International Journal of Grid and Distributed Computing
Vol. 10, No. 12 (2017), pp.53-64
http://dx.doi.org/10.14257/ijgdc.2017.10.12.06

Saving Deployment Costs of Smart Contracts by Eliminating Gas-wasteful Patterns

JaeYong Park, Daegeon Lee and Hoh Peter In*

Department of Computer Science & Engineering, Korea University, Korea
*loput12ouj35@korea.ac.kr, eorjs3885@gmail.com, *hoh_in@korea.ac.kr*

Abstract

Smart contracts are blockchain-based programs that have developed with the emergence of Ethereum, one of the most well-known blockchains. Gas, paid in Ethers (i.e., the cryptocurrency in Ethereum), is required for the costs to upload and run smart contracts on Ethereum. As cost-inefficiently designed smart contracts result in unnecessary costs, it is vital to eliminate any gas-wasteful code fragments to optimize the deployment costs. In this study, we define five gas-wasteful patterns: 'Over-public variables', 'Redundant initial values', 'Loose packing', 'Non-base unit types', and 'Non-constant variables', based on the state variables in Solidity, the most commonly used implementation language for smart contracts in Ethereum. We also propose improvement methods related to these patterns and a solution to identify and eliminate the patterns. Furthermore, we analyze 143 real-world contracts deployed on Ethereum and find that 56% of them include the above-mentioned patterns. We also upgrade 43 of the pattern-matched contracts and demonstrate that their deployment costs are decreased on an average by 13.47%, and the most-reduced rate is 64%.

Keywords*: Smart contract, Ethereum, Solidity, Gas, Cost, Pattern*

1. Introduction

Since the advent of Bitcoin [1] in 2008, blockchains that are decentralized ledgers containing simple transaction records, have steadily developed, and consequently, smart contracts (*i.e.,* programs run on blockchains) have emerged with the launch of Ethereum in 2014 [2]. Ethereum is one of the most well-known blockchains with more than 50 million transactions and approximately 94 million Ether, the cryptocurrency of Ethereum [3]. In Ethereum, smart contracts are implemented in three high-level languages: Solidity, Serpent, and LLL (Lisp-Like Language). Regardless of the language a smart contract is coded in, it will be compiled into bytecode, the executable machine code in the Ethereum Virtual Machine (EVM), and transferred to the Ethereum blockchains.

'Gas' is required for meeting the costs to deploy smart contracts and execute functions [4]. Ether-based gas is the payment for the computing resources required for the computational steps in smart contracts, and miners who build the blocks and offer their computing resources receive the payment. Gas requirement depends on the EVM opcode used in a smart contract [4]. As gas is paid in Ether that is paid in real-world money, saving gas means saving real-world money. In this paper, we address the issue of the gas requirement (*i.e.,* cost) for uploading a new smart contract, and not for executing existing contracts. An uploading cost consists of transaction and execution costs [4]. The former is for sending the bytecode to the blockchains, and the latter is for the necessary computation processes, namely, initializing the state variables (*i.e.,* the variables whose values are permanently stored in the contract storage) and functions and executing a

Received (September 5, 2017), Review Result (December 2, 2017), Accepted (December 7, 2017)
* Corresponding Author

ISSN: 2005-4262 IJGDC

constructor. Inclusion of numerous variables and functions in a smart contract implies a long bytecode and production of the computation processes with increased difficulty, leading to an increment in the transaction and execution costs. Hence, poorly designed smart contracts can generate unessential gas demand, and thus, it is very important to remove unnecessary and gas-wasteful parts present in the codes before uploading them. In addition, the price of Ether sharply increased from approximately 10 US dollar in January 2017 to approximately 380 US dollar in June 2017 [3]. The price of gas is dynamic and determined by the miners. It reached approximately 3×10^{-8} Ether, and could be exchanged to 0.0000115486336142 US dollars in June 2017 [3]. According to this price, approximately 1.15 US dollars are consumed to purchase 100,000 units of gas.

