, NM, MA, and DF). Using a *memory* array (UMA), as it happens with any other kind of variable, means that the values of the data structure are not saved in *storage*. *Storage* in Solidity refers to a mechanism that holds data between function calls, while *memory* keyword makes Solidity to create a chunk of space for the variable at method runtime, guaranteeing its size and structure for future use in that method. By using a metaphor, *storage* could be seen as a hard drive, while *memory* as RAM. However, in our dataset we observed a very low usage of *memory* arrays (for 2112 smart contracts in our dataset the value of this metric is 0). This could explain why UMA is weakly correlated with the gas cost. Concerning the usage of `string` type instead of `bytes` type (SB), we expected different results. Consequently, we did not find quantitative evidence about this smell. Also in this case, for the majority of the smart contracts in our dataset, we observe that the value of this metric is 0. Since the results of the survey supports CS8, we can conclude that this smell deserves further investigation. Mappings are lightweight data structures, free of additional functions like the length computation, bound checking, and optimization. Similar to the other metrics exhibiting weak correlation with gas consumption, we observe that for many smart contracts in our dataset we report a MA value equal to 0. This could explain why MA resulted as weakly correlated with the gas cost. On the contrary, the NM metric assumes non-zero values for the great majority of smart contracts in our dataset

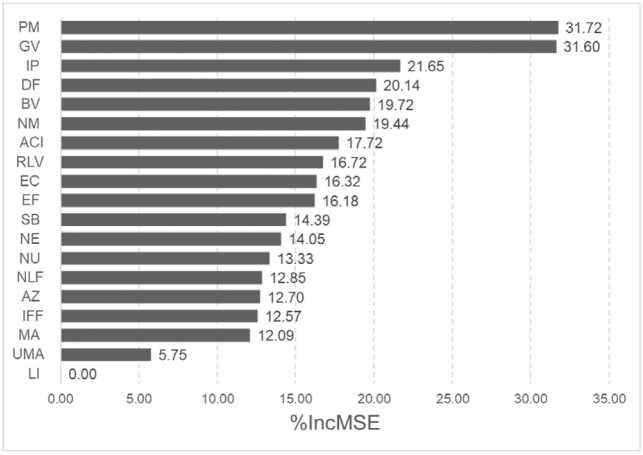


Fig. 5. Importance of variables used in random forest modeling.

and we believe that further investigation is needed to understand the reason behind the weak correlation between the number of mappings and the gas consumption. The number of defined functions (DF) is not significantly correlated with the gas cost, while the proliferation of public members (PM) might affect gas consumption significantly.

By observing the pairwise correlations between the metrics, we can discover the developers' habits in writing smart contracts that can negatively influence gas consumption. Among the correlations with a *large* effect size, we can observe (i) the one between the Number of Indexed Parameters (*IP*) and Number of Events (*NE*), with $\rho = 0.96$, and (ii) the one between the Assignment Within Cycles (*ACI*) and Number of Loops (*NLF*), with $\rho = 0.93$. From these two relationships it emerges that two bad practices are pretty common: the first one is the use of indexed parameters within the events, while the second one consists of valuing variables inside the cycles. As previously mentioned, both the practices lead to an increasing of gas cost, and should be discouraged. The correlation between *ACI* and *AZ*, with $\rho = 0.63$, suggests that assignments to default values often occur within the cycles and this trend is also confirmed by the *large* correlation between *NLF* and *AZ* ($\rho = 0.69$). The *large* effect size ($\rho = 0.52$) of the correlation between the Number of Loops (*NLF*) and the Mappings and Array (*MA*) metric also implies that loops are frequently used to iterate over mappings. Fig. 4 also shows the correlations between the various types of functions. In particular, focusing on the metrics modeling internal (IFF), external (EF), public (PM), and defined (DF) functions, we only observe a correlation with *medium* effect size ($\rho = 0.39$) between the PM-IFF pair, while for all the other pairs only small or not significant correlations occur. Finally, the number of global variables (*GV*) usually grows when higher numbers of (i) public members (*i.e., PM*, $\rho = 0.58$), (ii) internal functions (*i.e., IFF*, $\rho = 0.50$), (iii) events (*i.e., NE*, $\rho = 0.59$), or (iv) indexed parameters (*i.e., IP*, $\rho = 0.60$) are adopted.

