Verification Assisted Gas Reduction for Smart Contracts

Bo Gao
Singapore University of Technology and Design, Singapore
Email: bo_gao@mymail.sutd.edu.sg

Siyuan Shen
State Key Laboratory For Novel Software Technology, Nanjing University, China

Ling Shi
Singapore Management University
Singapore

Jiaying Li
Singapore Management University
Singapore

Jun Sun
Singapore Management University
Singapore

Lei Bu
State Key Laboratory For Novel Software Technology, Nanjing University, China
Email: bulei@nju.edu.cn

***Abstract*—Smart contracts are computerized transaction protocols built on top of blockchain networks. Users are charged with fees, a.k.a. gas in Ethereum, 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 codes for security checks. The trouble is that those codes consume gas every time they are executed. Many of the inserted codes are however redundant. In this work, we present *sOptimize*, a tool that optimizes smart contract gas consumption automatically without compromising functionality or security. sOptimize works on smart contract bytecode, statically identifies 3 kinds of code patterns, and further removes them through verification-assisted techniques. The resulting code is guaranteed to be equivalent to the original one and can be directly deployed on blockchain. We evaluate sOptimize on a collection of 1,152 real-world smart contracts and show that it optimizes 43% of them, and the reduction on gas consumption is about 2.0% while in deployment and 1.2% in transactions, the amount can be as high as 954,201 gas units per contract.**

***Index Terms*—smart contract, optimization, gas reduction**

I. 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 [1], 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 gas, which is determined by the market (i.e., the miners). The average price in year 2020 is around 60 gwei/unit (i.e., 1 gwei = 10^{-9} Ether). Gas_amount is the number of gas units consumed for any computation or storage usage. It can be classified as $amount$ that is consumed while in deployment and $amount$ while in 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 [1].

Smart contracts are getting more and more popular in recent years, i.e., the volume 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 [2]. Furthermore, after several high-profile contracts were attacked, security is more relevant a concern for contract developers than ever. A common practice for preventing security problems is to adopt standardized 'secure' libraries. Kondo *et al.* [3] 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.

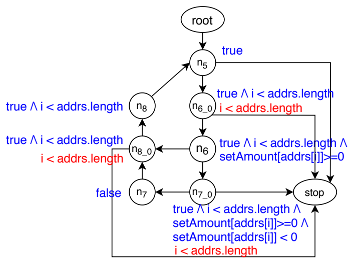
These standardized secure libraries introduce run-time security checking codes. For instance, once the `SafeMath.sol` library is adopted, run-time checks for possible overflow are introduced for every arithmetic operation in the contract. We foresee that such a practice will become increasingly popular (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 can we reduce such gas consumption without sacrificing the security?

Studies related to gas reduction in smart contracts have only recently attracted some attention. In [4], Chen *et al.* proposed GasReducer which identifies multiple anti-patterns from the execution traces of smart contracts and replaces

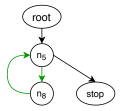
This work was partly supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001).

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

(a) Unreachable Branches in Loop



(b) Labeled CFG



(c) Optimized CFG

Fig. 1: Optimization for Loop Contract

these patterns with optimized code to reduce gas consumption. GASPER [5] applied symbolic execution to locate patterns which often consume excessive gas. Later, Gasol [6] proposed a cost model which allows users to infer the gas consumption for transactions and ensures the contract 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* [7] 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.* [8] 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 security concerns). *sOptimize* applies static analysis techniques (i.e., lazy annotation [9]) and loop invariant generation techniques [10] 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*, *redundant node*, and *partial-redundant node* (refer to Section III-C for detailed definitions).

To summarize, this paper makes the following contributions:

- We apply the verification techniques of loop invariant learning and invariant inference to optimize 3 kinds of nodes for smart contracts.
- We develop an end-to-end tool *sOptimize* to reduce the gas cost for smart contracts.
- We evaluate the effectiveness of *sOptimize* with a set of 1,152 smart contracts on private chain, and find that on average 25,575 units (2%) of gas are reduced during contract deployment and 954,201 units (1.2%) of gas can be saved during transactions for a contract at most.