In this research, we identify five gas-wasteful patterns in the declarations of the state variables in contracts written in Solidity, the most commonly used language for implementing smart contracts in Ethereum, and propose methods to remove such patterns to save deployment costs of smart contracts. Moreover, we analyze and upgrade existing smart contracts on Ethereum using the methods.

2. Background

Gas is paid in exchange for the computing resources serviced by the miners to execute the operations. According to the Ethereum yellow paper [4], the ADD operation, which simply adds two values, spends three units of gas or approximately 0.00004 US dollars. This is a very small value for a single operation, but as the smart contracts become larger and more complex, they require more gas. Furthermore, most operations demand more gas than the ADD operation [4]. Table 1 lists the frequently used operations in smart contracts, and their gas consumption.

Table 1. Operations in Smart Contracts and their Gas Consumption [4]

Operation	Gas	Operation	Gas
ADD, SUB	3	LT, GT, SLT, SGT, EQ, ISZERO	3
MUL, DIV, SDIV	5	JUMP	8
MODE, SMOD	5	JUMPi	10
ADDMOD, MULMOD	8	MLOAD, MSTORE, MSTORE8	3
AND, OR, XOR, NOT, BYTE	3	SLOAD	200
POP	2	STORAGEMODE	5000
DUP, SWAP, PUSH	3	STORAGEADD	20000

Arithmetic operations, stack-related operations, bit operations, and comparison operations require a very small amount of gas as they are processed in the stack-based EVM [4]. Memory-related operations (*e.g.*, MLOAD) that access the EVM memory are also inexpensive, but burn 3 units of gas per word (256 bits), and thus, large memory data can drain the gas. Storage-related operations (*e.g.*, SLOAD) are considerably expensive as they access the fixed memory area of a smart contract where its state variables are allocated. Therefore, it can be wasteful to store large data into the storage instead of the memory. To lower the gas demand, unnecessary operations, particularly storage-related ones, must be removed.

3. Related Work

There is only one research in regard to the gas in Ethereum. Ting Chen *et al.* [7] defined seven gas-costly patterns and classified them into two categories: useless code-related patterns and loop-related patterns. Useless code-related patterns

contain two patterns: dead code (*i.e.*, a code is never executed in any case) and opaque predicates (*i.e.*, predicates whose results are obviously true or false without requiring execution). Loop-related patterns include five patterns: expensive operations in a loop, constant outcome of a loop, loop fusion (*i.e.*, unmerged loops), repeated computations in a loop, and comparison with a unilateral outcome in a loop. According to this study, unnecessary gas can be wasted in gas-costly designed smart contracts owing to the useless code- and loop-related patterns. Using different approaches in our study, we address the gas requirement to upload smart contracts by focusing on the state variables.

The concept of gas in Ethereum can be considered to correspond to the power or memory costs in embedded processors because contracts should utilize limited power or memory resources. Leupers [5] introduced code optimization techniques such as common subexpression elimination and constant folding. According to this work, one of the optimization goals in embedded processors is to avoid expensive memory accesses. Similarly, storage-related operations in Solidity are corresponding to the accesses and high-priced. Thus, we focus on state variables to avoid these expensive operations.

4. Gas-Wasteful Patterns and Improvement Methods

We define five inefficient patterns in terms of gas consumption based on the declarations and initialization of the state variables as they are associated with storage-related operations. The compiler of Solidity version 0.4.15 with the enabled optimization option is used for determining the gas requirement. Note that we focus on state variables and not on local variables (*i.e.*, variables used in functions and not stored in the contract storage). The five gas-wasteful patterns are as follows:

- Pattern 1: Over-public variables
- Pattern 2: Redundant initial values
- Pattern 3: Loose packing
- Pattern 4: Non-base unit types
- Pattern 5: Non-constant variables

These patterns are disadvantageous for gas consumption because they involve expensive operations, particularly storage-related ones.