As reported in Section 6.2, we used Random Forest regression to predict gas consumption based on the values of the GasMet metrics. This analysis allowed us to estimate the importance of the different metrics in predicting gas consumption. Specifically, the Random Forest regression algorithm achieved an average MAE of 0.0042, an average RMSE of 0.0096, and an average R^2 of 0.71. This means that, on average, the trained models can explain more than 70% of the variance in the gas consumption. The analysis of variable importance (see Fig. 5) shows that all the metrics except UMA and LI exhibit an increase in Mean Square

Error (MSE) higher than 12%. More specifically, the most important variables influencing the gas consumption are the number of public members (PM) and the number of global variables (GV), both increasing the MSE by more than 30% (31.72% and 31.60%, respectively) when randomly permuted. It is worth noticing that PM and GV are also the metrics exhibiting the highest correlation coefficient values with gas consumption. This result confirms that the numbers of public members (PM) and global variables (GV) might both have a significant impact on the gas consumption predictions. Thus, they are the most important metrics developers should monitor. In addition, we also report an increase of MSE higher than 20% when the number of indexed parameters (IP) or the number of defined functions (DF) are randomly permuted.

RQ_2 Summary: *Thirteen metrics of the GasMet suite exhibit large (PM, GV, IP, NE) or medium (ACI, AZ, IFF, RLV, NLF, NU, MA, EC, BV) correlations with gas consumption required by smart contract deployment. The correlations between the pairs of GasMet metrics allow identifying frequent coding patterns that influence gas consumption. PM and GV are the most important metrics to monitor, as they might both affect gas consumption estimations significantly.*

7. Threats to validity

Threats to construct validity concern the relationship between theory and observation. The most important threat that could affect our results is related to possible imprecision/incompleteness in identifying *cost smells*. In particular, such smells have been identified by relying on information encompassed in specialized forums/books focused on the development of Solidity smart contracts. Thus, some of the identified smells (and, consequently, the related metrics in our suite) could be related to anecdotal observations. To partially mitigate this weakness, in our RQ_1, we studied if domain experts (*i.e.,* smart contract developers) perceive such smells as relevant. The majority of survey respondents generally agree with the identified smells. However, we asked developers to rate the importance of smells by only providing short descriptions of the problems. To counteract this issue, such descriptions were accompanied by some explanatory examples. Indeed, no respondents indicated possible misunderstanding in the questions.

Threats to conclusion validity concern the relationship between treatment and outcome. Appropriate, non-parametric statistical procedures have been adopted to draw our conclusions concerning RQ_2. More specifically, we used the Spearman rank correlation coefficient, to investigate the relationships between the different metrics in the GasMet suite and the gas cost. To cope with multiple tests, Holm's correction procedure has been adopted to adjust p-values.

Threats to internal validity concern factors that can affect our results. The smart contract dataset considered in our study comprises small to medium size smart contracts (see Table 3), and this could have reduced the likelihood of specific cost smells being present in a given smart contract. In particular, such an issue may have hindered the statistical relevance of some tests. Indeed, there are three metrics (*i.e.,* UMA, SB, and NM) for which we only found a *small* effect size in the correlation with gas consumption ($\rho < 0.3$), while our results show that for two metrics (*i.e.,* DF and EF) the correlation with gas consumption was not statistically significant ($p > 0.05$). However, the results of the analysis for estimating the importance of the GasMet metrics when used to predict gas consumption (see Fig. 5) showed that all the aforementioned metrics (with the exception of UMA), if randomly permuted, might have a significant impact on the gas consumption predictions (*i.e., %IncMSE* $> 12\%$). To cope with this problem, in the future, we plan to replicate our study at a larger scale by also considering smart contracts of larger size.