The rest of the paper is organized as follows. In Section II, we illustrate how *sOptimize* works through two simple examples. In Section III, we present the details of our approach. In Section IV, we discuss the evaluation results and Section V reviews related works and lastly we conclude our work in Section VI.

II. Overview

Given a smart contract, the goal of *sOptimize* is to optimize its gas usage through detecting and eliminating redundant codes, i.e., dead nodes, redundant nodes or partial-redundant nodes on the premise of security. In this section, we illustrate how *sOptimize* works through two examples. They are both excerpted from real-world contracts but modified for illustration.

Example 1: In this 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 Ethereum) to the contract owner. Afterwards, the contract owner allocates `amt` 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 unreachable branches which are never executed and the condition which is an unnecessary check caused by a mistake at line 6.

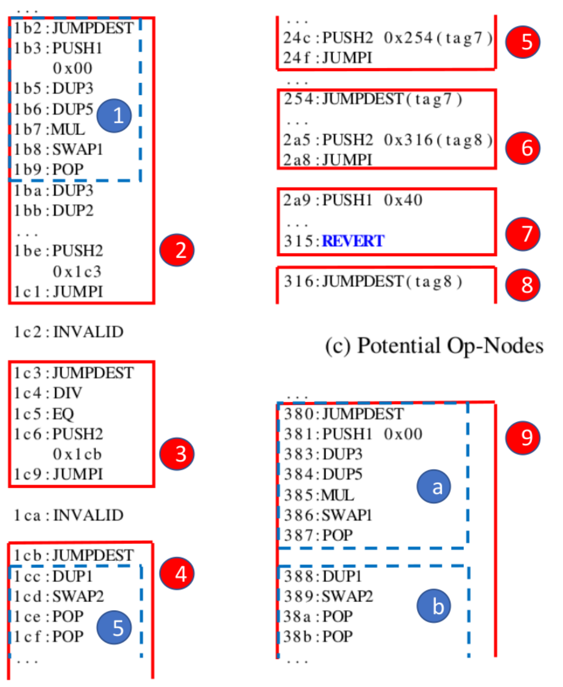
sOptimize first constructs the control flow graph (CFG) of the `setDistributeToken` function as shown in Figure 1b. In this figure, node *root* and node *stop* represent the entry and exit of the function respectively, and other nodes represent the corresponding statements in the contract. The predicates (in blue) associated with the nodes are node invariants, and the predicates in red are 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, 7 and 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 $setAmt[addrs[i]] < 0$ 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 invariant for each node using a combination of program inference, lazy annotation and loop invariant learning techniques. Initially, the invariant for each node is $true$. *sOptimize* iteratively and monotonically

[1]contract address: 0x2deF52220E91EB42B2CaF8005F7f671dC692Bf89

```
 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



```
...
1b2:JUMPDEST
1b3:PUSH1
      0x00
1b5:DUP3
1b6:DUP5
1b7:MUL
1b8:SWAP1
1b9:POP
1ba:DUP3
1bb:DUP2
...
1be:PUSH2
      0x1c3
1c1:JUMPI

1c2:INVALID

1c3:JUMPDEST
1c4:DIV
1c5:EQ
1c6:PUSH2
      0x1cb
1c9:JUMPI

1ca:INVALID

1cb:JUMPDEST
1cc:DUP1
1cd:SWAP2
1ce:POP
1cf:POP
...
```

```
...
24c:PUSH2  0x254(tag7)
24f:JUMPI
...
254:JUMPDEST(tag7)
...
2a5:PUSH2  0x316(tag8)
2a8:JUMPI

