Verification Assisted Gas Reduction for Smart Contracts

Anonymous Author(s)

ABSTRACT

Smart contracts are computerized transaction protocols built on top of blockchain networks. Users are charged with fees, a.k.a. gas, when they create, deploy or execute smart contracts. Since smart contracts may contain vulnerabilities which may result in huge financial loss, developers and smart contract compilers often insert code for security check. The trouble is that those code consume gas every time they are executed. Many of the inserted code are however redundant. In this work, we present *sOptimize*, a method that optimizes smart contract gas consumption automatically without compromising functionality or security. sOptimize works on smart contract byte code, statically identifies 3 kinds of code patterns, namely *dead code*, *opaque code*, and *part-opaque code*, and then removes redundant code through verification-assisted techniques. The resulting code is guaranteed to be equivalent to the original code and can be directly deployed on blockchain without any further modification. We evaluate sOptimize on a collection of 1,152 real-world smart contracts and show that it optimizes 43% of them, and reduces as high as 954,201 gas unit per contract.

KEYWORDS

smart contract, optimization, gas reduction

ACM Reference Format:

Anonymous Author(s). 2021. Verification Assisted Gas Reduction for Smart Contracts. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Smart contracts, as an innovative blockchain application, allow users to define complex protocols among distrusting parties. These protocols are strictly complied with by stakeholders through transactions, which invoke functions in smart contracts. The transactions together with the blockchain state are recorded by a large number of third-party entities, which are called *miners*. In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing completeness [33], users are charged with fees to execute transactions. The fees are calculated as *gas_price*∗*gas_amount* in Ethereum. *Gas_price* is the unit price of the gas, which is determined by the market (i.e., the miners). *Gas_amount* is the number of units of gas consumed for any computation or storage usage. It can be classified as *amount* that is consumed while deployment and *amount* that is consumed while transaction. In the former case, the cost is greatly affected by the size of the smart contract, since the size decides the storage needed. In the latter case, the cost depends on the operations executed, which amounts to the computation needed for each transaction. Every operation and every byte usage of storage are associated with a specific amount of gas, which is defined in [33].

Smart contracts are getting more and more popular in recent years, i.e., the number of transactions daily has increased from 7.1k in 2015 to 815k in 2020. At the same time, the gas consumption for each transaction on average also increased from 40K units to 70K [14]. Furthermore, after several high-profile contracts being attacked, security is more relevant a concern for contract programmers than ever. A common practice for preventing security problems is to apply standardized 'secure' libraries. Kondo *et al.* [21] report that the most frequently reused code block in smart contracts is the `SafeMath.sol` library from OpenZeppelin, which is a prominent project devoted to creating secure libraries and template contracts for smart contract developers.

These standardized secure libraries introduce run-time security checking code. For instance, once the `SafeMath.sol` library is adopted, run-time checks for possible overflow are introduced for every arithmetic operation in the contract. We foresee that such a practice will become increasingly popular (and rightfully so) and more and more run-time checks will be introduced due to the security concerns. As a result, more and more gas (in addition to time as well as energy) will be 'wasted' if some of these run-time checks are redundant. According to our analysis, there are as many as 43.3% contracts which contain such redundant instructions. The challenge is then: how do we reduce such gas consumption without sacrificing the security?

Studies related to gas reduction in smart contract have only recently attracted some attention. In [11], Chen *et al.* proposed GasReducer which identifies multiple anti-patterns from the execution traces of smart contracts and replaces these patterns with optimized code to reduce gas consumption. GASPER [9] applies symbolic execution to locate patterns which often consume excessive gas. Later, Gasol [3] proposes a cost model which allows users to infer the gas consumption for transactions and ensures the contract is free from out-of-gas vulnerabilities. Users can optionally choose the optimization mode which reduces the gas consumption associated to the usage of storage only. On the other hand, Nagele *et al.* proposed *ebso* [26] which leverages a constraint solver to automatically find an optimized alternative (through exhaustive search in a limited space) given certain code blocks. Albert *et al.* [4] attempted to find an optimized replacement for a block of code that produces the same result by applying *Max-SMT* techniques. These works mainly focus on finding gas-optimal instructions' sequence for a block, instead of whether the block is necessary in the first place.

In this work, we develop a toolkit called *sOptimize* which aims to reduce gas consumption for Solidity smart contracts by removing redundant run-time checks (which are typically introduced due to

ISSTA 2021, 12-16 July, 2021, Aarhus, Denmark

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

security concerns). *sOptimize* applies static analysis techniques (i.e., lazy annotation [25] and loop invariant generation techniques [22]) to verify whether a certain code block is redundant or not before optimization is applied. *sOptimize* focuses on optimizing 3 kinds of code blocks, i.e., *dead node*, *opaque node*, and *part-opaque node* (refer to Section 3.3 for detailed definitions). Given a smart contract, *sOptimize* automatically constructs a labeled control-flow graph (CFG) for each function. Each node in the CFG is annotated lazily with an invariant (which is initially true). The invariants are then monotonically strengthened through sound inference rules. Invariants associated with loop-head nodes are learned automatically with a combination of concrete testing, machine learning and symbolic execution. With the node invariants, we can soundly identify *dead nodes* and *opaque nodes*, which are subsequently removed for gas saving. In the case of *part-opaque nodes*, we redirect the parent node to the operation node directly, so that gas reduction is achieved for some (not all) transactions.

sOptimize is implemented with around 6,000 lines of C++ code. It has been systematically evaluated with a set of 1,152 smart contracts, each of which has more than 100 transactions. By comparing the original contract bytecode on Ethereum mainnet against the optimized one with the same transaction inputs on private chain, we demonstrate that on average 25,575 units of gas are reduced during contract deployment and 954,201 units of gas can be saved for the transactions for each contract at most.

The rest of the paper is organized as follows. In Section 2, we illustrate how *sOptimize* works through two simple examples. In Section 3, we present the details of our approach. In Section 4, we discuss the evaluation results. In Section 5, we review related works and lastly we conclude with a discussion on future work in Section 6.

2 OVERVIEW

Given a smart contract, the goal of *sOptimize* is to optimize its gas usage through detecting and eliminating redundant codes, i.e., unreachable instructions or unreachable branches which are often due to unnecessary run-time checks. In this section, we illustrate how *sOptimize* works through two examples. They are both excerpted from real-world contracts but modified for presentation.

Example1. In this first example, we highlight how invariant learning helps to identify opportunities for optimization. The `multiSend` contract[1] shown in Figure 1 attracts users to join the contract as a bounty hunter by sending 0 ether (i.e., unit of cryptoconcurrency in Solidity) to the contract owner. Afterwards, the contract owner allocates `amount` tokens to the users' addresses by invoking function `setDistributeToken`. This function first adds the user's address into `bountyAddr` if this user never joins the contract before, and then allocates the token to the users at line 8. In this example, we aim to identify and remove the dead branches which are never executed and the condition which is an unnecessary check caused by a mistake at line 6.

sOptimize first constructs the CFG of the `setDistributeToken` function as shown in Figure 1b. In this figure, node *root* and node *stop* represent the entry and exit of the function respectively, and other nodes represent the corresponding statements in the contract. The predicates (in blue) associated with the nodes are node invariants, and the predicates in red are node assertions. In this example, assertions are introduced by the *Solidity* compiler for boundary check before the array is accessed every time (e.g., $i < addrs.length$ for `addrs` array). We depict all the assertions in red at nodes n_{6_0}, n_{7_0} and n_{8_0} in Figure 1b. They are all derived from the array `addrs[]` at line 6, line 7 and line 8 in Figure 1a. *Solidity* first checks whether the index i is in the range of the array length at node n_{6_0}, and then checks whether the condition $setAmount[addrs[i]] < 0$ is satisfied at node n_6. Similar checks are in place also for node n_{7_0} and node n_{8_0}.