4.1. Over-Public Variables

This pattern exists when a public variable is not required to be public. An extreme example is a contract in which all the variables are public. In Solidity, a variable can have three types of visibility: internal, public, or private, and the compiler automatically generates the getter functions for all the public variables [8]. Over-public variables engender inessential getter functions that expand the size of the bytecode and thereby increase the deployment cost of a contract.

To improve the contracts that contain this pattern, the over-public variables should be modified to non-public. More precisely, the getter functions of the variables containing temporary or meaningless values are not necessary. The same is the case for those of static constants (*i.e.*, constant variables whose values are known before compilation of the source code, such as simple integer or address values) as their values can be derived from the source code. Therefore, these variables should be internal or private instead of public. Moreover, despite the getters generated by the compiler, generating a new getter manually also unnecessarily increases the placement cost. Thus, all the duplicate getters of the public variables should be eliminated. However, non-constant variables associated with Ether transfer or required to be referenced by other contracts must be public. The same is the case for the dynamic constants (*i.e.*, constant variables whose values

are set at deployment, such as timestamp or message sender) as their values are unpredictable. These variables are the exceptions of the pattern.

Table 2 presents an example of this pattern in which all the variables are public. It also provides the corresponding improved version. In this example, Public tags for temporary variables can be deleted to economize gas. Fewer getters are produced in the improved smart contract; therefore, it is noticeably less expensive and saves approximately 36.85% of its deployment cost.

Table 2. Contract with over-Public Variables and its Improved Version

	Over-public	Improved
Source code	`contract A { uint constant public limit = 10 ether; uint public referenced; uint public referenced2; int public tmp; bool public lock; }`	`contract B { uint constant limit = 10 ether; uint public referenced; uint public referenced2; int tmp; bool lock; }`
Deployment cost (gas)	227,070	143,386

4.2. Redundant Initial Values

In this pattern, an initial value that is identical to a default value (*e.g.*, 0, false, and "" for an integer, a Boolean, and a string, respectively) is assigned to a variable, triggering an extra and unnecessary initialization process. Assigning the default values to variables for initialization can enhance the legibility of codes, but it is not beneficial for the gas consumption. In detail, the compiler initializes all the variables using two steps: allocating the default values depending on the types, and then assigning their own values expressed in the declaration or constructor. Those with no given values skip the second step and take the default values. Therefore, in terms of semantics, there is no difference between a variable with no assigned value and one for which the default is the assigned value. However, the latter does not bypass the unessential assignment (*e.g.*, change from 0 to 0) in the second step and requires more operations (*e.g.*, SSTORE) and gas.

Table 3. Contracts with Redundant Initial Values and their Improved Version with No Initial Value

	Redundant initial values		Improved
	Provided in declaration	Provided in constructor	
Source code	`contract A { int a = 0; }`	`contract B { int a; function B { a = 0; } }`	`contract C { int a; }`
Deployment cost (gas)	90,072	90,212	79,852

Thus, to improve the contracts matching this pattern, initial values equal to the default should be discarded to omit the needless assignment. For instance, 'int i = 0', 'bool b = false', and 'string s' in the declaration should be 'int i', 'bool b', and 'string s', respectively. As an exception, all the constant variables must be assigned the initial values even if they are the default values according to the syntax of Solidity. As an example, Table 3 compares contracts containing redundant initial values and their improved version. The first and second contracts assign as initial values the default of integer for their variables in the declaration and constructor,

respectively. Retaining the same value in the variables, these contracts can be upgraded by removing the given values. Despite no change in the semantic, the gas expenditure decreases by approximately 11.35%.