2a9:PUSH1  0x40
...
315:REVERT

316:JUMPDEST(tag8)
```

```
...
380:JUMPDEST
381:PUSH1  0x00
383:DUP3
384:DUP5
385:MUL
386:SWAP1
387:POP
388:DUP1
389:SWAP2
38a:POP
38b:POP
...
```

(b) Part-opaque Node (c) Potential Op-Nodes (d) Op-part-opaque Node

Fig. 2: Optimization for Common Contract

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, it 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 amt), 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 $setAmt$ is non-negative, since the element is defined as *uint* at line 2. Thus the invariant of node n_6 is strengthened as $true \wedge i < addrs.length \wedge setAmt[addrs[i]] >= 0$, node n_{7_0} is $true \wedge i < addrs.length \wedge setAmt[addrs[i]] >= 0 \wedge setAmt[addrs[i]] < 0$ (equivalent to $false$), and node n_7 is $false$. Note 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 potential optimization nodes based on the CFG. 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 redundant nodes. They are thus removed and a new edge is generated to link node n_5 with node n_8 directly as shown in Figure 1c, i.e., codes at lines 6 and 7 are removed after optimization.

Example 2: In this example, we show how different nodes are optimized at bytecode level. As shown in Figure 2a, contract `ethBank`[2] receives deposit 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 `onlyOwner` at line 17 restricts that only the user can 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 nodes for line 21 after compilation are opcode blocks labelled with 6, 7 and 8 shown at Figure 2c. The node labelled with 6 is identified as a redundant node; and the node 7 is an dead node. Since the modifier `onlyOwner` functioning at line 20 allows only the owner to proceed, the *require* statement at line 21 must be satisfied. The corresponding checking node labelled 6 always jumps to node 8. As a result, node 7 is never reached. Thus, we can redirect the edge to line 316 directly from line 24c and remove all the opcodes in node 6 and 7. 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

[2]contract address: 0xf3fa62dd25504a7b05300a1ddd56a22a100fd4df

for run-time code deposit), and 269 units of gas for each transaction afterwards.

This example also contains a partial-redundant node. The assert statement at line 6 from the `SafeMath` library is redundant when it is invoked from the fallback function at line 19 in Figure 2a. The corresponding bytecode sequence is at line 1b2 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 22. *sOptimize* generates a new copy of function `mul` represented by node 9 in Figure 2d, this code snippet is appended in the end of the optimized bytecode. Part a and b are optimized from part 1 and part 5. All the other opcodes between line 1ba and line 1cb will not be executed in such transactions. 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 gas units during deployment and at the same time reduce 46 units of gas for each transaction subsequently. It means this optimization is profitable if the transaction volume is larger than 250.

III. Our Approach

In this section, we present our approach 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 (line 1). Then, we update node invariants in CFG_{f_i} with function $updateInv(CFG_{f\_i})$ at line 3 and initiate the optimized CFG OP_{f_i} with the updated CFG_{f_i} at line 4. $set(N)$ is the set of all the nodes in the CFG, which is defined in Definition 1. Next, we examine every node in the CFG to systematically identify and optimize dead nodes and redundant nodes including partial-redundant nodes from line 5 to line 10. Lastly, we reorganize the bytecode sequences of the optimized CFG OP_{f_i} to output the bytecode sequence through function $reOrganizeBytecode(OP_{f\_i})$ at line 12. In the following, we present details of the main steps.

A. 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. We omit the definitions for the opcodes, readers can refer to Ethereum yellow paper [1] for further details. A function of a smart contract is composed of a sequence of opcodes. Typically the opcodes are organized into basic blocks, i.e., a sequence of opcodes which do not contain a branching opcode except the last one or a starting opcode except the first one.

Definition 1: 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 representing basic blocks 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.

Algorithm 1: Overall Optimization Algorithm