Next, *sOptimize* infers the invariants for each node using a combination of program inference, lazy annotation and loop invariant learning techniques. Initially, the invariant of each node is $true$. *sOptimize* iteratively and monotonically strengthens the node invariants step by step. To infer the invariant for the loop-head node (i.e., a node representing the start of a loop), *sOptimize* invokes a *loop invariant generator* to learn an invariant, which is subsequently propagated to the nodes in and after the loop. Take node n_5 as an example, which is the head node of the loop started with an edge from node *root* and ended with an edge to *stop* in Figure 1b. *sOptimize* invokes the *loop invariant generator* for invariant inference. During the learning process, *sOptimize* first generates random valuations of all relevant variables (including i, $addrs.length$, $bountyAddr.length$ and $amount$), and then categorizes the valuations according to whether any of the assertions is violated or not. Afterwards, *sOptimize* invokes a *learner* to generate a candidate invariant which is then validated by a *validator*. If the candidate invariant is not valid, a counterexample in the form of variable valuations is generated and used to learn a new candidate invariant until a valid invariant is generated. In Figure 1b, the learnt invariant is $true$, that means the assertion $i < addrs.length$ is always satisfied at node n_{6_0} as well as node n_{7_0} and n_{8_0} in the loop. Note that there is an implicit condition in this contract which is that any element in $setAmount$ is non-negative, since the element is defined as *uint* at line 2. Thus the invariant of node n_6 is strengthened as $true \wedge i < addrs.length \wedge setAmount[addrs[i]] >= 0$, node n_{7_0} is $true \wedge i < addrs.length \wedge setAmount[addrs[i]] >= 0 \wedge setAmount[addrs[i]] < 0$ (equivalent to $false$), and node n_7 is $false$. Notice that, we simplify the invariant of nodes n_{8_0} and n_8 to be $true \wedge i < addrs.length$.

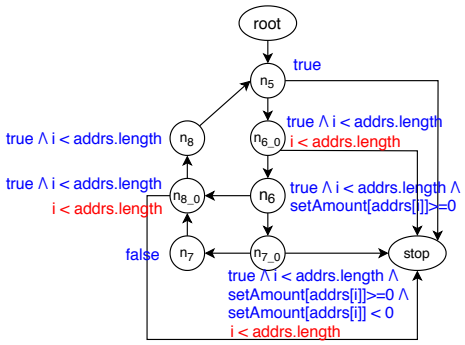
Once the invariant of each node is inferred (and a fixed-point has been reached after propagation), *sOptimize* checks whether there exists dead nodes or opaque nodes based on the CFG. To identify dead nodes, *sOptimize* checks whether the invariant of a node is evaluated to $false$. If so, such node is removed. To identify opaque nodes, *sOptimize* evaluates whether the implication between the node invariant and the branch condition or the negation of the branch condition of the node is successful. If it is the case, *sOptimize* removes the branch node and redirects the edge from its parent node(s) to the child node(s).

In this example, nodes n_{7_0} and n_7 are dead nodes and n_{6_0}, n_6 and n_{8_0} are all opaque nodes. They are thus removed and a new edge is generated to link the node n_5 with node n_8 directly as shown in Figure 1c, i.e., codes at lines 6 and 7 which are removed after optimization.

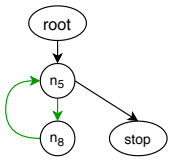
[1] contract address: 0x2deF52220E91EB42B2CaF8005F7f671dC692Bf89

```
1  contract multiSend {
2      mapping (address => uint) setAmount;
3      address[] bountyAddr;
4      function setDistributeToken(address[] memory addrs,
           uint amount) public{
5        for(uint i = 0; i < addrs.length; i++){
6          if(setAmount[addrs[i]] < 0)
7              bountyAddr.push(addrs[i]);
8          setAmount[addrs[i]] += amount;
9        }
10     }
11 }
```

(a) Unreachable Branches in Loop



(b) Labeled CFG



(c) Optimized CFG

Figure 1: Optimization for Loop Contract

```
1  library SafeMath {
2      function mul(uint256 a, uint256 b) internal pure
           returns (uint256){
3          if (a == 0)
4              return 0;
5          uint256 c = a * b;
6          assert(c / a == b);
7          return c; }
8      function div(uint256 a, uint256 b) internal pure
           returns (uint256){
9          uint256 c = a / b;
10         return c; }
11 }
12 contract ethBank{
13     using SafeMath for *;
14     address public owner;
15     uint rate;
16     uint constant ethWei = 1 ether;
17
18     modifier onlyOwner{require(msg.sender == owner); _;}
19
20     function() payable external{
21         require (msg.value == msg.value.div(ethWei).mul(
               ethWei), "invalid msg value"); }
22     function withdrawForUser(address payable _address, uint
           amount) onlyOwner public{
23         require(msg.sender == owner, "only owner ...");
24         uint pay = rate.mul(amount);
25         _address.transfer(pay); }
26 }
```

(a) Source Code

```
...
01b2:JUMPDEST
01b3:PUSH1 0x00
01b5:DUP3
01b6:DUP5
01b7:MUL
01b8:SWAP1
01b9:POP
01ba:DUP3
01bb:DUP2
...
01be:PUSH2 0x01c3
01c1:JUMPI

01c2:INVALID

01c3:JUMPDEST
01c4:DIV
01c5:EQ
01c6:PUSH2 0x01cb
01c9:JUMPI

01ca:INVALID

01cb:JUMPDEST
01cc:DUP1
01cd:SWAP2
01ce:POP
01cf:POP
...
```

(b) Part-opaque Node

```
...
024c:PUSH2 0x0254(tag7)
024f:JUMPI
...
0254:JUMPDEST(tag7)
...
02a5:PUSH2 0x0316(tag8)
02a8:JUMPI

02a9:PUSH1 0x40
...
0315:REVERT

0316:JUMPDEST(tag8)
```

(c) Potential Op-Nodes

```
...
0380:JUMPDEST
0381:PUSH1 0x00
0383:DUP3
0384:DUP5
0385:MUL
0386:SWAP1
0387:POP

0388:DUP1
0389:SWAP2
038a:POP
038b:POP
...
```

(d) Op-part-opaque Node

Figure 2: Optimization for Common Contract

Example2. In this second example, we show how different nodes are optimized at bytecode level. As shown in Figure 2a, contract `ethBank`[2] receives deposit in the form of ethers from the fallback function, and allows the owner only to transfer ethers out to addresses specified by the owner in function `withdrawForUser`. The modifier function `onlyOwner` at line 18 restricts the user to access a function when it is used.

