Towards Saving Money in Using Smart Contracts

Ting Chen[†‡], Zihao Li[†], Hao Zhou[‡], Jiachi Chen[‡], Xiapu Luo[‡§], Xiaoqi Li[‡] , Xiaosong Zhang[†]
†Center for Cybersecurity, University of Electronic Science and Technology of China, China
‡Department of Computing, The Hong Kong Polytechnic University, China
{brokendragon,johnsonxsz}@uestc.edu.cn,csxluo@comp.polyu.edu.hk

ABSTRACT

Being a new kind of software leveraging blockchain to execute real contracts, smart contracts are in great demand due to many advantages. Ethereum is the largest blockchain platform that supports smart contracts by running them in its virtual machine. To ensure that a smart contract will terminate eventually and prevent abuse of resources, Ethereum charges the developers for deploying smart contracts and the users for executing smart contracts. Although our previous work shows that under-optimized smart contracts may cost more money than necessary, it just lists 7 anti-patterns and the detection method for 3 of them. In this paper, we conduct the *first* in-depth investigation on such under-optimized smart contracts. We first identify 24 anti-patterns from the execution traces of real smart contracts. Then, we design and develop `GasReducer`, the *first* tool to automatically detect all these anti-patterns from the bytecode of smart contracts and replace them with efficient code through bytecode-to-bytecode optimization. Using `GasReducer` to analyze all smart contracts and their execution traces, we detect 9,490,768 and 557,565,754 anti-pattern instances in deploying and invoking smart contracts, respectively.

KEYWORDS

Smart contract, Anti-patterns, Detection, Optimization

ACM Reference Format:
Ting Chen[†‡], Zihao Li[†], Hao Zhou[‡], Jiachi Chen[‡], Xiapu Luo[‡§], Xiaoqi Li[‡] , Xiaosong Zhang[†]. 2018. Towards Saving Money in Using Smart Contracts. In *ICSE-NIER'18: 40th International Conference on Software Engineering: New Ideas and Emerging Results Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, 4 pages. https://doi.org/10.1145/3183399.3183420

1 INTRODUCTION

Leveraging blockchain techniques, a *smart contract* is an autonomous computer program, which, once started, can execute automatically and mandatorily according to the program logic defined beforehand [9]. Due to many unique advantages (e.g., automatic execution in deterministic manner, without trusted intermediaries, highly resistant to forgery, etc.), smart contracts have the power to reshape a number of industries, in which retail banking, insurance, financial exchange, and content platforms are the most impacted [7].

ICSE-NIER'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183420

Ethereum is the largest blockchain supporting smart contracts, and has already 599,959 deployed smart contracts as of June, 2017 (https://etherscan.io/). Smart contracts are typically developed in a high-level language (e.g., Solidity) and then compiled into the bytecode form, dubbed EVM bytecode [9]. Since every node of Ethereum maintains the same and complete copy of the blockchain, after a smart contract is deployed, all nodes will have a copy of it in their disk space. Moreover, when a user invokes a smart contract by sending a transaction, all nodes will execute it in their Ethereum Virtual Machines (EVM) and communicate the execution result for achieving consensus, and thus the computing resources on all nodes will be consumed. Therefore, to ensure that a smart contract will terminate eventually and prevent abuse of resources, Ethereum introduces the *gas* mechanism that charges the developers for deploying smart contracts and the users of smart contracts for the execution of every EVM operation. The execution fee is computed by $gas\_price \times gas\_cost$ [9]. *gas_price* is the monetary value of one unit of gas, which is determined by the market. *gas_cost* is the amount of gas consumed by all executed operations, and the gas required by individual EVM operation is defined in [9]. Besides, the developers of smart contracts will also be charged[5] since the deployment of smart contracts consumes resources (e.g., disk), and the gas needed for deployment is proportional to the size of smart contracts in bytecode.