4.3. Loose Packing

In this pattern, the state variables are unsorted, activating loose packing (*i.e.*, the variables are not packed tightly). Grouping and arranging storage variables (*i.e.*, state variables) by topics can be conducive to legibility, but it is inefficient with respect to the gas requirement. With the compiler bundling multiple storage items into a single 256-bit storage slot as densely as possible in sequence and combining multiple readings or writings into a single operation, the order of the items influences the length of the total slots and operations [8]. For example, it is recommended to declare storage variables in the order of 'uint128, uint128, uint256' instead of 'uint128, uint256, uint128' as the former occupies two slots, whereas the latter that is not packed tightly, occupies three slots and produces more operations [8]. Note that tight packing is only efficacious for storage values (*i.e.*, values of storage variables or their elements) and not for memory values (*i.e.*, values of function arguments or local variables). Thus, the compiler can bind unordered storage variables loosely and merge less reading and writing operations compared to ordered ones, which consequently increases the uploading cost.

To improve the contracts containing this pattern, the storage variables and their values in the collections (*i.e.*, arrays or structs) should be arranged to fit in 256-bit slots for tight packing by grouping small-sized ones together. In addition, prefixes and suffixes indicating topics can be utilized for legibility instead of grouping and sorting by the topics [6].

Table 4. Contract with Unsorted Variables and Its Improved Version with Sorted Variables

	Unsorted (loose packing)	Improved (tight packing)

Source code	`contract A {`
	`uint128 a = 1;`
	`uint256 b = 1;`
	`uint128 c = 1;`
	`}`
		`contract B {`
		`uint128 a = 1;`
		`uint128 c = 1;`
		`uint256 b = 1;`
		`}`

Deployment cost (gas)	203,448	163,072

Table 4 presents an example of this pattern and its improves version in which the deployment cost is drastically decreased by approximately 19.85% as fewer slots are occupied. In addition, it can be considered for a solution to replace unit types with smaller ones as values can be compressed more tightly. However, it should be carefully applied because it is not always effectual owing to the next pattern.

4.4. Non-base Unit Types

In this pattern, the types of integer or byte smaller than the base unit (*i.e.*, one word or 256 bits) are used immoderately. The default integer types in Solidity (*i.e.*, int and uint) occupy 256 bits [8], which is a remarkably larger occupancy compared with other programming languages. For instance, the integer types of C and JAVA occupy only 32 bits. However, Solidity supports short-unit types for an integer and a byte from 8 bits to 256 bits in steps of 8 bits (*e.g.*, int8, int16, and int24) for saving memory. Nonetheless, in regard to the gas requirement, these types are not always efficient and can increase the gas consumption as the EVM performs on one base unit at a time. Data types under 256 bits necessitate additional operations to

downscale the elements from 256 bits to their desired size that lengthens the bytecode and increases the uploading and executing costs. For storage values, some of these operations can be integrated through tight packing, but the memory values related to the storage values still require resizing operations. Hence, it is desirable to only utilize compact-unit types for tight packing, as mentioned for the previous pattern, when the gain from tight packing is larger than the disadvantage from the extra downscaling operations.

To improve the contracts containing this pattern, for all the integer and byte variables, the operations or estimated costs of the minimized-size and base-size should be compared, and each variable should adopt the more beneficial size. Table 5 presents a simple example with an 8-bit type integer and its improvement by replacing the type of the variable with the corresponding base-unit. The first contract is slightly more expensive than the second one owing to the bit-scaling operations.

Table 5. Contract with a Non-Base Unit Integer Variable and its Improved Version with a Base Unit Integer

	Non-based unit (8-bit unit)	Improved (256-bit unit)

Source code	`contract A {`
	`uint8 public count;`
	`}`
		`contract B {`
		`uint public count;`
		`}`

Deployment cost (gas)	123,298	119,958

4.5. Non-Constant Variables

In this pattern, a variable whose value is never modified after its first allocation is not declared as a constant. Non-constant variables are intertwined with storage operations such as SSTORE and SLOAD for saving and loading, whereas constant variables do not require these operations because they are not stored and substituted by their own values at the compile-time as if they are not present in the codes. Not only the expensive storage operations but also the storage space can be wasted, increasing the placement and executing costs substantially. The unnecessary operations generated by this pattern require more exorbitant costs compared with any other patterns, and hence, this pattern is the most gas-wasteful.