All three kinds of nodes which are subject to optimization are identified by *sOptimize* in this contract. The opcodes for line 23 after compilation is shown at Figure 2c. The node from line 0254 to line 02a8 is identified as an opaque node; and the node from line 02a9 to line 0315 is a dead node. Since the modifier `onlyOwner` functioning at line 22 allows only the owner to proceed to execute, the *require* statement at line 23 must be satisfied. The corresponding checking node starting from line 0254 at Figure 2c always jumps to 0x0316. As a result, the node starting from line 02a9 is never reached. Thus, we can redirect the edge to line 0316 directly from line 024c and remove all the opcodes from line 0254 to line 0315. A total of 108 bytes are removed, which saves 28,944 units of gas during deployment (i.e., 68 per byte for transaction and 200 per byte for run-time code deposit), and 269 units of gas for each transaction afterwards.

This example also contains a part-opaque node. The assert statement at line 6 from the `SafeMath` library is redundant when it

[2]contract address: 0xf3fa62dd25504a7b05300a1ddd56a22a100fd4df

Algorithm 1: Overall Optimization Algorithm

```
1  {CFG_f_i} ← CFG_construct(C);
2  for CFG_f_i do
3      CFG_f_i ← updateInv(CFG_f_i);
4      OP_f_i ← CFG_f_i;
5      for node n ∈ set(N) do
6          if I(n) = false then
7              OP_f_i ← rmDeadNode(CFG_f_i, OP_f_i, n);
8          end
9          OP_f_i ← opONode(CFG_f_i, OP_f_i, n);
10         OP_f_i ← opPartONode(CFG_f_i, OP_f_i, n);
11     end
12 end
13 reOrganizeBytecode({OP_f_i});
```

is invoked from the fallback function at line 21 in Figure 2a. The corresponding bytecode sequence is at line 01b2 to line 01ca in Figure 2b. It prevents potential overflow problem caused by the multiplication at line 5. However, overflow is impossible in such a case since `ethWei` is a constant. That means $c/a = b$ is always true. We cannot remove this statement directly, because it still works for other cases like the multiplication at line 24. *sOptimize* generates a new copy of function `mul` and removes the redundant codes in the new code snippet as shown in Figure 2d, which is optimized from line 01b2 to line 01cf in Figure 2b. Afterwards, the fallback function is directed to this new bytecode sequence to avoid the redundant checks. Note that this copy introduces 42 bytes new codes in this example which cause an increase of 11,256 units of gas during deployment and at the same time reduces 46 units of gas for each transaction subsequently. It means that this optimization is profitable if the transaction volume is larger than 250.

3 OUR APPROACH

In this section, we present our approach step-by-step in detail. The overall approach is shown in Algorithm 1. Given a smart contract C with M functions, we first construct a CFG for each function (at line 1). Then, we synthesize an invariant for each node in the CFG by function $updateInv(CFG_{f\_i})$ to update CFG_{f_i} at line 3. Initiate OP_{f_i} to keep the optimized CFG for each function with updated CFG_{f_i} at line 4. $set(N)$ is the set of all the nodes in the CFG, which is defined in Definition 2. Next, we examine every node in the CFG to systematically identify and optimize dead nodes and opaque nodes including part-opaque nodes from line 5 to line 11. Lastly, we redirect the edges of the optimized CFG OP_{f_i} to output the bytecode sequence through function $reOrganizeBytecode(OP_{f\_i})$ at line 13. In the following, we present details of the main steps.

3.1 CFG Construction

In this step, we systematically construct the CFG of each function in the smart contract. Given the bytecode of a smart contract, the CFG is constructed based on the compiled Ethereum Virtual Machine (EVM) opcode. In the following, we present a core set of the opcodes.

Definition 1 (Opcode). *An EVM opcode (Opcode) is defined as follows.*

$$
\begin{array}{lcl}
Opcode & ::= & Sta\_OP \mid Mem\_OP \mid Sto\_OP \mid Bran\_OP \mid \\
 & & Arith\_OP \mid Logic\_OP \mid Cmp\_OP \\
Sta\_OP & ::= & PUSHi \mid DUPj \mid SWAPj \mid POP \\
Mem\_OP & ::= & MLOAD \mid MSTORE \\
Sto\_OP & ::= & SLOAD \mid SSTORE \\
Bran\_OP & ::= & JUMP \mid JUMPI \\
Arith\_OP & ::= & ADD \mid MUL \mid SUB \mid DIV \mid MOD \mid EXP \\
Logic\_OP & ::= & NOT \mid AND \mid OR \mid XOR \\
Cmp\_OP & ::= & ISZERO \mid LT \mid GT \mid EQ
\end{array}
$$

Each opcode is associated with an operational semantics, as specified in [18, 19]. In the following, we briefly and intuitively present the semantics of the most relevant opcodes. We refer the readers to [18, 19] for a full discussion of the semantics, based on which we can systematically construct the CFG. An opcode can be a stack operation, memory operation, storage operation, branching operation, arithmetic operation, logic operation, and comparison operation. Stack operations include pushing bytes onto the stack ($PUSHi$), cloning the top value on the stack ($DUPj$), swapping the top two values on the stack ($SWAPj$), and removing the value from the top of the stack (POP). The range for i is from 1 to 32 and j is from 1 to 16 respectively. Memory operation ($MLOAD/MSTORE$) reads/writes a value from/to memory. Storage operation $SLOAD$ reads a value at a position from storage, while $SSTORE$ writes a value to a position in storage. Branching operations include unconditional jump ($JUMP$) and conditional jump $JUMPI$ which moves to the destination if the condition evaluates to be true. An opcode can also be an arithmetic operator (ADD, MUL, SUB, DIV, MOD and EXP), Boolean operator (NOT, AND, OR and XOR) or comparison operator ($ISZERO$, LT, GT and EQ).

A function of a smart contract is composed of a sequence of opcodes. Typically the opcodes are organized into nodes, i.e., a sequence of opcodes which do not contain a branching operation, other than the last one in a node.