Our previous work shows that under-optimized smart contracts may cost more money than necessary [4]. However, it just lists 7 anti-patterns and the detection method for 3 of them. In this paper, we conduct the *first* in-depth investigation on such smart contracts. We first identify 24 anti-patterns from the execution traces of real smart contracts (Section 2). An anti-pattern denotes an EVM operation sequence that can be replaced with another one that has the same semantics but needs less gas. It is non-trivial to identify such anti-patterns and the corresponding efficient code because of the huge number of possible EVM operation sequences. Then, we design and develop `GasReducer` (Section 3), the *first* tool to automatically detect all these anti-patterns from the bytecode of smart contracts and replace them with efficient code through bytecode-to-bytecode optimization. To keep the optimized smart contract valid, we also address the challenge of recomputing the targets of control flow transfers.

Using `GasReducer` to analyze *all* deployed smart contracts (i.e., 599,959) as of 10 June, 2017, we detect 9,490,768 instances of anti-patterns, which waste 2,040,892,224 units of gas, for deploying smart contracts. Furthermore, by reproducing and inspecting all execution traces (i.e., 9,250,400) of these deployed smart contracts, we discover 557,565,754 instances of anti-patterns, which waste 7,185,048,532 units of gas, for invoking smart contracts. We also examine whether existing compiler optimizations can eliminate these anti-patterns, and find that 506 out of the 515 open-source smart contracts after

compiler optimization still contain 15,205 instances of anti-patterns. This research sheds light on the development and optimization of smart contracts.

The major contributions of this work are as follows.

(1) We identify 24 anti-patterns, which waste gas in deploying and invoking smart contracts, and determine the corresponding efficient code for replacing the anti-pattern instances.

(2) We design and develop `GasReducer`, the *first* tool for automatically detecting these anti-patterns from the bytecode of smart contracts and replacing them with efficient code. `GasReducer` will be released after paper publication.

(3) We conduct experiments on all deployed smart contracts and their execution traces, and detect lots of anti-pattern instances for deploying and invoking smart contracts.

The remainder of this paper is organized as follows. Section 2 introduces anti-patterns. Section 3 details the design of `GasReducer`. Section 4 presents experimental results. Section 5 reviews related work. Section 6 concludes with future plan.

```
swap1| 276,363          OS(len = 1): swap1; push1 0x01; add; swap1; swap1
push1 0x01 | 276,360   OS(len = 2): swap1, push1 0x01; push1 0x01, add; add, swap1; swap1, swap1
add | 276,357           OS(len = 3): swap1, push1 0x01, add; push1 0x01, add, swap1; add, swap1, swap1
swap1 | 276,364        OS(len = 4): swap1, push1 0x01, add, swap1; push1 0x01, add, swap1, swap1
swap1 | 276,351        OS(len = 5): swap1, push1 0x01, add, swap1, swap1
```

Figure 1: Identify *P1* from an execution trace.

2 ANTI-PATTERNS

2.1 Definition

The state of EVM is defined as a tuple $\sigma = \{g, pc, m, s, st\}$ for denoting the remaining gas, program counter, memory, runtime stack (EVM is a stack-based VM), and storage. The *memory* is associating with each transaction that provides parameters to a smart contract and stores return values of the invoked functions [9]. The *storage* is a permanent space, like a database, to store information (e.g., the balance of an account, bytecode of smart contracts). The execution of an operation OP can be defined as applying a function ξ to an EVM state σ and transferring it to another state σ'. $\sigma' = \xi(\text{OP}, \sigma)$, where $g' = \xi(\text{OP}, g)$, ..., $st' = \xi(\text{OP}, st)$.

In a smart contract, an operation sequence OS_1 is an *anti-pattern* instance if there exists another operation sequence OS_2 that is semantically equivalent with OS_1 but consumes less gas. Given the current state σ, if the state (except the field g) after executing OS_1 is the same as that after executing OS_2, OS_1 and OS_2 are semantically equivalent. Let gd_1, ge_1, gd_2 and ge_2 denote the amount of gas required for deploying and executing OS_1 and OS_2, respectively. Then, OS_2 is more efficient than OS_1 if $(gd_1 - gd_2) + (ge_1 - ge_2) \times T > 0$, where T denotes the expected times of invoking the host smart contract. Note that a smart contract can only be deployed once but will be executed many times.