Threats to external validity concern the generalization of the findings. The set of identified cost smells is surely incomplete. Further research is needed to more-in-depth explore broader sets of Solidity coding practices that can negatively influence gas consumption. Indeed, our work jointly attempts to (i) tackle the problem of cost smells in Solidity source code, and (ii) conceive approaches to help developers more easily identifying them during smart contract development. With regards to RQ_2, our study has been carried out on a data collection comprising 2186 real-world Solidity smart contracts for which Etherscan provides the source code. The smart contracts in our dataset may be not representative of all smart contracts deployed on the blockchain, and some of the findings may depend on the specific data we used. For partially alleviating this threat, we collected a dataset that is (i) a statistically significant sample of the smart contracts for which Etherscan provides the source code, and (ii) sufficiently large to be representative of the smart contracts actually deployed on Ethereum. However, while this study is only observational, in the future we plan to carry out experiments at a larger scale to verify the generalizability of the obtained results. The Ethereum platform and Solidity are constantly evolving at a fast pace (Wohrer and Zdun, 2018) and future optimizations might be applied in opcodes and/or in the compilation process of Solidity smart contracts. Clearly, the latter could have effects on the relationships exhibited by some of the metrics in our suite and the gas consumption. Furthermore, as our research is not exhaustive, in the future, additional metrics better outlining the code quality of smart contracts (from the gas consumption perspective) may be identified.

8. Conclusion

Although Blockchain technology has been established as the enabling layer for allowing the transactions of electronic cash, namely cryptocurrency, without the brokerage of a financial institution, it is now increasingly applied to many other domains. One of the key aspects to govern when developing a distributed application (dApp), i.e. an application built on the top of a DLT, is the cost of execution that, if not properly limited, can easily lead to relevant diseconomy, especially considering the issues related to guaranteeing the service levels when scaling up the distributed application. The back-end logic of a dApp is defined in smart contracts that run on the blockchain. Currently, to the best of authors' knowledge, there are no tools that can be used *while* coding to help developers properly identifying the code segments that need optimizations for achieving lower gas consumption. Considering that the choices are done by the developer while writing the smart contracts can affect the deployment and execution costs, we identified 19 patterns of code that can increase (or reduce) the gas consumption, namely *cost smells*. Through a survey involving real smart contract developers, we demonstrated that the majority of respondents perceive 15 out of 19 smells as relevant. The vast majority of respondents also agree or strongly agree on the usefulness of a suite of metrics for more easily identifying such cost smells.

On top of the identified smells, we defined a set of metrics, namely the *GasMet* suite, in which each metric tries to capture the occurrences of a cost smell. Through a study involving 2186 smart contracts, we empirically demonstrate that a subset of GasMet strongly correlates with the gas consumption, namely *GV*, *PM*, *IP*, and *NE*, while associations with *medium* effect size are observed between a further subset of the defined metrics, namely *ACI, AZ, IFF, RLV, NLF, NU, MA, EC*, and *BV*, and the deployment cost. Our suite can be acquired as a tool for allowing developers to

optimize smart contracts by easily localizing cost smells and improving the related code segments. In particular, since we found significant links between the metrics in our suite and deployment costs, the proposed metrics can be beneficial for especially novice smart contract developers (Ajienka et al., 2020). *GasMet* metrics will guide inexperienced developers on source code portions that could be modified or refactored for reducing deployment cost. Besides, since the *GasMet* suite statically collects smart contract-related metrics, it can also be used by developers and project managers as a tool for evaluating the code quality of alternative solutions from the gas consumption perspective.

This paper provides several contributions to the research community:

- a catalog of cost smells for Solidity programming language, whose relevance has been assessed through a survey involving smart contract developers (see Sections 3 and 4);
- a corresponding measurement suite, namely *GasMet* for easily identifying cost smells while coding (see Section 5);
- an empirical evaluation of the GasMet suite, along with a corpus of smart contracts that can be used for further experiments, accessible from our replication package; and
- a tool that computes the GasMet metrics by statically inspecting the Solidity code of smart contracts (see Section 5).

While this study aims at (i) evaluating the perceived relevance of the identified cost smells, and (ii) proposing a set of metrics for better estimating the gas required for the deployment of the smart contracts, future work will focus on empirically setting accurate thresholds for these metrics. Since establishing robust threshold values for source code metrics is not a trivial task (Fontana et al., 2011), further research is needed to enable the accurate detection of the proposed smells. Actually, some of the identified code smells could be fixed directly by the Solidity compiler, which could lift the burden of the modifications off of the developer. Of course, not all the changes can be automated, since in some cases, keeping a smell might be beneficial in the economy of the overall system. For this reason, in the future, we want to investigate which code smells could be solved as compiler optimization and how to perform such optimizations. As future work, we also plan to further refine our metrics to help developers more easily applying gas consumption optimizations. In addition, future analyses will be aimed at more closely exploring the relationships existing between the increasing number of code lines and the specific degradations that might arise in gas consumption. Furthermore, since the cost smells defined in this work are strictly dependent on the Solidity programming language, as future work, we will investigate if more general design practices, not related to a specific programming language, could be defined to achieve savings in gas consumption.