Definition 2. *Given a function of a smart contract, its CFG is a 4-element tuple $(N, root, E, \mathcal{I})$ where N is a set of nodes, each of which represents a block of opcodes and these opcodes have the same node label; $root \in N$ is the entry node; $E \subseteq N \times N$ is a set of edges; $\mathcal{I} : N \rightarrow Pred$ is a function that labels each node with an invariant.*

Note that an edge from a block n to another block n' is in E if and only if the (conditional or unconditional) $JUMP$ at the end of n leads the execution to n'. While it is conceptually simple, constructing the CFG in practice is non-trivial. That is, given the bytecode of a smart contract C, we first disassemble the bytecode into a sequence of EVM opcode instructions, and then construct the CFG for each function based on the opcode instructions. To identify the edges of the CFG, we must figure out the target of `JUMP` and `JUMPI` instructions, which may depend on what is on the stack. Thus, we simulate the stack completely in our approach, i.e., by executing those stack related operations precisely. At the same time, some nodes may be visited multiple times, like the nodes in the library in Figure 2a. We would duplicate such nodes according to different control flows. Readers are referred to [2, 7] for details on how the CFG is constructed. Initially, the node labeling function $\mathcal{I}$ is defined such that $\mathcal{I}(n) = true$ for every $n \in N$.

$$\text{SSTORE } (p,v)\ \frac{V' = V[storage(p) \mapsto v]}{(n, pc, V) \longrightarrow_s (n, pc+1, V')}$$

$$\text{JUMPI } (cond, T)\text{-1}\ \frac{V \models cond}{(n, pc, V) \longrightarrow_s (n', T, V)}$$

$$\text{JUMPI } (cond, T)\text{-2}\ \frac{V \not\models cond}{(n, pc, V) \longrightarrow_s (n', pc+1, V)}$$

Figure 3: Instruction Execution rules

Example 3.1. The CFG in Figure 1b starts from the root node, and jumps to node n_5 only, which forms an edge between root node and node n_5. As the execution goes on, all the nodes and edges are resolved in the 4-element tuple $(N, root, E, \mathcal{I})$ tuple, which forms the completed CFG lastly shown in Figure 1b.

3.2 Invariant Generation

In this step, we systematically strengthen invariants for each node in the CFG. We first define what is an invaraint based on the semantics of the function. Hereafter, we refer to the function and its CFG interchangeably.

Definition 3 (Symbolic Semantics). *Let $(N, root, E, \mathcal{I})$ be a function of a smart contract, its (symbolic) semantics is defined as a labeled transition system $(S, init, \longrightarrow_s, \mathcal{I})$, where S is a set of symbolic states, and each state s is a pair (n, pc, V) where $n \in N$, pc is the program counter of opcodes in a node and V is a symbolic valuation function which maps each storage variable to an expression constituted of symbolic variables; $init \in S$ is the initial state composed of root and the initial valuation of pc and the storage variables (which are all symbolic); $\longrightarrow_s \subseteq S \times S$ is the transition relation conforming to the symbolic semantic rules.*

A few simplified execution rules are shown in Figure 3 to make this paper self-contained. We refer the readers to [16] for a complete list of semantic rules, as it is not the main contribution of this work. Here, rule *SSTORE* updates the position p with v in V', and moves pc to the next opcode. Rule *JUMPI* moves pc to a new location that depends on the symbolic valuation V and JUMPI condition $cond$. If V satisfies the condition $cond$, pc will be moved to the new target T, and correspondingly, n is updated to n'. We define a mapping function *pc2node* to decide the node label n according to the value of pc. Here, n' is the same as the value of the new pc, which is T. Otherwise, pc is moved to the succeeding opcode, which is $pc+1$. The node n is also updated by function *pc2node* to the new node, whose value is $pc+1$.

A (symbolic) trace tr is a sequence of symbolic states in the form of $tr = \langle s_0, s_1, \ldots, s_{k+1} \rangle$, where $s_0 = init$ and $s_i \longrightarrow_s s_{i+1}$ for all $0 \leq i \leq k$. We write $last(tr)$ to denote the last state of the trace, i.e., $last(tr) = s_{k+1}$. The set of symbolic traces of a function F, written as $Trace(F)$, is the set of all traces which can be generated according to the symbolic semantics.

Definition 4 (Node Invariant). *Given a smart contract function $F = (N, root, E, \mathcal{I})$, a predicate ϕ is an invariant at node n, denoted as $\mathcal{I}(n) = \phi$, if and only if $last(tr) \models \phi$ for all $tr \in Trace(F)$ s.t. $\pi_n(last(tr)) = n$.*

Algorithm 2: Invariant Inference $inferI(F, n)$

```
1 Ψ ← false;
2 for (m, n) ∈ E do
3 |   Ψ ← Ψ ∨ φ(m);
4 end
5 if Ψ ≠ false then I(n) ← I(n) ∧ Ψ ;
```

Note that $v \models \phi$ means ϕ is satisfied by the variable valuation of v. Intuitively, the above definition of state ϕ is an invariant at node n if and only if ϕ is satisfied by all the traces leading to node n, i.e., when the trace reaches n, its variable valuation satisfies ϕ. Function π_n maps the state to the corresponding node n.

Definition 5 (Strongest Postcondition). *Given an opcode op and a precondition ϕ, the strongest postcondition $sp(c, \phi)$ is defined as:*

$$\begin{aligned}
&sp(SSTORE(p,v), \phi) = \phi \oplus (storage[p] \mapsto v)\\
&sp(MSTORE(p,v), \phi) = \phi \oplus (mem[p] \mapsto v)\\
&sp(op, \phi) = \phi \wedge b && \text{if } op = JUMPI(b)\\
&sp(op, \phi) = \phi && \text{if } op = JUMP\\
&sp(op, \phi) = \phi && \text{if } op = SLOAD(x) \text{ or } MLOAD(x)
\end{aligned}$$

In the above definition, the predicate ϕ is overwritten by predicate $(storage[p] \mapsto v)$ for *SSTORE* command. Note that symbol $\oplus$ overwrites the predicate related to $storage[p]$[3] if it exists; otherwise, the postcondition is the conjunction of the predicate and ϕ. The strongest postcondition of *MSTORE* is similar except overriding the memory location. For the branching command *JUMPI*, the strongest condition is the conjunction of ϕ and condition b. Since there is no condition introduced for *JUMP*, the strongest postcondition keeps the same. For command *SLOAD* or *MLOAD*, the strongest postcondition is ϕ. Worthing to say, the strongest postcondition for *MLOAD* is easy to understand, since memory is volatile in Ethereum, there must be a *MSTORE* to save the content before *MLOAD*, thus the strongest postcondition stays the same. *SLOAD* may introduce new predicate when the storage position is first visited, however, the content must be in some other forms integrated into the strongest postcondition, like assigning to other variable or acting as a part of the branch condition. Thus, it keeps the same.

Algorithm 2 shows details on how to update the invariant of a node n based on the strongest postcondition. Let Ψ be a predicate which is initially $false$. We have the strongest postcondition of each node m linking to node n, which is $\phi(m)$. Their disjunction is a constraint which must be satisfied by the invariant at node n. Intuitively, this is because n can only be reached via one of its parents. Lastly, at line 5, we set the invariant at node n to be the conjunction of $\mathcal{I}(n)$ and Ψ so that it is monotonically strengthened over time. The condition at line 5 ensures that any node which has no parent node like the root node is not updated.

[3] which is implemented through variable elimination

Algorithm 3: $updateInv(CFG(F))$

```
1  I' ← ∅;
2  while I' ≠ I do
3  |  I' ← I;
4  |  for n ∈ N do
5  |  |  if n is loop head then
6  |  |  |  I(n) ← generateLI(CFG(F), n);
7  |  |  else
8  |  |  |  I(n) ← inferI(CFG(F), n);
9  |  |  end
10 |  end
11 end
```

Then, how do we generate non-trivial invariants? As shown in Algorithm 3, we adopt two ways to generate the invariants depending on whether a node is a head-node for a loop or not. Note that thanks to structural programming, we can always identify the head-node for a loop based on the CFG. If the node is not the head of a loop, it is inferred by function $infer I(F, n)$ which is the disjunction of the strongest post-condition of all its parents. The strongest post-condition is computed based on the invariant of its parent node.