2.2 Discovering Anti-Patterns

We identify the anti-patterns by looking for their instances from the execution traces of deployed smart contracts. An execution trace is an ordered sequence of executed EVM operations. We first collect N execution operations from the trace (N=10,000 by default), and then employ a sliding window to generate operation sequences. The window size (i.e., the number of operations in a window) ranges from 1 to 5 in this study. Consequently, 286,647 different operation sequences in total are collected. By manually scrutinizing all of those operation sequences, we discover lots of instances of anti-patterns and generalize them into 24 anti-patterns.

A simple way to get execution traces is to invoke the API, *debug.traceTransaction(txid)* provided by Ethereum. It will return the execution trace triggered by the transaction whose hash value is *txid*. However, this API is very slow because all transactions happened before *txid* will be replayed to construct the exact state for replaying the transaction *txid*, and hence this API is not suitable for batch processing. To address this issue, we propose a new approach that records the execution traces during synchronization by instrumenting EVM. Note that synchronization is necessary for the blockchain to ensure that all nodes are in the same state. More precisely, we insert logging code into each handler in EVM for interpreting the corresponding EVM operation (e.g., *opAdd()* is responsible for executing the addition operation). Moreover, to identify different transactions, we insert logging code into *ApplyTransaction()* in /core/state_processor.go, which is the entry point of applying individual transactions.

Fig. 1 illustrates how to extract an anti-pattern from an execution trace. The trace contains 347 EVM operations and we show a fragment with 5 operations for the ease of presentation. The content before '|' indicates EVM operations, and the numbers after '|' are the gas available after the execution of those operations. Hence, the gas for executing an operation can be computed by subtraction (e.g., the gas cost of add is $3 = 276,360 - 276,357$). Fig. 1 also shows the operation sequences extracted from the execution trace by sliding window. Different sequences are separated by ';', and we find an instance of anti-pattern (i.e., *P1*, underlined, explained later) by manually checking all 15 operation sequences.

2.3 Selected Anti-Patterns

We only present 5 anti-patterns due to page limit, and give a complete list in https://goo.gl/wJpAZ6. The operation sequence before $\rightarrow$ denotes an anti-pattern and the sequence after $\rightarrow$ is the corresponding efficient code after optimization (*delete* means the anti-pattern instance should be removed). Currently, the consistency of anti-patterns and their corresponding efficient code is examined manually. We will explore automated approaches (e.g., symbolic execution) in future.

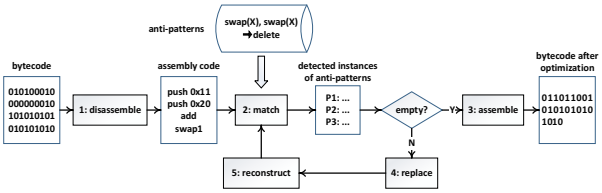

Figure 2: Workflow of `GasReducer`

P1 — {swap(X), swap(X)} $\rightarrow$ *delete*, $1 \leq X \leq 16$. () indicates a number inside instead of a part of an operation. swap(X) exchanges the 1st and the $(X+1)$th stack items. The two consecutive swap(X)s can be removed since the execution of them has no effect. The anti-pattern accounts for two bytes (e.g., the bytecode of swap1 is 0x90 [9]) and its deployment wastes 136 units of gas (every non-zero bytecode requires 68 units [9]). The execution of this pattern wastes 6 units of gas because the execution of a swap(X) needs 3 units [9].