CRediT authorship contribution statement

Andrea Di Sorbo: Conceptualization, Methodology, Investigation, Visualization, Formal analysis, Writing – original draft, Writing – review & editing. **Sonia Laudanna:** Software, Methodology, Investigation, Validation, Visualization, Data curation, Formal analysis, Writing – original draft, Writing – review & editing. **Anna Vacca:** Software, Methodology, Investigation, Validation, Visualization, Data curation, Formal analysis, Writing – original draft, Writing – review & editing. **Corrado A. Visaggio:** Conceptualization, Methodology, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Gerardo Canfora:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank all the participants in our survey. We gratefully thank Michele Fredella for his valuable help during the development of the GasMet tool.

References

Ajienka, N., Vangorp, P., Capiluppi, A., 2020. An empirical analysis of source code metrics and smart contract resource consumption. J. Softw. Evol. Process. 32 (10).

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., Rubio, A., 2020. GASOL: Gas analysis and optimization for ethereum smart contracts. In: Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held As Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II, pp. 118–125.

Albert, E., Gordillo, P., Rubio, A., Sergey, I., 2019. Running on fumes - preventing out-of-gas vulnerabilities in ethereum smart contracts using static resource analysis. In: Verification and Evaluation of Computer and Communication Systems - 13th International Conference, VECoS 2019, Porto, Portugal, October 9, 2019, Proceedings, pp. 63–78.

Aniche, M.F., Bavota, G., Treude, C., Gerosa, M.A., van Deursen, A., 2018. Code smells for model-view-controller architectures. Empir. Softw. Eng. 23 (4), 2121–2157.

Anonymous, 2019. Blockchain technology market size, share, trends analysis report by type, by component, by application, by enterprise size, by end use, by region and segment forecasts, 2019 - 2025 . Online; https://www.reportlinker.com/p05807295/Blockchain-Technology-Market-Size-Share-Trends-Analysis-Report-By-Type-By-Component-By-Application-By-Enterprise-Size-By-End-Use-By-Region-And-Segment-Forecasts.html?utm_source=PRN. (Accessed 29 February 2020).

Badr, B., Horrocks, R., Wu, X.B., 2018. Blockchain By Example: A Developer's Guide To Creating Decentralized Applications using Bitcoin, Ethereum, and Hyperledger. Packt Publishing Ltd.

Baird, K., Jeong, S., Kim, Y., Burgstaller, B., Scholz, B., 2019. The economics of smart contracts. arXiv preprint arXiv:1910.11143.

Bentley, J.L., 1982. Writing Efficient Programs. Prentice-Hall, Inc..

Bhushan, V.A., 2018. Optimizing smart contracts for cost. Online; https://labs.imaginea.com/optimizing-smart-contracts-for-cost/. (Accessed 21 February 2020).

Blitz, N., 2018. Storing structs is costing you gas. Online; https://medium.com/@novablitz/storing-structs-is-costing-you-gas-774da988895e. (Accessed 21 February 2020).

Borrelli, A., Nardone, V., Di Lucca, G.A., Canfora, G., Di Penta, M., 2020. Detecting video game-specific bad smells in unity projects. In: MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020, pp. 198–208.

Brandstätter, T., Schulte, S., Cito, J., Borkowski, M., 2020. Characterizing efficiency optimizations in solidity smart contracts. In: 2020 IEEE International Conference on Blockchain. Blockchain, IEEE, pp. 281–290.

Canfora, G., Di Sorbo, A., Fredella, M., Vacca, A., Visaggio, C.A., 2021. iSCREAM: a suite for smart contract readability assessment. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 579–583.

Chen, Y.-C., 2018a. [Solidity] how does function name affect gas consumption in smart contract. Online; https://medium.com/joyso/solidity-how-does-function-name-affect-gas-consumption-in-smart-contract-47d270d8ac92. (Accessed 21 February 2020).