If the node is the head of a loop, we generate the loop invariant through a "guess and check" approach, which is adopted from [22]. Intuitively, the loop invariant learning function $generateLI(F, n)$ is composed of three phases, i.e., *data labeling*, *learning*, and *validation*. *sOptimize* executes the loop part with the concrete variable valuations and labels these valuations as negative or positive samples against assertions. Note that in addition to assertions provided by users or added by the compiler, we automatically instrument the negation of the condition before every branch node as the assertion (so that we can check the feasibility of each branch). A concrete variable valuation is labeled positive if no assertion is violated during the execution; otherwise, it is labeled negative. Based on the labelled samples, *sOptimize* learns an invariant using a classification algorithm (such as SVM [12] and the decision tree [30]). The learnt candidate invariant is then validated by the validator by checking whether the invariant still holds after one iteration of the loop through symbolic execution. If the candidate invariant fails the validation, i.e., there exists a concrete variable valuation (hereafter a counterexample) which satisfies the candidate invariant before the loop and fails the candidate invariant after one iteration, the counterexample is added into the set of labelled samples to learn a new invariant. Once validated, the candidate invariant is returned as the output of function $generateLI(F, n)$. Since learning loop invariant is not the main contribution of this work, we skip the detail and refer the readers to [22] for details.

Example 3.2. Given the example shown in Figure 1a, *sOptimize* learns invariant for the loop shown in Figure 1a against the compiler-inserted assertion ($i < addrs.length$). A set of samples is randomly generated first. The tricky part is that, if a sample can go into the loop, it must satisfy the assertion since the loop condition is same as the assertion. Thus, *sOptimize* proposes a candidate invariant

Algorithm 4: $opONode(CFG(F), OP(F), n)$

```
1 if n ∈ {b?n1 : n2} then
2 |  if I(n) ⇒ b then
3 |  |  OP(F) ← rmOpaqueNode(CFG(F), OP(F), n, n1));
4 |  end
5 |  if I(n) ⇒ ¬b then
6 |  |  OP(F) ← rmOpaqueNode(CFG(F), OP(F), n, n2);
7 |  end
8 end
```

Algorithm 5: $opPartONode(CFG(F), OP(F), n)$

```
1 if n is duplicate node && n ∈ {b?n1 : n2} then
2 |  if I(n) ⇒ b then
3 |  |  OP(F) ← dupPartONode(CFG(F), OP(F), n, n1));
4 |  end
5 |  if I(n) ⇒ ¬b then
6 |  |  OP(F) ← dupPartONode(CFG(F), OP(F), n, n2));
7 |  end
8 end
```

of *true* as only positive samples can be collected at this step. Symbolic execution is then used to validate this invariant, we collect the precodition at the beginning of the loop as $i = 0$. Obviously, $(i = 0 \rightarrow true)$. Next, propagate the node invariants in the loop with *true* from the loop head. Just as shown in Figure 1a, the invariants of node n_6_0 and n_8_0 are both $(true \wedge i < addrs.length)$, the node invariant of node n_7_0 is equivalent to $false$, they can all imply the corresponding assertions successfully. For the inductive inference, since the candidate invariant is always *true*, the inference is always successful. Thus, this candidate invariant is successfully validated by the validator. It is returned as the valid invariant for this loop.

3.3 Contract Optimization

In the following, we present how to optimize the contract based on the CFG (whose nodes are now labeled with invariants). In this work, we focus on three ways to optimize the contract, i.e., removing dead nodes, removing opaque nodes and duplicating part-opaque nodes.

Definition 6 (Dead Node). *Given a CFG $(N, root, E, \mathcal{I})$ (which represents a function in a smart contract), a node n in N is a dead node if and only if $\mathcal{I}(n)$ is false.*

Intuitively, a node is a dead node if all symbolic traces reaching the node are infeasible, i.e., the node invariant evaluates to $false$. For the example shown in Figure 2c, the node starting from line 02a9 to line 0315 is a dead node. It is straightforward to see that dead nodes can be removed from the CFG directly without affecting any feasible traces and thus are safe to remove.

Before to define *Opaque Node* and *Part-opaque Node*, we introduce the conception of duplicate nodes, which are duplicated from another node. As mentioned before, there may be multiple nodes which share the same opcode sequences in a CFG, this is caused

by different control flows. We define *Opaque Node* and *Part-opaque Node* which are relevant with such nodes.

Definition 7 (Opaque Node). *An opaque Node is a branch node but not a duplicate node, whose node invariant can successfully imply its branch condition or the negation of the branch condition in the labelled CFG.*

Definition 8 (Part-opaque Node). *A part-opaque node is a duplicate branch node, whose node invariant can successfully imply its branch condition or the negation of the branch condition.*

As shown in Algorithm4, if the invariant $I(n)$ of a branching node n can successfully imply the branch condition, and n is not a duplicate node, that means the edge always starts from node n and stops at node $n1$, we invoke the function *rmOpaqueNode* to link the parent node of n to node $n1$ directly, and remove the current node n. Otherwise, if the invariant $I(n)$ of node n can successfully imply the negation of the branch condition as shown in line 5, we link the parent node of n to node $n2$, and remove the current node n. For instance, in Figure 2c, the node starting from line 0254 to line 02a8 is an opaque node, whose node invariant is ($msg.sender = owner$) which is due to the modifier in Figure 2a, and the branch condition is also ($msg.sender = owner$) which maps to line 21 in Figure 2a. Thus the implication is always successful, this node will always go to node starting from line 0316. After invoking procedure *rmOpaqueNode*, the opaque node is removed from the CFG and the tag at line 024c is updated to tag8 to form the new edge.

Identifying part-opaque nodes resembles identifying the opaque nodes, as shown in Algorithm 5, *sOptimize* first checks whether the node is a duplicate node. If it is a duplicate node, *sOptimize* further check whether it is a part-opaque node. If the check succeeds, *sOptimize* invokes the function *dupPartONode* to instrument new nodes which are a modified copy of current code paragraph for specific function module. As the example shown in Figure 2b, *sOptimize* discovers the nodes starting from line 01b2 to line 01c1 and from line 01c3 to 01c9 are both part-opaque nodes, which maps to the assert statement at line 6 in function *mul* invoked by statement at line 21 in Figure2a. Since the assertion never fails in such a case, the node always goes from node 01b2 through node 01c3 to node 01cb. Function *dupPartONode* instruments a new modified copy of nodes starting from line 01b2 to line 01cf as shown in Figure 2d, which removes the part-opaque nodes, together with others to the end of the contract CFG. Obviously, this optimization suffers overhead (i.e., extra code is introduced), and we only allow single copy of nodes in our implementation. Too much copies may introduce too many codes and increase the gas cost.