Table 6. Contract with A Non-Constant Integer Variable and Its Improved Version with A Constant Integer

	Non-constant	Improved

Source code	`contract A {`
	`uint public a = 1;`
	`}`
		`contract B {`
		`uint public constant a = 1;`
		`}`

Deployment cost (gas)	160,310	119,618

To improve the contracts that match this pattern, all the invariant variables should be declared as constants. Table 6 gives a simple comparison between an ordinary variable and a constant in terms of the gas consumption. The improved contract is significantly more economical because of the comparatively less storage operations and spaces. Although it has only one variable, 43,692 units of gas (*i.e.*, approximately 27.25% of its deployment cost) are conserved.

5. Pattern Detection and Elimination in Contracts

For the detection and clearance of the five gas-wasteful patterns to reduce the deployment costs, all the state variables in a contract should be verified and modified so as not to match with the patterns. The most easily detected pattern is Pattern 5, followed by Patterns 2, 1, 3, and 4, and each variable can be checked whether it has a connection with the patterns in this sequence. In addition, the most uneconomical pattern is Pattern 5 as the consequent redundant operations required are drastically expensive.

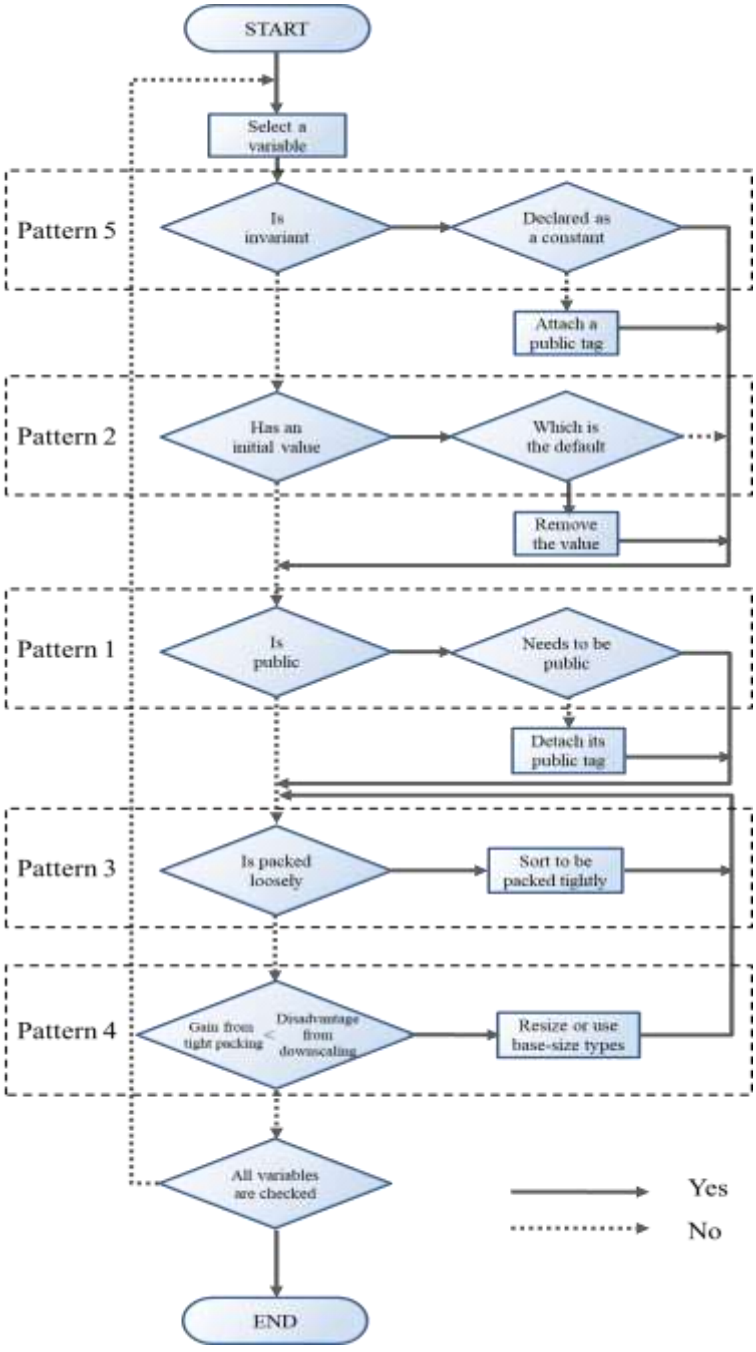

Figure 1. Flow Chart of the Proposed Solution Based On the Five Gas-Wasteful Patterns

We propose a solution for the purpose of detecting the patterns and improving the contracts suffering from them based on these ideas and the improvement methods. The solution is the following:

1. Select an unverified state variable.
2. If it is invariant or immutable but not declared as a constant, attach a constant tag and skip the next step because constants are the exception of Pattern 2 owing to the syntax of Solidity.
3. If it has an initial value identical to the default, remove the value.
4. If it is public but not required to be public, detach its public tag.
5. If it or its items are packed loosely, modify their order until they fit into the base unit and are tightly packed.
6. If its disadvantage from downscaling because of size-reduction is more significant than its gain from the tight packing, resize it or consider using the base-size types, and return to the previous step.