```
1  $\{CFG_{f\_i}\} \leftarrow CFG\_construct(C)$;
2  for $CFG_{f\_i}$ do
3      $CFG_{f\_i} \leftarrow updateInv(CFG_{f\_i})$;
4      $OP_{f\_i} \leftarrow CFG_{f\_i}$;
5      for $node\ n \in set(N)$ do
6          if $\mathcal{I}(n) = false$ then
7              $OP_{f\_i} \leftarrow$
                 $rmDeadNode(CFG_{f\_i}, OP_{f\_i}, n)$;
8          end
9          $OP_{f\_i} \leftarrow opONode(OP_{f\_i}, n)$;
10     end
11 end
12 $reOrganizeBytecode(\{OP_{f\_i}\})$;
```

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. Then, 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 due to different control flows, like the nodes in the library in Figure 2a. Such nodes are duplicated in the CFG construction. Readers are referred to [11], [12] for details on how the CFG is constructed.

B. Invariant Generation

We strengthen the invariant for each node in the CFG at this step. We first define what is an invariant based on the semantics of the function.

Definition 2 (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 $(S, init, \rightarrow_s, \mathcal{I})$, 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); $\rightarrow_s \subseteq S \times S$ is the transition relation conforming to the symbolic semantic rules.

A few execution rules are shown in Figure 3 to make this paper self-contained. Readers can refer to [13] for the other rules. 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'. Otherwise, pc is moved to the succeeding opcode, which is $pc + 1$.

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

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

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

Fig. 3: Instruction Execution rules

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 $s_i \rightarrow_s s_{i+1}$ 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 3 (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. $\pi(last(tr)) = n$.

Note that $v \models \phi$ means ϕ is satisfied by the variable valuation 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 maps the state to the corresponding node n.

Definition 4 (Strongest Postcondition): Given an opcode op and a precondition ϕ, the strongest postcondition $sp(c, \phi)$ is defined as:

$$\begin{aligned}
&sp(SSTORE(p, v), \phi) = \\
&\qquad\qquad \exists y,\ \phi[y/storage[p]] \wedge storage[p] = v \\
&sp(op, \phi) = \phi \wedge b \qquad \text{if } op = JUMPI(b) \\
&sp(op, \phi) = \phi \qquad\quad\ \ \text{if } op = JUMP \text{ or } SLOAD(x)
\end{aligned}$$

In the above definition, the fresh variable y represents the previous values of $storage[p]$ in the strongest postcondition for command $SSTORE$. For the branching command $JUMPI$, the strongest condition is the conjunction of ϕ and condition b. Since there is no condition introduced for $JUMP$ and $SLOAD$, the strongest postcondition keeps the same. Worthing to say, $SLOAD$ may introduce new predicate when the position x is first visited, however, the content must be in other forms integrated into the strongest postcondition, like assigning to other variables or acting as a part of the branch condition. Thus, it keeps the same here. All the values in the storage including global storage, memory storage and stack storage [14] are in the form of static single assignment. Thus, they are all manifest in the states which reach the fix point finally.

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, the invariant of node n is monotonically strengthened by the conjunction of $\mathcal{I}(n)$ and Ψ at line 5.

Algorithm 2: $inferI(F, n)$

```
1 $\Psi \leftarrow false$;
2 for $(m, n) \in E$ do
3     $\Psi \leftarrow \Psi \vee \phi(m)$;
4 end
5 $\mathcal{I}(n) \leftarrow \mathcal{I}(n) \wedge \Psi$
```

Algorithm 3: $opNodes(OP(F), n)$

```
1 if $n \in \{b?n_1 : n_2\}$ then
2     if $\mathcal{I}(n) \Rightarrow b$ then
3         $OP(F) \leftarrow lkNodes(OP(F), n, n_1)$;
4     else if $\mathcal{I}(n) \Rightarrow \neg b$ then
5         $OP(F) \leftarrow lkNodes(OP(F), n, n_2)$;
6     end
7 end
```

Algorithm 4: $updateInv(CFG(F))$