Chen, Y.-C., 2018b. [Solidity] optimize smart contract gas usage. Online; https://medium.com/joyso/solidity-save-gas-in-smart-contract-3d9f20626ea4. (Accessed 21 February 2020).

Chen, T., Feng, Y., Li, Z., Zhou, H., Luo, X., Li, X., Xiao, X., Chen, J., Zhang, X., 2020a. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. IEEE Trans. Emerg. Top. Comput..

Chen, T., Li, X., Luo, X., Zhang, X., 2017. Under-optimized smart contracts devour your money. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 442–446.

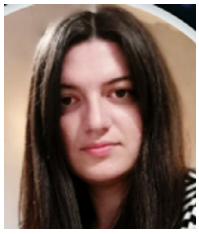
Chen, X., Liao, P., Zhang, Y., Huang, Y., Zheng, Z., 2021. Understanding code reuse in smart contracts. In: 28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021, pp. 470–479.

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T., 2020b. Defining smart contract defects on ethereum. IEEE Trans. Softw. Eng..
Cipher, Z., 2018. Optimizing your solidity contract's gas usage. Online; https://medium.com/coinmonks/optimizing-your-solidity-contracts-gas-usage-9d65334db6c7. (Accessed 21 February 2020).
Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences, second ed. Lawrence Erlbaum Associates, Publishers.
Correas, J., Gordillo, P., Román-Díez, G., 2021. Static profiling and optimization of ethereum smart contracts using resource analysis. IEEE Access 9, 25495–25507.
Daniel, W.W., 1990. Spearman rank correlation coefficient. In: Applied Nonparametric Statistics, second ed. PWS-Kent, Boston, pp. 358–365.
Demir, M., Alalfi, M., Turetken, O., Ferworn, A., 2019. Security smells in smart contracts. In: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion. QRS-C, IEEE, pp. 442–449.
Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., Hierons, R., 2018. Smart contracts vulnerabilities: a call for blockchain software engineering? In: 2018 International Workshop on Blockchain Oriented Software Engineering. IWBOSE, IEEE, pp. 19–25.
Dey, T., Mockus, A., 2020. Deriving a usage-independent software quality metric. Empir. Softw. Eng. 25 (2), 1596–1641.
Dhawan, M., 2017. Analyzing safety of smart contracts. In: Proceedings of the Conference: Network and Distributed System Security Symposium, San Diego, CA, USA, pp. 16–17.
Ducasse, S., Rocha, H., Bragagnolo, S., Denker, M., Francomme, C., 2019. Open-source tool suite for smart contract analysis. Blockchain and Web 3.0: Social, Economic, and Technological Challenges, Routledge.
Fontana, F.A., Mariani, E., Morniroli, A., Sormani, R., Tonello, A., 2011. An experience report on using code smells detection tools. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings, pp. 450–457.
Gencer, A.E., Basu, S., Eyal, I., Van Renesse, R., Sirer, E.G., 2018. Decentralization in bitcoin and ethereum networks. In: International Conference on Financial Cryptography and Data Security. Springer, pp. 439–457.
Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y., 2018. MadMax: surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. 2 (OOPSLA), 116:1–116:27.
Group, T.B., 2019a. Ganache. Online; https://www.trufflesuite.com/ganache. (Accessed 1 November 2019).
Group, T.B., 2019b. Truffle. Online; https://www.trufflesuite.com/. (Accessed 1 November 2019).
Grover, P., Kar, A.K., Janssen, M., 2019. Diffusion of blockchain technology. J. Enterp. Inf. Manag..
Gupta, M., 2018. Solidity gas optimization tips. Online; https://mudit.blog/solidity-gas-optimization-tips/. (Accessed 21 February 2020).
Gupta, M., 2019. Solidity tips and tricks to save gas and reduce bytecode size. Online; https://blog.polymath.network/solidity-tips-and-tricks-to-save-gas-and-reduce-bytecode-size-c44580b218e6. (Accessed 21 February 2020).
Hegedűs, P., 2019. Towards analyzing the complexity landscape of solidity based ethereum smart contracts. Technologies 7 (1), 6.
Holm, S., 1979. A simple sequentially rejective multiple test procedure. Scand. J. Stat. 65–70.
Jelski, K., 2019. Three tips for optimizing gas. Online; http://blockbites.io/bites/bite2.html. (Accessed 21 February 2020).
Kitchenham, B.A., Pfleeger, S.L., 2002a. Principles of survey research part 2: designing a survey. ACM SIGSOFT Softw. Eng. Notes 27 (1), 18–20.
Kitchenham, B.A., Pfleeger, S.L., 2002b. Principles of survey research: part 3: constructing a survey instrument. ACM SIGSOFT Softw. Eng. Notes 27 (2), 20–24.
Kitchenham, B.A., Pfleeger, S.L., 2002c. Principles of survey research part 4: questionnaire evaluation. ACM SIGSOFT Softw. Eng. Notes 27 (3), 20–23.
Kitchenham, B.A., Pfleeger, S.L., 2002d. Principles of survey research: part 5: populations and samples. ACM SIGSOFT Softw. Eng. Notes 27 (5), 17–20.