7. The variable has been verified. Return to the step 1 and choose the next variable until all the variables are examined.

Figure 1 illustrates the flow chart of the solution with marks of the five patterns around the relevant steps. Note that a variable can match more than one pattern. Legibility is fundamental to code maintenance [6]; therefore, we recommend saving a copy (*i.e.*, developer version) because steps 5 and 6 can hinder the legibility.

6. Evaluation

We had downloaded 8948 real-world Solidity contracts uploaded on Ethereum from Etherscan [3], the Ethereum block explorer, on June 23, 2017, in the order of the highest Ether and excluded identical, overlapping, or similar contracts. We also revised old-version contracts uncompiled by the compiler of the current version, and finally, 143 contracts were analyzed.

6.1. Analysis of Real-World Contracts

We analyze the Solidity codes of the contracts to find the five gas-wasteful patterns and discover the number of patterns that match to the contracts. As Figure 2 displays, the patterns are not detected in 63 of the 143 contracts (*i.e.*, approximately 44%), whereas 80 contracts (*i.e.*, the rest of them) are cost-inefficiently programmed including at least one of the patterns. Furthermore, approximately 46% of the pattern-matched contracts suffer from multiple patterns, implying that numerous contracts are not appropriately designed with respect to the efficiency of the deployment costs and are required to be improved.

Figure 3 shows the percentages of the pattern-matched contracts in which the respective patterns are detected. Pattern 2, redundant initial values pattern, is detected the most at 65%, and Pattern 5, non-constant pattern, follows at 52.5%, indicating that programmers habitually assign the default values and do not attach constant tags for the unchanging variables while not recognizing the fact that these are disadvantageous to the uploading costs of their contracts. In addition, 32.5% of the contracts include both Patterns 5 and 2. The proportion of Pattern 1, over-public variable pattern, is 22.5%, suggesting that public tags are overused in some of the contracts. Moreover, 16.25% of the contracts involve both Patterns 5 and 2. The percentages of Pattern 3 (*i.e.*, loose packing) and Pattern 4 (*i.e.*, non-base unit types) are 17.5% and 12.5%, respectively. These are relatively small as compact-sized types are rarely used in the contracts.