3.4 Bytecode Reorganization

After optimization, we reorganize the control flow for all the nodes in the optimized CFG. We flag the opcode which determines the control flow among nodes when constructing CFG, and the corresponding tag sequence for each node. As shown in 2c, the target for line 024c is 0x0254, which is labelled with a tag of tag7, we also link line 02a5 with 0x0316 by tag8 in the same way before optimization. After removing the redundant codes, we update the tag at line 024c with tag8, and further recalculate the target address for all the *PUSH* opcodes. Finally, *sOptimize* outputs the reorganized bytecode.

Theorem 3.3. *Algorithm 1 is sound.*

Proof. The soundness of Algorithm 1 is established on the fact that all inferred invariants are indeed invariants. There are two ways of inferring invariants, either by Algorithm 2 or by the "guess and check" approach. In the former case, the inferred invariant is indeed an invariant according to Definition 4. In the latter case, the correctness of the inferred invariant generated by *generateLI* is ensured by the validator which checks whether the learned invariant is inductive. Given that all inferred invariants are sound, Algorithm 1 is sound as it removes the dead nodes only when the node invariants are $false$, removes the opaque nodes or duplicates the part-opaque nodes when the branch condition can be implied by the node invariants. □

The complexity of the algorithm is $o(n)$ without considering the complexity of the invariant learning procedure, since the learning process is a guess-and-check based method, it is very hard to estimate the complexity especially when involving the concrete execution of the contract. We thus evaluate it empirically in the next section.

4 IMPLEMENTATION AND EVALUATION

sOptimize is implemented in C++ with about 6,000 lines of code. The smart contract is first compiled into EVM bytecode and further disassembled into EVM opcodes with the help of Solidity compiler and Ethereum toolkit. *sOptimize* then constructs labelled CFG with EVM opcodes to get node invariants and node assertions for each node. To update the node invariants of loop-related nodes, *sOptimize* implements the LinearArbitrary algorithm based on LIBSVM [6] and C5.0 [29]. Z3 SMT solver [13] is adopted to check the satisfiability of constraints in the candidate validation phase and the node invariants in the redundant nodes identification phase.

4.1 Evaluation

In the following, we evaluate the effectiveness and efficiency of *sOptimize* in practice by answering the following research questions (RQ).

- **RQ1:** Are there many redundant opcodes in Ethereum smart contracts?
- **RQ2:** Is *sOptimize* effective in reducing gases in practice?
- **RQ3:** How much are the overhead by *sOptimize*?
- **RQ4:** Is *sOptimize* efficient?

Note that there is no comparison design against other tools in above RQs, as *sOptimize* is the only publicly tool which aims to reduce gas consumption in smart contracts. Some tools are not open-source (e.g., GasReducer, and GASPER etc.), some are designed for different purposes (e.g., Gasol infers gas consumption.), others are optimization tools for instructions' sequence (e.g., *ebso* and *syrup*), which concentrate on the optimization of within a block, and moreover, part of the tool (which converts the target bytecode blocks to SFS [4], an intermediate form) is not available currently.

We collected 8,140 verified solidity contracts with open-source licenses on Etherscan[4], a leading BlockChain Explorer for Ethereum. Since a lot of contracts have few transactions, which may limit the

[4]Accessed on https://etherscan.io/exportData?type=open-source-contract-codes as of Jun.14, 2020.

Table 1: Average Information for optimized Contracts

RT_Size(bytes)	D_Node(bytes)	O_Node(bytes)	D_GasReduce	T_GasReduce
5,616	96.1	15.8	29,900	328

evaluation of effectiveness regarding the total gas consumption which is proportional to the transaction volume, we select 1,263 contracts whose transactions are more than 100 to evaluate the performance of *sOptimize*. The highest transaction for contract is 999,366, and the total transactions for all selected contracts are 9.4 million as of June 14, 2020.

All experiment results are obtained on a machine running on Ubuntu 16.04 with EVM version 1.9.10. The detailed hardware configuration is 2.8 GHz x 8 Intel processor, 23.4 GB ram.

Identification and Optimization of Redundant Codes. To conduct the experiment, we further acquired the detailed information from Etherscan, such as compiler versions, optimize options and deployed contract names. The timeout set for *sOptimize* is: global wall time, 3600 seconds and Z3 solver timeout, 10 seconds.

sOptimize identified 499 contracts that can be optimized by removing redundant instructions from 1,152 (43.3%) contracts which finish in wall time. The average statistics of the identified contracts is shown in Table 1, column *RT_Size* shows the average size of run-time bytecode, columns *D_Node* and *O_Node* are the size of dead node and opaque node identified by *sOptimize* in bytes. Columns *D_GasReduce* and *T_GasReduce* stand for the average gas unit reduced when a contract is deployed to the blockchain and executed in a transaction. Column *D_GasReduce* is calculated with $bytes\_removed * 268$ and column *T_GasReduce* is the summation of the gas consumption for each executed instruction defined in Ethereum yellow paper [33]. Note that we calculate the number with the base case, which is the minimum gas reduced if those instructions are executed. We can see the optimized bytes take a portion of 2.0% $((96.1 + 15.8)/5616)$ against the contract bytecode size, which causes a decrease of gas consumption of 29,900 when the contract is deployed to the blockchain. The gas reduction of the transaction depends on the transaction volumes, each transaction can reduce 328 units of gas, the more frequently the optimized codes are invoked, the more gas is reduced.

To answer RQ1: About 43.3% test subjects on average can be optimized and the contract size can be reduced 2.0%, which can save 29,900 unit of gas for deployment and 328 units of gas can be saved for each transaction related to these nodes at least.

Effectiveness of sOptimize. We intend to study the effectiveness of *sOptimize* through comparing the total gas consumption of all transactions between the optimized contracts and the original contracts from the Ethereum Mainnet. We build up private chains in docker containers with the same setup. To minimize the computation resources, consensus of proof-of-authority is adopted and the block time/interval is set to 3 seconds to accelerate the mining rate. Then, we deploy the optimized contracts and the original contracts respectively on two docker containers, replay all the transactions with the same input downloaded from Ethereum Mainnet on the private chains, and record the gas consumption for the deployment and the transactions. We deployed 212 contracts to the private chain

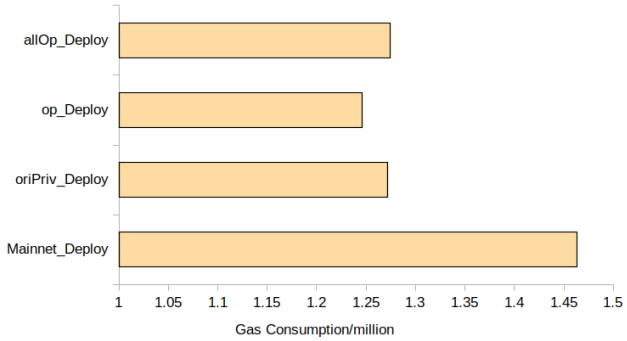


Figure 4: Gas Consumption for Deployment

(omit those contain a special opcode COPDECOPY in runtime bytecode currently, which requires complex adjustment on the offset and length).