```
1 $\mathcal{I} \leftarrow init(true)$;
2 $\mathcal{I}' \leftarrow \emptyset$;
3 while $\mathcal{I}' \neq \mathcal{I}$ do
4     $\mathcal{I}' \leftarrow \mathcal{I}$;
5     for $n \in N$ do
6         if $n$ is loop head then
7             $\mathcal{I}(n) \leftarrow learnI(CFG(F), n)$;
8         else
9             $\mathcal{I}(n) \leftarrow inferI(CFG(F), n)$;
10        end
11    end
12 end
```

Then, how do we generate non-trivial invariants for each node? As shown in Algorithm 4, we adopt two ways to generate the invariants depending on whether a node is a head-node for a loop or not. We distinguish head nodes of certain loops (i.e., a node representing the start of a loop statement) and apply a different approach to infer invariants for such nodes. If the node is not the head of a loop, it is inferred by function $inferI(F, n)$. If the node is the head of a loop, we generate the loop invariant through a "guess and check" approach, which is adopted from [10]. Intuitively, the loop invariant learning function $learnI(F, n)$ 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 [15] and the decision tree [16]). 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 sample set to learn a new invariant. Once validated, the candidate invariant is returned as the output of function $learnI(F, n)$. As the example shown in Figure 1a, *sOptimize* learns loop invariant $true$ against the compiler-inserted assertion ($i < addrs.length$), and it is successfully validated by the validator. Since learning loop invariant is not the main contribution of this work, we refer interested readers to [10] for further details.

C. Optimization

With the definition above, we present how to optimize the contracts on dead node, redundant node and partial-redundant node.

1) Dead node.: Dead code refers to code which can never be executed at run-time [17]. In our labelled CFG, a node is dead if the node invariant $\mathcal{I}(n)$ is evaluated to be $false$, that is all symbolic traces reaching the node are infeasible.

Example 3: As shown in Figure 2c, the invariant of node 7 which corresponds to *require* statement at line 21 in Figure 2a, is $false$, thus it is a dead node. We can remove this node from the CFG directly without affecting any feasible traces.

2) Redundant Node.: A computation is redundant if it has been computed previously and its result is guaranteed to be available at that point [18]. Redundant node is a kind of redundant code. Intuitively, a node is redundant if its invariant can successfully imply its branch condition or the negation of the branch condition in the labelled CFG.

Example 4: For instance, in Figure 2c, the node 6 is a redundant 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 always succeeds, this node always goes to node 8. After invoking function $lkNodes$, the redundant node is removed from the CFG and the tag at line 24c is updated to tag8 from tag7 to form the new edge.

3) Partial-redundant Node.: An expression is partially redundant at program point p if it is redundant along some, but not all, paths that reach p [19]. Identifying partial-redundant node is simple, since we have marked the node as duplicate when constructing CFG if a node is linked by different control flows. Thus, if a node is redundant and also marked as duplicate, it must be a partial-redundant node.

Example 5: As the example shown in Figure 2b, *sOptimize* discovers node 2 and node 3 are both partial-redundant nodes, which maps to the assert statement at line 6 in function *mul* invoked by statement at line 19 in Figure 2a. Since the assertion never fails in such a case because the variable $ethWei$ is a constant, the node always goes from node 2 through node 3 to node 4. Function $lkNodes$ modifies nodes 2, 3 and 4 and forms node 9 as shown in Figure 2d, which only keeps the necessary opcodes part 1 and part 5 (in blue box) in Figure 2b. 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.

We illustrate the optimization of redundant nodes and partial-redundant nodes in Algorithm 3. If node n is a branching node, *sOptimize* will evaluate whether the node invariant $I(n)$ can imply the branch condition. If the implication succeeds, that means the edge always starts from node n and stops at node $n1$, *sOptimize* invokes function $lkNodes$ to update the target of the parent node of n to link it to node $n1$ directly, and remove the current node n on the CFG. Otherwise, *sOptimize* will further evaluate whether the node invariant $I(n)$ can imply the negation of the branch condition and links the parent node of n to node $n2$ if the implication succeeds.