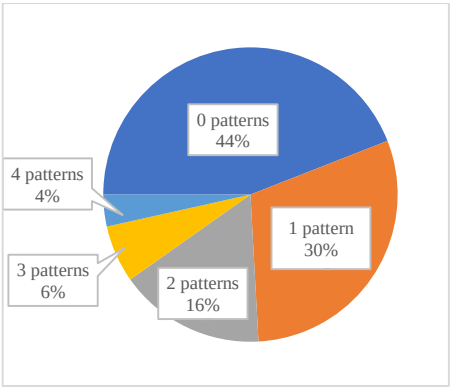

Figure 2. Ratio of the Contracts Matching With N-Patterns

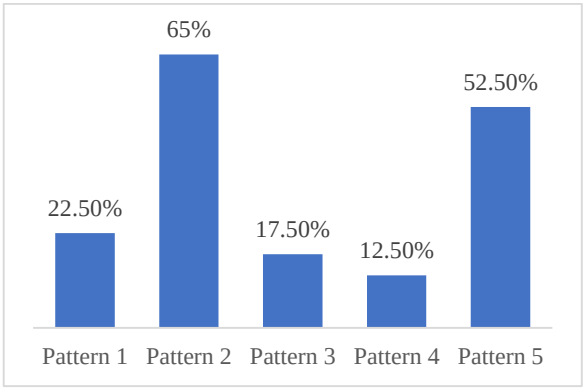

Figure 3. Rate of the Contracts Matching With the Respective Patterns

6.2. Reduced Costs of Improved Contracts

We select and upgrade 43 random cases from the 80 pattern-detected contracts using our solution and evaluate it by analyzing how much gas consumption the respective cases cut for decreasing the deployment costs. Note that the compiler of Solidity version 0.4.15 with enabled optimization option is used for determining the gas consumption.

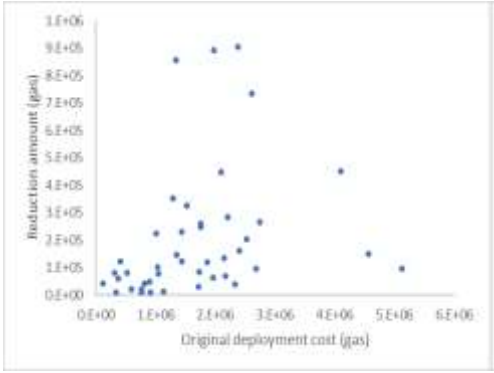

Figure 4. Reduction Amount as a Function of the Deployment Cost

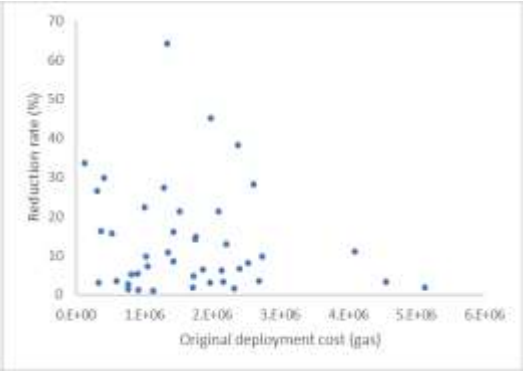

Figure 5. Reduction Rate as a Function of the Deployment Cost

Figures 4 and 5 illustrate the reduction amount and rate as a function of the deployment cost for the cases after applying the solution, respectively. The rates are derived from the difference between the original and improved costs divided by the

original cost. The highest value of saved cost is 906,698 units of gas, followed by 893,944 and 858,246 units of gas, and these three amounts are equivalent to approximately 10.47, 10.32, and 9.91 US dollars, respectively. Most of the high-ranked cases contain invariant variables that are not declared as constants and match with Pattern 5. This supports the fact that this pattern is the most gas-wasteful and the reason the first step of the solution should start with Pattern 5. In addition, the high-ranked cases include numerous pattern-matched variables, whereas the low-ranked cases are simple or comparatively well-designed with only one or two minor problems.