Kitchenham, B.A., Pfleeger, S.L., 2003. Principles of survey research part 6: data analysis. ACM SIGSOFT Softw. Eng. Notes 28 (2), 24–27.
Liaw, A., Wiener, M., 2002. Classification and regression by randomforest. R News 2 (3), 18–22, URL https://CRAN.R-project.org/doc/Rnews/.
Mäntylä, M., Vanhanen, J., Lassenius, C., 2004. Bad smells - Humans as code critics. In: 20th International Conference on Software Maintenance, ICSM 2004, 11-17 September 2004, Chicago, IL, USA, pp. 399–408.
Marchesi, L., Marchesi, M., Destefanis, G., Barabino, G., Tigano, D., 2020. Design patterns for gas optimization in ethereum. In: 2020 IEEE International Workshop on Blockchain Oriented Software Engineering. IWBOSE, IEEE, pp. 9–15.
Meng, M.H., Qian, Y., 2018. A blockchain aided metric for predictive delivery performance in supply chain management. In: 2018 IEEE International Conference on Service Operations and Logistics, and Informatics. SOLI, IEEE, pp. 285–290.
Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. Online; https://bitcoin.org/bitcoin.pdf. (Accessed 29 February 2020).
Oliva, G.A., Hassan, A.E., Jiang, Z.M.J., 2020. An exploratory study of smart contracts in the ethereum blockchain platform. Empir. Softw. Eng. 25 (3), 1864–1904.
Oppenheim, A.N., 2000. Questionnaire Design, Interviewing and Attitude Measurement. Bloomsbury Publishing.
Ortu, M., Orrú, M., Destefanis, G., 2019. On comparing software quality metrics of traditional vs blockchain-oriented software: An empirical study. In: 2019 IEEE International Workshop on Blockchain Oriented Software Engineering. IWBOSE, IEEE, pp. 32–37.
Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., 2014. Do they really smell bad? A study on developers' perception of bad code smells. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014, pp. 101–110.
Peng, C., Rajan, A., 2019. Sif: A framework for solidity code instrumentation and analysis. arXiv preprint arXiv:1905.01659.
Pfleeger, S.L., Kitchenham, B.A., 2001. Principles of survey research: part 1: turning lemons into lemonade. ACM SIGSOFT Softw. Eng. Notes 26 (6), 16–18.
Pierro, G.A., Tonelli, R., Marchesi, M., 2020. An organized repository of ethereum smart contracts' source codes and metrics. Future Internet 12 (11), 197.
Sabatti, C., Lalanne, C., 2009. Applied statistical genetics with R for population-based association studies. J. Stat. Softw. 31 (1), 1–5.
Salkind, N.J., 2010. Encyclopedia of Research Design, Vol. 1. sage.
Sammut, C., Webb, G.I., 2011. Encyclopedia of Machine Learning. Springer Science & Business Media.
Škvorc, B., 2018. Audits beyond code: optimizing gas. Online; https://audithor.io/audits-beyond-code-optimizing-gas/. (Accessed 21 February 2020).
Strobl, C., Boulesteix, A., Kneib, T., Augustin, T., Zeileis, A., 2008. Conditional variable importance for random forests. BMC Bioinform. 9.
Szego, D., 2018. Solidity gas optimization - memory arrays. Online; http://danielszego.blogspot.com/2018/01/solidity-gas-optimization-memory-arrays.html. (Accessed 21 February 2020).
Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y., 2018. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, pp. 9–16.
Tonelli, R., Destefanis, G., Marchesi, M., Ortu, M., 2018. Smart contracts software metrics: a first study. arXiv preprint arXiv:1802.01517.
Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016. An empirical investigation into the nature of test smells. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp. 4–15.
Vacca, A., Di Sorbo, A., Visaggio, C.A., Canfora, G., 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. J. Syst. Softw. 174, 110891.
Wohrer, M., Zdun, U., 2018. Smart contracts: security patterns in the ethereum ecosystem and solidity. In: 2018 International Workshop on Blockchain Oriented Software Engineering, IWBOSE@SANER 2018, Campobasso, Italy, March 20, 2018, pp. 2–8.
Ye, J., Ma, M., Peng, T., Peng, Y., Xue, Y., 2019. Towards automated generation of bug benchmark for smart contracts. In: 2019 IEEE International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 184–187.
Zhang, P., Walker, M.A., White, J., Schmidt, D.C., Lenz, G., 2017. Metrics for assessing blockchain-based healthcare decentralized apps. In: 2017 IEEE 19th International Conference on E-Health Networking, Applications and Services. Healthcom, IEEE, pp. 1–4.
Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., Bailey, M., 2018. Erays: Reverse engineering ethereum's opaque smart contracts. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, pp. 1371–1385.
Zou, W., Lo, D., Kochhar, P.S., Le, X.-B.D., Xia, X., Feng, Y., Chen, Z., Xu, B., 2019. Smart contract development: Challenges and opportunities. IEEE Trans. Softw. Eng..