D. Bytecode Reorganization

To make the optimized contract work as a valid EVM contract, we need to reorganize the control flow in the new bytecode sequence. We have marked the opcode which determines the control flow and the corresponding tag sequence for each node when constructing the CFG.

Example 6: As shown in Figure 2c, the target for line 24c is 0x254, which is labelled with tag7, we also link line 2a5 with 0x316 by tag8 in the same way before optimization. After removing the redundant codes, we update the tag at line 24c with tag8, and further recalculate the target addresses for all the *PUSH* opcodes. Finally, *sOptimize* outputs the reorganized bytecode.

E. Soundness of Overall Algorithm

The soundness of overall algorithm (i.e., 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 3. In the latter case, the correctness of the inferred invariant generated by $learnI$ 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.

IV. 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 [20] and C5.0 [21]. Z3 SMT solver is adopted to check the satisfiability of constraints in the invariant candidate validation phase and the redundant nodes identification phase.

A. 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:** What are the overhead in terms of gas and time by *sOptimize*?

To the best of our knowledge, there are no off-the-shelf tools which aim at reducing gas consumption on 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[3]), others are optimization tools for instructions' sequence (e.g., *ebso* and *syrup*), which concentrate on the optimization within a block, and moreover, part of the tool (which converts the target bytecode blocks to SFS [8], an intermediate form) is not available currently, which prevents us from accessing the tool. That's the reason why there is no comparison design against other tools in above RQs.

In this evaluation, we collected 8,140 verified Solidity contracts with open-source licenses on Etherscan[4], a leading BlockChain Explorer for Ethereum. Since the total gas consumption is proportional to the transaction volume, we select 1,152 contracts whose transactions are more than 100 to evaluate the performance of *sOptimize*. The highest transaction volume is 999,366, and the total transactions for all selected contracts are 9.4 million units of gas 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.

[3]There are options for optimization on storage related operations, but the installation package provided is broken, we did not get the feedback.

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

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

sOptimize identified 499 contracts that can be optimized from 1,152 (43.3%) contracts in wall time. The result is shown in Table I, column *RT_Size* shows the average size of run-time bytecode, columns *D_Node* and *O_Node* are the size of dead node and redundant 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 [1]. 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 gas units when the contract is deployed to the blockchain. The gas reduction on 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 are potential to be optimized and the contract size can be reduced 2.0% in terms of bytes averagely, which can save 29,900 units of gas while in deployment and 328 units of gas for transactions relevant with these nodes afterwards in the run-time environment.

2) Effectiveness of sOptimize: We intend to study the effectiveness of *sOptimize* through comparing the total gas consumption while in deployment and 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 on the private chains with the same input from Ethereum Mainnet. 212 contracts are deployed to demonstrate the effectiveness of sOptimize. Those containing a special opcode CODECOPY in run-time bytecode is omitted at the time, as non-trivial engineering work is required for complex adjustment on the optimized bytecode sequence. We will improve it in the future work.