The more complex a contract becomes, the more possibilities for cost-cutting exist. However, the reduction rate of its deployment cost can decrease because its original cost can increase faster than the improved cost. Thus, a large amount of gas can be cut in large contracts with low reduction rates. The maximum and minimum of the reduction rates are approximately 64% and 1%, respectively, and the average is approximately 13.47%. Approximately 67% of the cases have reduction rates over 5%, and most of the remaining cases are not related to Pattern 5. Nevertheless, the other patterns are not negligible in multiple-patterns-matched contracts or contracts in which almost all the variables are associated with them. For example, one case in which the variables are almost perfectly matched with Patterns 2 and 3 is found to be at approximately 21.39%. Although eliminating Patterns 3 or 4 also contributes to reduction of the uploading costs, owing to high difficulty in detection and improvement, the solution can be simplified for small-sized contract designers by skipping the procedure related to the patterns, but focusing on others.

7. Conclusion

We identify five gas-wasteful patterns: ‘Over-public variables’, ‘Redundant initial values’, ‘Loose packing’, ‘Non-base unit types’ and ‘Non-constant variables’, and propose the improvement methods associated with the patterns and a solution to identify and remove the patterns. Furthermore, we analyzed 143 contracts existing on Ethereum, and determine that 56% of them suffer from these patterns. Patterns 1, 2, 3, 4, and 5 match with 22.5%, 65%, 17.5%, 12.5%, and 52.5% of the pattern-detected contracts, respectively. We also improve 43 contracts and demonstrate that the reduction rate of the deployment costs is 13.47% on an average, ranging from 1% to 64%. In addition, the most-saved cost among the contracts is 906,698 units of gas.

Smart contract designers and programmers are expected to reduce the deployment costs of smart contracts by removing the five gas-wasteful patterns. This work can also contribute to future research related to cost optimization or design patterns in blockchain-based programs.

Acknowledgements

This research was supported by the MIST(Ministry of Science and ICT), Korea, under the National Program for Excellence in SW(2015-0-00936) supervised by the IITP(Institute for Information & communications Technology Promotion)

References

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system”, **(2008)**.
[2] V. Buterin, “A next-generation smart contract and decentralized application platform”, **(2004)**.
[3] Etherscan, “The Ethereum block explorer”, https://etherscan.io, accessed Jun., **(2017)**.
[4] G. Wood, “Ethereum: A secure decentralized generalized transaction ledger”, **(2014)**.

[5] [5] R. Leupers, "Machine-independent optimizations in Code optimization techniques for embedded processors: Methods, algorithms, and tools", Springer Science & Business Media, **(2013)**, pp.16-24.
[6] J. R. de Almeida, J. B. Camargo, B. A. Basseto and S. M. Paz, "Best practices in code inspection for safety-critical software", IEEE Software, vol. 20, no. 3, **(2017)**, pp. 56-63.
[7] T. Chen, X. Li, X. Luo and X. Zhang, "Under-optimized smart contracts devour your money", Software Analysis, Evolution and Reengineering (SANER), IEEE 24th International Conference on. IEEE, **(2017)**.
[8] Ethereum, "Solidity document v0.4.15", http://solidity.readthedocs.io, accessed Jun., **(2017)**.

Authors

JaeYong Park, he is a Master Course student in the Department of Computer Science and Engineering at Korea University in Seoul, Korea. His major areas of study are static analysis and testing on Android, and smart contracts on Blockchain. He received the Bachelor's degree in Computer Science and Engineering from Korea University in 2017.

Daegeon Lee, he is a Master Course student in the Department of Computer Science and Engineering at Korea University in Seoul, Korea. His major areas of study are smart contract on Blockchain and Blockchain's storage saving. He received the Bachelor's degree in Computer Science and Engineering from Korea University in 2017.

Hoh Peter In, he received his Ph.D. degree in Computer Science from the University Of Southern California (USC). He was an Assistant Professor at Texas A&M University. At present, he is a professor in Department of Computer Science and Engineering at Korea University in Seoul, Korea. He is an editor of the IJSI, TIIS and BPIM journal. His primary research interests are software engineering, social media platform and services, and software security management. He earned the most influential paper award for 10 years in ICRE 2006. He has published over 100 research papers