P2 — {M consecutive jumpdests} $\rightarrow$ {jumpdest}, $M \geq 2$. jumpdest marks a valid jump target (i.e., any valid jump operation must jump

to a jumpdest) [9]. M consecutive jumpdests can be replaced with one because they point to the same operation (i.e., the operation after the final jumpdest). The deployment and invocation of this anti-pattern waste $68 \times (M-1)$ units and $(M-1)$ units of gas (the execution of a jumpdest needs 1 unit of gas [9]), respectively.

P3 — {OP, pop} → {pop}, OP ∈ {iszero, not, balance, calldataload, extcodesize, blockhash, mload, sload}. OP consumes the top stack item, produces an outcome, and pushes the outcome on stack [9]. pop removes the top stack item. This anti-pattern is semantically equivalent with one pop because the outcome of OP is immediately removed by pop. Due to page limit, we cannot explain the semantics of every EVM operation, and interested readers can refer to Ethereum's yellow paper [9]. The anti-pattern wastes 68 units of gas in deployment, and the amount of gas wasted by executing the anti-pattern instance depends on *gas_cost* of OP.

P4 — {swap1, swap(X), OP, dup(X), OP} → {dup2, swap(X+1), OP, OP}, $2 \leq X \leq 15$, OP ∈ {add, mul, and, or, xor}. dup(X) duplicates Xth stack item on stack top. Let the first three stack items be $i0$, $i1$ and $i2$, X be 2, and OP be add. After executing this anti-pattern, the number of stack items decreases by 1 and the top two items become $i0 + i1 + i2$, $i1$. Hence, one swap can be removed since OP(e.g. add) is commutative. The anti-pattern wastes 68 units and 3 units of gas for deployment and execution of a swap, respectively.

P5 — {OP, stop} → {stop}, OP can be any operation except jumpdest, jump, jumpi and all operations that change storage. stop halts the execution of the smart contract and then stack and memory are cleared [9]. For example, let OP be add, after the execution of stop, the outcome of add on stack will be discarded, and hence add can be removed. On the other hand, those operations involved in control flow transfer or changing the permanent storage space cannot be removed.

3 GASREDUCER

3.1 Workflow

We develop GasReducer to automatically detect all 24 anti-patterns in the bytecode of smart contracts and conduct bytecode-to-bytecode optimization by replacing the gas-costly code with efficient code. GasReducer (Fig. 2) takes in the bytecode of a smart contract and outputs the optimized bytecode. More precisely, GasReducer firstly disassembles the bytecode into assembly code (e.g., add), which is readable by human using the *disasm* command provided by Ethereum (step 1). Then, it conducts anti-pattern detection on the assembly code (step 2). If found, a report containing the location of each discovered instance of anti-pattern and the corresponding efficient code will be produced. Otherwise, the assembly code will be assembled into bytecode using the *asm* command (step 3). Leveraging the report, GasReducer replaces the instances of anti-patterns with efficient code (step 4) and then reconstructs the assembly code (step 5, Section 3.2). GasReducer iterates the detection and optimization until no anti-patterns can be found in the assembly code.

3.2 Reconstruction of Assembly Code

After replacing gas-costly code with efficient code, the structure of assembly code may be broken. Specifically, Ethereum's protocol specifies that the jump target of a control flow transfer (e.g., jump) should be determined in compilation and a jumpdest should be placed in the jump target [9]. The replacement procedure may cause the jump operation to point to a wrong position, leading to runtime exceptions when executing the optimized smart contract.

GasReducer reconstructs the assembly code by three steps. First, it computes the location of every jumpdest in the assembly code after optimization by searching for jumpdest according to the report. Afterwards, GasReducer associates a jump operation (e.g., jumpi) with its corresponding jumpdest. In most cases, a jump operation is immediately preceded by a push operation whose operand is the location of jumpdest. The association can be identified by matching the operand of push with the location of jumpdest. However, the matching approach does not always work because the jump target can be computed by a sequence of operations.