The results are shown in Table II and Table III. Column *Mainnet_Deploy* in Tabel II is the average gas consumption while deploying a smart contract to the Ethereum Mainnet. Column *oriPriv_Deploy* is to deploy the original contracts to the private chain. *op_Deploy* and *allOp_Deploy* demonstrate the average gas consumption for optimized contracts while deployed to the private chain. *op_Deploy* stands for optimiza-

TABLE I: 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

TABLE II: Average Gas Consumption of Benchmark Contracts for 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 III: Average Gas Consumption of Benchmark Contracts for 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

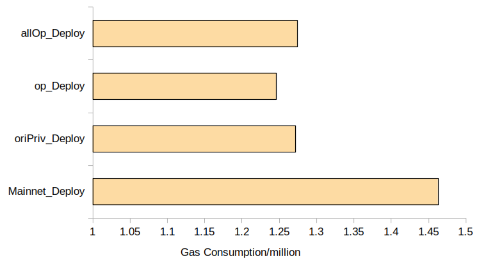
tion against dead nodes and redundant nodes. *allOp_Deploy* takes into account the partial-redundant nodes besides the previous two nodes, which should consume more gas than *op_Deploy* and *oriPriv_Deploy*. Columns Δ*op_Deploy* and Δ*allOp_Deploy* are the differences of gas consumption between optimized contracts and original contracts when deployed to the private chain, i.e., difference between *op_Deploy*, *allOp_Deploy* and *oriPriv_Deploy*. Thus, "+" means increase of the gas consumption, while "-" means decrease. Figure 4a presents the results of Table II 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 gas consumption of deployment for allOp contracts are the highest among all situations.

In Table III, Column *Mainnet_txSum* is the average total gas consumption for a contract on Ethereum mainnet. Column *oriPriv_txSum* demonstrates the average total gas consumption while all transactions are replayed on the private chain. Columns *op_txSum* and *allOp_txSum* are the gas consumption of the optimized contracts. Δ*op_txSum* and Δ*allOp_txSum* are the difference between *op_txSum*, *allOp_txSum* and *oriPriv_txSum*. There are two cases in this table. Case1 stands for the scenario that account1 deploys the contracts and account2 invokes 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. Figure 4b illustrates the results of Table III, from which we can get the following observations,

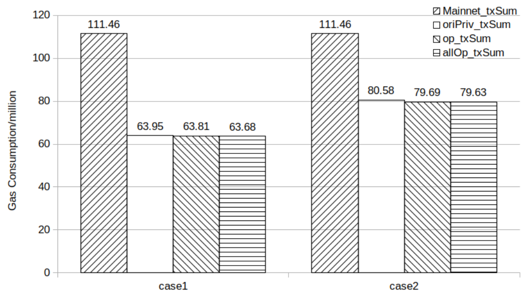
1) Gas consumption for deployment and transactions on Ethereum Mainnet is larger than that on private chain. This is reasonable as the users' inputs may include account-specific information in real Mainnet run-time, which our emulation cannot completely depict on the private chain. Thus, some transactions are reverted, which causes the big gap on the gas consumption between the Mainnet and the Private chain in both tables. As shown in Figure 5, 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 reduction for all-nodes optimization (column Δ*allOp_txSum*) is larger than optimization on dead nodes and redundant nodes only in both cases in Table III, which is exactly consistent with our expectation.
3) Gas consumption in row Case2 is higher than that in Case1. This is also due to the access problems, and also the reason of two cases design. As illustrated by function *burn_address* in Figure 5, this function can only be invoked by the owner due to the constraints of modifier *onlyOwner*. If it is invoked by other accounts, the optimized parts are never executed. 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 can be as high as 25,575 units (2.1%) in deployment and 954,201 units (1.2%) in transactions . We acknowledge that our optimization may increase the gas consumption on deployment in terms of *overhead*. Specifically, the overhead is generated while the partial-redundant nodes are taken into consideration, as some instructions are instrumented into the smart contracts. However, if the optimization is only restricted on dead nodes and redundant nodes, there is no overhead generated. As the column Δ*allOp_Deploy* shows in Table II, the gas consumption for contract deployment grows by 2,529 units averagely 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 can be 954,201 units totally, but also 2,529 gas units is introduced while it is deployed. We expect a better performance in the production run-time.

To answer RQ3: We answer this RQ from two aspects. One the one hand, the overhead is generated in terms of gas



(a) Gas Consumption for Deployment



(b) Gas Consumption for Transactions

Fig. 4: Gas Consumption