The results are shown in Table 2 and Table 3. Column *Mainnet_Deploy* in Tabel 2 is the average gas consumption while deploying a smart contract to the Ethereum Mainnet. Column *oriPriv_Deploy* is the average gas consumption to deploy the original contracts to the private chain. Columns *op_Deploy* and *allOp_Deploy* demonstrate the average gas consumption for optimized contracts while deployed to the private chain. *op_Deploy* stands for optimization against dead nodes and opaque nodes. *allOp_Deploy* takes into account the part-opaque node besides the dead nodes and the opaque nodes, which should consume more gas than *op_Deploy* and *oriPriv_Deploy*. Columns Δop_Deploy and $\Delta allOp\_Deploy$ are the differences of gas consumption between respective optimized contracts and original contracts deployed to the private chain, i.e., difference between *op_Deploy* and *oriPriv_Deploy*. Thus, "+" means increase of the gas consumption, while "-" means decrease.

Figure 4 presents the results of Table 2 intuitively. From the graph, we can clearly observe that, the gas consumption for deployment of the optimized contracts is lower than that of the original contracts and the Ethereum Mainnet. In contrast, the consumption for allOp contracts are more than that of optimized contracts, and even more than that of the original contracts.

In Table 3, Column *Mainnet_txSum* is the average gas consumption for all transactions to a contract. Column *oriPriv_txSum* demonstrates the average total gas consumption for all these transactions replayed on the private chain. Columns *op_txSum* and *allOp_txSum* are the results of the optimized contracts. Columns Δop_txSum and $\Delta allOp\_txSum$ are the differences between optimized contracts and original contracts on the private chain, e.g., the difference between column *oriPriv_txSum* and column *op_txSum*. There are two cases in this table. Case1 stands for the scenario that account1 deploys the contracts and account2 invokes all the transactions, and case2 stands for that account1 deploys the contracts and account1 invokes all the transactions. This design originates from our observation that, the accounts for deployment and transaction invocation are significant for the gas consumption on the private chain, which is also expected, since the contracts may restrict the access rights for different accounts at deployment and transaction run-time, e.g., some functions can only be accessed by the owner.

Table 2: Average Gas Consumption of Optimized Contracts when Deployment

	Mainnet_Deploy	oriPriv_Deploy	op_Deploy	Δop_Deploy	allOp_Deploy	ΔallOp_Deploy
Deploy	1,462,809	1,271,657	1,246,082	-25,575	1,274,186	+2,529

Table 3: Average Gas Consumption of Optimized Contracts when Transaction

txSum	Mainnet_txSum	oriPriv_txSum	op_txSum	Δop_txSum	allOp_txSum	ΔallOp_txSum
Case1	111,455,394	63,949,658	63,806,611	-143,047	63,682,008	-267,650
Case2	111,455,394	80,583,595	79,691,229	-592,366	79,629,394	-954,201

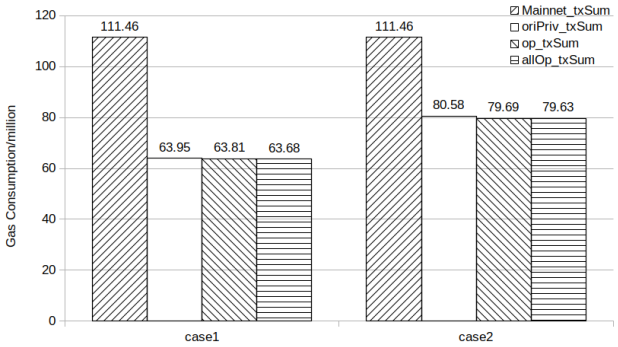


Figure 5: Gas Consumption for Transactions

```
1 contract GoMoney2 {
2     ...
3     function burn_address(address _target) public onlyOwner
          returns (bool){...
4     }
5     function mint(uint256 _amount) public beforeDeadline returns (
          bool){...
6     }
7 }
```

Figure 6: Access Control Example

Figure 5 illustrates the results of Table 3, from which we can get the following observations,

(1) Gas consumption on Ethereum Mainnet is larger than those on private chain, both for the deployment and the transactions. This is reasonable as the users' inputs may include some account-specific information in real Mainnet run-time, which our emulation cannot completely depict on the private chain. Thus, some transactions are reverted at the very beginning, which causes the big gap on the gas consumption between that on Mainnet and that on Private chain in both tables. As shown in Figure 6, the function of *mint* only works before a certain date, which is constrained by the modifier *beforeDeadline*, and thus the transactions to this function are all reverted on private chain.

(2) Gas consumption for optimization on dead nodes and opaque nodes only is smaller than original contracts but larger than all-nodes optimization in both cases on private chain, which is consistent with our expectation.

(3) Gas consumption in case2 is higher than that in case1. This is also due to the access problems, and also the reason of two cases design. We illustrate this problem with the example function *burn_address* in Figure 6. This function can only be invoked by the owner with the constraints from modifier *onlyOwner*. If it is invoked by other accounts, the optimized parts are never be reached, and thus the optimization is a vain. This may be one of the reasons that the gas reduction is not so impressive in this private run-time experiment.

To answer RQ2: The reduction of gas consumption is 25,575 units in deployment and the total gas reduction of all transactions for each contract can be as high as 954,201 units.

Overhead for Optimization. We acknowledge our optimization may increase the gas consumption on deployment in terms of *overhead*. If the optimization is only restricted on dead nodes and opaque nodes, there should be no overhead generated. If the part-opaque nodes taken into consideration, there are some instructions instrumented into the smart contracts, and that introduces overhead. As the column Δ*allOp_Deploy* shows in Table 2, the gas consumption for contract deployment grows by 2,529 averagely. This is caused by the code instrumentation when optimising the part-opaque nodes as explained in Example 2. The size of the instrumented opcodes depends on the commonly-used module, if it is too large, the gas consumption increases when the contract is deployed, although it saves more gas as the transaction volume becomes larger. In our experiment, the total gas saved of all transactions for a contract is 954,201, but also 2,529 gas is introduced when it is deployed to the blockchain if the users choose to optimize the part-opaque nodes.

To answer RQ3: There are overhead when we optimize the part-opaque nodes, and no overhead for optimization of dead nodes and opaque nodes only. The overhead is small compared to the overall saving.

Efficiency of sOptimize . The average analysis time is 261.5 seconds if taking the loops into consideration, otherwise, the average time is down to 153.5 seconds.

To answer RQ4: The optimization analysis can be achieved in a timely fashion.

Threats to Validity. There are several threats to validity in our experiments. First, *sOptimize* may miss some redundant codes in

the analysis. The reasons come from two aspects, the limitations of loop invariant learning and capabilities of constraint solver. If a valid invariant is not learned within a certain number of iterations (the default value is 20 in *sOptimize*), the node invariant will be true. The redundant codes within the loop will be missed. Another factor is the constraint solver, an opportunity for optimization is identified only when the solver returns a "SAT" result. Thus, if the constraint of a node is so complicated that an "UNKNOWN" result is returned, we soundly assume that this node is non-redundant. Second, our implementation currently has limitations, e.g., we only support contracts which contains only one `CODECOPY` in the bin part. The reason is that we must output the correct bytecode sequence after optimization, whereas in the case of instructions such as `CODECOPY`, the operands for the instruction are not easy to update simultaneously after optimization. Third, our experiments are conducted with a private test network (i.e., we compare the executions of contract before and after optimization). The contract might behaves differently on the private network from the Ethereum Mainnet (e.g., due to dependency on the Mainnet status). Experimenting directly with the Ethereum Mainnet however is not feasible due to the cost.