To tackle the issue, our approach uses *emulated execution* to find the association. Specifically, we execute EVM operations in an emulated environment until the jump operation, and then get the location of jump target on stack, because right before executing a jump, the location of its jump target should be stored on stack [9]. Note that it is enough to just emulate all stack operations (e.g., push1), arithmetic operations (e.g., add), bitwise operations (e.g., and) and comparison operations (e.g., iszero). Finally, we replace the sequence of operations that set the jump target with a push(X), $1 \leq X \leq 32$. The operand of that push(X) is obtained through emulated execution.

Table 1: Experimental results

	# contracts/traces	# instances	wasted disk (byte)	wasted gas
deploy	386,906	9,490,768	32,426,256	2,040,892,224
invoke	5,663,971	557,565,754	/	7,185,048,532

4 EXPERIMENTS

We conduct experiments to answer three research questions. **RQ1**: how many units of gas are wasted in deploying smart contracts? **RQ2**: how many units of gas are wasted in using smart contracts? **RQ3**: can existing compiler optimizations eliminate all anti-patterns?

To investigate the gas wasted in deployment, we need to obtain the bytecode of smart contracts. Since all smart contracts are stored in the blockchain and Ethereum provides the API (i.e., *getCode()*) to download bytecode, we use this API to download all (i.e., 599,959) deployed smart contracts as of 10 June, 2017. Since 145,907 smart contracts have no code (i.e., they have been removed from the blockchain), GasReducer processes the remaining 454,052 smart contracts. To examine the gas wasted in using smart contracts, we need to identify anti-pattern instances in the historical executions of smart contracts. It is non-trivial to obtain historical executions since they are not stored in the blockchain. We tackle this issue by instrumenting EVM to record all execution traces (detailed in Section 2). Consequently, we obtain all (i.e., 9,250,400) execution traces from the launch of Ethereum to 10 June, 2017 and adapt GasReducer to analyze them.

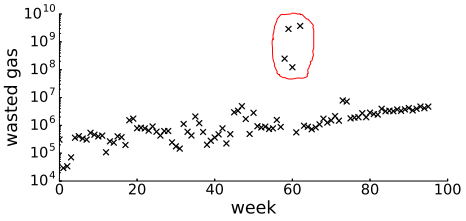

Figure 3: Amount of wasted gas

We have noticed that a proportion of smart contracts are not optimized (perhaps due to the ignorance of developers or the consideration of program stability), even if they have opportunities to be optimized during compiling. This observation does not offset the value of our empirical study because the deployed smart contracts cannot be modified due to the immutability of the blockchain. Table 1 lists the number of smart contracts or execution traces containing anti-patterns (column 2), the number of discovered anti-pattern instances (column 3), the size of disk space wasted to store under-optimized smart contracts (column 4), and the amount of wasted gas (column 5) in two scenarios including the deployment and the execution of smart contracts if applicable. These results empower us to answer RQ1 and RQ2: *the deployment and invocation of smart contracts waste 2,040,892,224 and 7,185,048,532 units of gas.* It is worth noting that the amount of wasted gas reported in the paper is the lower bound because the anti-patterns identified by us are by no means complete.

Fig. 3 shows the wasted gas detected in the execution traces of every week. We observe a steadily increasing trend (y-axis is in log scale) except four consecutive weeks (circled, will be explained later) as time goes on, indicating that users waste more gas with the increasing popularity of Ethereum. By checking the execution traces in the abnormal four consecutive weeks, GasReducer discovers 93,637,713 sequences of {extcodesize, pop} and 8,207,970 sequences of {balance, pop}. These sequences match *P3*, and can be replaced with a single pop. It turns out that our analysis result is accordant with the report about the DoS attacks aiming to exhaust the computing resources of Ethereum's nodes by repeatedly executing extcodesize, balance etc. [8]. More precisely, attackers exploit the fact that the two operations are resource-consuming but set low *gas_cost* [8]. Interestingly, our bytecode-to-bytecode optimization can prevent DoS attacks by replacing those resource-consuming sequences with a cheap pop.