Andrea Di Sorbo is a research fellow at the University of Sannio, Italy. He received the Ph.D. degree in information technology from the University of Sannio, in 2018. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, text analysis and software security and privacy. He coauthored several papers appeared in flagship international conferences (ICSE, FSE, ASE) and journals (TSE, JSS, IST, JSEP). He serves and has served as review editor and guest associate editor for Frontiers in Big Data, guest editor

for the Information and Software Technology journal, and reviewer for several journals in the field of software engineering, such as Transactions on Software Engineering, edited by IEEE, Transactions on Software Engineering and Methodology, edited by ACM, and the Empirical Software Engineering journal edited by Springer. He is also a program committee member of some international conferences (ARES, MOBILESoft, SEAA).

Sonia Laudanna is a Ph.D. student in Software Engineering at the Department of Engineering of the University of Sannio, Italy. Since 2018, her research activity focuses on machine learning, deep learning, blockchain and CyberSecurity.



Anna Vacca is a Ph.D. student in Software Engineering at the Department of Engineering of the University of Sannio, Italy. She obtained B.Sc. in Computer Engineering from the University of Sannio, Italy, in 2014, while, in 2017, she received M.Sc. in Computer Engineering from the same university. Her research interests include blockchain applications, software maintenance, and software security.

Corrado Aaron Visaggio is an associate professor of CyberSecurity at the Department of Engineering of University of Sannio. He is chair of the node of University of Sannio for the CINI National Cyber Security Lab. He is the scientific coordinator of several projects funded by firms operating in CyberSecurity, concerning malware analysis, vulnerability assessment, and data protection. He is also among the founders of the academic spin-off SER&Practice. He serves in the Editorial Board of the International Journal of Computer Virology and Hacking techniques (Springer), as associate editor in Frontiers in Big Data, and in several Program Committees (MALWARE, ARES, SECRYPT, SEKE, ITASEC, FoRSE, DATA, Hufo, MobiSys, WETSOM, ISSRE); he was also the workshop chair of WETSOM and WMA. His main research interests are: malware analysis, data privacy and protection, software security, empirical software engineering.

Gerardo Canfora is a professor of computer science at the School of Engineering of the University of Sannio, Italy. He serves on the program and organizing committees of a number of international conferences. He was general chair of WCRE06 and CSMR03, and program co-chair of ICSE15, WETSoM12 and 10, ICSM01 and 07, IWPSE05, CSMR04 and IWPC97. He was co-editor of the Journal of Software: Evolution and Process. Canfora authored 200 research papers; his research interests include software maintenance and evolution, security and privacy, empirical software engineering, and service-oriented computing.