5 RELATED WORK

sOptimize is an optimization tool for Ethereum contracts based on smart contract analysis. Extensive work has been done for smart contracts analysis. For instance, symbolic execution engines like Oyente, sCompile, SolAnalyser [1, 7, 23] systematically identify vulnerabilities, like Transaction-Ordering Dependence, Timestamp Dependence, and Black-hole contract. Oyente [23] is the first tool to apply symbolic execution to find potential security vulnerabilities, but Oyente can only perform intra-procedural analysis. sCompile [7] introduced an approach to reveal "money-related" vulnerabilities in smart contract by identifying a small number of critical paths for user inspection. MAIAN [27] further mimicked inter-procedural invocations to find deeper vulnerabilities. ZEUS, solc-verify, VerX and VeriSmart [17, 20, 28, 31] introduce the policies, which allow the users to define their own specifications and properties including contract invariants, loop invariants, and function pre- and post-conditions etc. They provide automated verification against user specified properties. However, rare tools take into consideration the gas analysis.

There are other works focusing on gas-related vulnerabilities. Madmax [15] detects the gas-focused vulnerabilities in smart contracts by combining a control-flow-analysis-based decompiler and declarative program-structure queries. Chen et al. [10] addressed the DoS attacks by dynamically adjusting the costs of EVM operations according to the number of total executions. Albert et al. [3, 5] proposed methods and tools for automatically inferring gas upper bounds for functions to avoid out-of-gas vulnerabilities in a smart contract. GasFuzz [24] applies feedback-directed fuzz testing to automatically generate inputs which could lead to a high gas consumption of contract functions. Zhang et al. [34] propose a novel data structure called GEM^2-*Tree* to substitute Merkle hash tree in Ethereum to reduce gas cost. SmartCheck [32] detects 21 kinds of issues in smart contracts. Two of them are gas-efficiency related. The first one is the usage of *byte[]*, which is preferred to be replaced with *bytes* to reduce the gas cost. The second one is to detect loops that contain big number of steps. These works all try to identify vulnerabilities through abnormal gas consumption rather than optimization.

Currently there are few solidity bytecode optimization tools. Chen et al. proposed several approaches on detecting under-optimized contracts and developed a series of tools, like GASPER [9], GasReducer [11], and GasChecker [8]. The tool GASPER can automatically locate 3 gas-costly patterns by analyzing the bytecode of smart contracts, but GASPER can only identify several under-optimized bytecode patterns, and cannot optimize them. Based on GASPER, GasReducer [11] conducts in-depth investigation on under-optimized smart contracts' bytecode and identifies 24 anti-patterns which will then be replaced with efficient codes. However, the reduced gas cost of each pattern that GasReducer can recognize is very little. Compared to GASPER, GasChecker [8] detects more gas-inefficient code patterns and proposes a new approach to parallelize symbolic execution to make detecting patterns scalable which can handle millions of smart contracts by leveraging cloud computing platform whereas GASPER uses sequential symbolic execution. However, to prevent path explosion, GasChecker unfolds the loops up to four that will result in false positives in detecting these patterns. Our tool sOptimize can precisely locate and remove the redundant code in the loop by invariant generation.

6 CONCLUSION

We leverage the static analysis techniques (i.e., lazy annotation and loop invariant generation techniques) to identify 3 kinds of code blocks, i.e., *dead node*, *opaque node*, and *part-opaque node* and further remove the identified redundant code blocks to optimize the contract. An automatic toolkit *sOptimize* is developed, and applied to 1,152 test subjects, as many as 499 contracts are optimized. With the comparison experiment on 212 contracts, the gas reduced for deployment is around 2.0% and the average gas cost of all transactions for a contract is around 1.2% at most.

REFERENCES

[1] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 482–489.

[2] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. Analyzing Smart Contracts: From EVM to a sound Control-Flow Graph. *arXiv preprint arXiv:2004.14437* (2020).

[3] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. Gasol: Gas analysis and optimization for ethereum smart contracts. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 118–125.

[4] Elvira Albert, Pablo Gordillo, Albert Rubio, and Maria A Schett. 2020. Synthesis of Super-Optimized Smart Contracts Using Max-SMT. In *International Conference on Computer Aided Verification*. Springer, 177–200.

[5] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2019. Running on fumes. In *International Conference on Verification and Evaluation of Computer and Communication Systems*. Springer, 63–78.

[6] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Issue 3. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. 2019. sCompile: Critical Path Identification and Analysis for Smart Contracts. In *Proceedings of the 21st International Conference on Formal Engineering Methods, ICFEM 2019*. 286–304.

[8] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang. 2020. GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts. *IEEE Transactions on Emerging Topics in Computing* (2020), 1–1.

[9] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference*

on Software Analysis, Evolution and Reengineering (SANER). IEEE, 442–446.

[10] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. 2017. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *International Conference on Information Security Practice and Experience*. Springer, 3–24.

[11] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 81–84.

[12] Corinna Cortes and Vladimir Vapnik. 1995. Support-Vector Networks. In *Machine Learning*. 273–297.

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, 337–340.

[14] Etherscan. [n.d.]. Ethereum (ETH) Blockchain Data. https://etherscan.io/charts#blockchainData. (Accessed on 06/27/2020).

[15] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[16] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.

[17] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for Solidity smart contracts. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 161–179.

[18] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

[19] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2018. Executable operational semantics of Solidity. *arXiv preprint arXiv:1804.01295* (2018).

[20] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *NDSS*.

[21] Masanari Kondo, Gustavo A Oliva, Zhen Ming Jack Jiang, Ahmed E Hassan, and Osamu Mizuno. [n.d.]. Code Cloning in Smart Contracts: A Case Study on Verified Contracts from the Ethereum Blockchain Platform. ([n. d.]).

[22] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation and refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 782–792.

[23] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[24] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. 2019. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *arXiv preprint arXiv:1910.02945* (2019).

[25] Kenneth L. McMillan. 2010. Lazy Annotation for Program Testing and Verification. In *Proceedings of the 22Nd International Conference on Computer Aided Verification (CAV'10)*. 104–118.

[26] Julian Nagele and Maria A Schett. 2020. Blockchain Superoptimizer. *arXiv preprint arXiv:2005.05912* (2020).

[27] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663.

[28] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP*. 18–20.

[29] J.R. Quinlan. 2017. C5.0: An Informal Tutorial. http://www.rulequest.com/see5-unix.html.

[30] J. R. Quinlan. 1987. Simplifying Decision Trees. *Int. J. Man-Mach. Stud.* 27, 3 (1987), 221–234.

[31] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2019. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. *arXiv preprint arXiv:1908.11227* (2019).

[32] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[33] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[34] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. 2019. GEMˆ 2-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain. In *2019 IEEE 35th international conference on data engineering (ICDE)*. IEEE, 842–853.