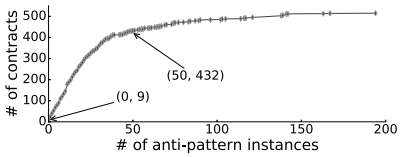

Figure 4: Number of instances in smart contracts

We further examine the effectiveness of optimizations in most recent and recommended compiler (i.e., Solidity 0.4.17) for smart contracts on removing anti-pattern instances. We download the source code of smart contracts from Etherscan (https://etherscan.io/), and compile them using Solidity 0.4.17 with optimization turned on. By applying GasReducer to the compiled smart contracts, we found that 506 out of 515 compiled smart contracts still have 15,205 instances of anti-patterns. After processing by GasReducer, 1,688,560 units of gas are saved. Each point (x, y) in Fig. 4 indicates that there are y smart contracts containing no more than x anti-patterns instances. Hence, the answer to RQ3 is: *existing compiler optimizations cannot eliminate all anti-patterns.*

5 RELATED WORK

We proposed GASPER in [4], which lists 7 anti-patterns and detects 3 of them. There are significant differences between it and GasReducer. First, the anti-patterns identified in this paper are bytecode-level operation sequences whereas the anti-patterns found in [4] are high-level structures (e.g., dead code). Moreover, we determine 24 anti-patterns whereas only 7 anti-patterns are listed in [4]. Third, GasReducer can detect 24 anti-patterns and remedy them whereas GASPER only detects 3 anti-patterns. Third, we quantify the gas wasted for deploying and executing smart contracts. Although there are studies on low-level anti-patterns [2] and approaches to derive anti-patterns automatically [1], they cannot be applied to our work directly for two reasons. First, they focus on accelerating execution instead of saving gas, and hence their anti-patterns are not proper to smart contracts. Second, since low-level anti-patterns are architecture-dependent, considerable efforts are needed to identify anti-patterns for EVM bytecode. Formal methods were used to check the properties of smart contracts [3]. OYENTE is a symbolic executor to discover security bugs in smart contracts [6].

6 CONCLUSION

We conduct the first in-depth investigation into under-optimized smart contracts by identifying 24 anti-patterns and developing GasReducer to automatically detect these anti-patterns from the bytecode of smart contracts and remedy them. Applying GasReducer to all deployed smart contracts and their execution traces, we detect lots of anti-pattern instances. This study could foster more research on the smart contracts, such as new development and optimization techniques for developers, new algorithms to drive all anti-patterns and remove them from smart contracts.

ACKNOWLEDGMENT

This work is supported in part by National Key R&D Plan (2016QY04X000), Sichuan Province Major Fundamental Research Project (2016JY0007), Scientific Supporting Plan of Sichuan Province, and HKPolyU Research Grants (G-YBJX). [§]Xiapu Luo is the corresponding author.

REFERENCES

[1] S. Bansal and A. Aiken. 2006. Automatic generation of peephole superoptimizers. In *Proc. ASPLOS.*
[2] S. Bansal and A. Aiken. 2008. Binary translation using peephole superoptimizers. In *Proc. OSDI.*
[3] K. Bhargavan and et al. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proc. PLAS.*
[4] T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Under-optimized smart contracts devour your money. In *Proc. SANER.*
[5] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. Au, and X. Zhang. 2017. An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks. In *Proc. ISPEC.*
[6] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making smart contracts smarter. In *Proc. CCS.*
[7] Angela Ruth. 2016. Why build decentralized applications: understanding Dapp. (2016). https://goo.gl/U5RBSs
[8] Jeffrey Wilcke. 2016. The Ethereum network is currently undergoing a DoS attack. (2016). https://goo.gl/QvW7Kj
[9] G. Wood. 2016. Ethereum: a secure decentralised transaction ledger, EIP-150 revision. (2016). http://gavwood.com/paper.pdf