```
function mint(uint256 _amount) public beforeDeadline
     returns (bool){...}
function burn_address(address _target) public onlyOwner
     returns (bool){...}
```

Fig. 5: Access Control Example

consumption. As explained in RQ2, there are about 2,529 (0.2%) more gas units consumed while in deployment if the partial-redundant nodes are taken into consideration. However, the overhead is relatively small comparing to the overall saving, 954,201 (0.3%) gas units, in the run-time transactions. One the other hand, about 261.5 seconds are consumed while in the analysis averagely for a contract. This is reasonable regarding the enquiries for solving and invariant learning, which is essential for the optimization on the premise of correctness of the contract.

3) Threats to Validity: There are several threats to validity in our evaluation. 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 10 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 experiments are conducted on a private test network (i.e., we compare the executions of contract before and after optimization) as experimenting directly with the Ethereum Mainnet is not feasible due to the cost. Thus, the contract might behave differently on the private network from the Ethereum Mainnet (e.g., due to dependency on the Mainnet status). However, the results from the private network provide a lower bound for our optimization because many optimization-related transactions may be stopped from executing which cut down the gas reduction of our tool.

V. Related Work

sOptimize is an optimization tool for Ethereum contracts based on smart contract analysis. Thus, we mainly concentrate on two aspects of smart contracts relevant in this section, i.e., existing works on analysis and those on optimization.

Extensive work has been done for smart contracts analysis. For instance, symbolic execution engines like Oyente, sCompile, SolAnalyser [12], [22], [23] systematically identify vulnerabilities, like Transaction-Ordering Dependence, Timestamp Dependence, and Black-hole contracts. Oyente [22] is the first tool to apply symbolic execution to find potential security vulnerabilities, but Oyente can only perform intra-procedural analysis. sCompile [12] introduced an approach to reveal "money-related" vulnerabilities in smart contract by identifying a small number of critical paths for user inspection. MAIAN [24] further mimicked inter-procedural invocations to find deeper vulnerabilities. ZEUS, solc-verify, VerX and VeriSmart [25]–[28] 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 [29] detected the gas-focused vulnerabilities in smart contracts by combining a control-flow-analysis-based decompiler and declarative program-structure queries. Chen et al. [30] addressed the DoS attacks by dynamically adjusting the costs of EVM operations according to the executions. Albert et al. [6], [31] proposed methods and tools for automatically inferring gas upper bounds for functions to avoid out-of-gas vulnerabilities in smart contracts. GasFuzz [32] applied feedback-directed fuzz testing to generate inputs which could lead to a high gas consumption by contract functions. SmartCheck [33] detected 21 kinds of issues in smart contracts, two of which are gas-efficiency related. The first one is to replace the usage of *byte[]* to *bytes* to reduce the gas cost. The second one is to detect loops that contain big number steps. These works all try to identify vulnerabilities through abnormal gas consumption rather than optimization.

Currently there are few optimization tools on smart contracts. Chen et al. proposed several approaches on detecting under-optimized contracts and developed a series of tools, like GASPER [5], GasReducer [4], and GasChecker [34]. 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 [4] 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 and the patterns identified are heavily dependent on experience. Compared to GASPER, GasChecker [34] 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. Such problems are avoided by *sOptimize* by leveraging the technique of invariant generation for loops and further correctly removing the redundant codes.

VI. 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*, *redundant node*, and *partial-redundant node* and further remove the identified code blocks to optimize the contracts. 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 25,575 gas units (2.0%) and the average gas consumption reduced of all transactions for a contract is around 954,201 gas units (1.2%).

References

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[2] Etherscan, "Ethereum (eth) blockchain data," https://etherscan.io/charts#blockchainData, (Accessed on 06/27/2020).

[3] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4617–4675, 2020.

[4] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "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, 2018, pp. 81–84.

[5] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[6] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, pp. 118–125.

[7] J. Nagele and M. A. Schett, "Blockchain superoptimizer," *arXiv preprint arXiv:2005.05912*, 2020.

[8] E. Albert, P. Gordillo, A. Rubio, and M. A. Schett, "Synthesis of super-optimized smart contracts using max-smt," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 177–200.

[9] K. L. McMillan, "Lazy annotation for program testing and verification," in *Proceedings of the 22Nd International Conference on Computer Aided Verification*, ser. CAV'10, 2010, pp. 104–118.

[10] J. Li, J. Sun, L. Li, Q. L. Le, and S.-W. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 782–792.

[11] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Analyzing smart contracts: From evm to a sound control-flow graph," *arXiv preprint arXiv:2004.14437*, 2020.

[12] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical Path Identification and Analysis for Smart Contracts," in *Proceedings of the 21st International Conference on Formal Engineering Methods, ICFEM 2019*, 2019, pp. 286–304.

[13] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[14] Solidity. Solidity documentation. [Online]. Available: https://docs.soliditylang.org/en/latest/

[15] C. Cortes and V. Vapnik, "Support-vector networks," in *Machine Learning*, 1995, pp. 273–297.

[16] J. R. Quinlan, "Simplifying decision trees," *Int. J. Man-Mach. Stud.*, vol. 27, no. 3, p. 221–234, 1987.

[17] L. A. Johnson *et al.*, "Do-178b: Software considerations in airborne systems and equipment certification," *Crosstalk, October*, vol. 199, pp. 11–20, 1998.

[18] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Transactions on Programming languages and Systems (TOPLAS)*, vol. 22, no. 2, pp. 378–415, 2000.

[19] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 159–170, 1994.

[20] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[21] J. Quinlan, "C5.0: An informal tutorial," http://www.rulequest.com/see5-unix.html, 2017.

[22] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[23] S. Akca, A. Rajan, and C. Peng, "Solanalyser: A framework for analysing and testing smart contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 482–489.

[24] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[25] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *NDSS*, 2018.

[26] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019, pp. 161–179.

[27] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts," in *2020 IEEE Symposium on Security and Privacy, SP*, 2020, pp. 18–20.

[28] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," *arXiv preprint arXiv:1908.11227*, 2019.

[29] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[30] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks," in *International Conference on Information Security Practice and Experience*. Springer, 2017, pp. 3–24.

[31] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "Running on fumes," in *International Conference on Verification and Evaluation of Computer and Communication Systems*. Springer, 2019, pp. 63–78.

[32] F. Ma, Y. Fu, M. Ren, W. Sun, Z. Liu, Y. Jiang, J. Sun, and J. Sun, "Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability," *arXiv preprint arXiv:1910.02945*, 2019.

[33] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[34] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2020.