V-Gas: Generating High Gas Consumption Inputs to Avoid Out-of-Gas Vulnerability

FUCHEN MA, Tsinghua University, China
MENG REN, Tsinghua University, China
FU YING, Tsinghua University, China
WANTING SUN, Baidu Online Network Technology Co.,Ltd, China
HOUBING SONG, Embry-Riddle Aeronautical University, USA
HEYUAN SHI, Big Data Institute, Central South University, China
YU JIANG, Tsinghua University, China
HUIZHONG LI, WeBank, China

Out-of-gas errors occur when smart contract programs are provided with inputs that cause excessive gas consumption and which will be easily exploited to perform DoS attacks. Various approaches have been proposed to estimate the gas limit of a function in smart contracts to avoid such error. However, underestimation often occurs when the contract is complex In this work, we propose V-Gas, which automatically generates inputs that maximize the gas cost and reduce underestimation. V-Gas is designed based on static analysis and feedback-directed mutational fuzz testing. First, V-Gas builds the gas weighted control flow graph (WCFG) of functions in smart contracts. Then, V-Gas develops gas consumption guided selection and mutation strategies to generate the input that maximize the gas consumption.

For evaluation, we implement V-Gas based on js-evm, a widely used ethereum virtual machine written in Javascript, and conduct experiments on 736 real-world transactions recorded on Ethereum. 44.02% of the transactions would have out-of-gas errors based on the estimation results given by solc, means that the recorded real gas consumption for those transactions is larger than the gas limit estimated by solc. In comparison V-Gas could reduce the under estimation ratio to 13.86%. In order to evaluate the performance of feedback-directed engine in V-Gas, we implemented other directed fuzzing engines and compare their performance with V-Gas's. The results showed that V-Gas generates the same or higher gas estimation value on 97.8% of the transactions with less time, usually within 5 minutes. Furthermore, V-Gas has exposed 25 previously unknown out-of-gas vulnerabilities in widely-used smart contracts, 6 of which have been assigned unique CVE identifiers in the US National Vulnerability Database.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**.

Additional Key Words and Phrases: Smart contracts, Ethereum, Fuzzing, Gas Estimation.

Authors' addresses: Fuchen Ma, Tsinghua University, Haidian District, Beijing Shi, Chinamafc19@mails.tsinghua.edu.cn; Meng Ren, Tsinghua University, Haidian District, Beijing Shi, China; Fu Ying, Tsinghua University, Haidian District, Beijing Shi, China; Wanting Sun, Baidu Online Network Technology Co.,Ltd, Haidian District, Beijing Shi, China; Houbing Song, Embry-Riddle Aeronautical University, USA; Heyuan Shi, Big Data Institute, Central South University, Changsha, Hunan Province, China; Yu Jiang, Tsinghua University, Haidian District, Beijing Shi, China; Huizhong Li, WeBank, Nanshan District, Shenzhen Shi, China.(Yu Jiang is the corresponding author.)

0004-5411/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Fuchen Ma, Meng Ren, Fu Ying, Wanting Sun, Houbing Song, Heyuan Shi, Yu Jiang, and Huizhong Li. 2018. V-Gas: Generating High Gas Consumption Inputs to Avoid Out-of-Gas Vulnerability. *J. ACM* 37, 4, Article 111 (August 2018), 22 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

In order to avoid excessive consumption of resources, users are charged with gas to execute smart contracts [10]. Gas cost is determined by the amount of resources consumed by computation and storage [4][19]. Meanwhile, users may set a gas limit for each transaction to prevent potential malicious gas consumption. However, if a transaction needs more gas than the limit set by the user, an out-of-gas error occurs, and the transaction would fail. As a result, the transaction would be reverted, and the gas spent on executing the transaction is wasted. Malicious users may provide specially crafted inputs to trigger a high-gas cost transaction to launch a Denial-of-Service (DoS) attack [1]. Indeed, the out-of-gas error happens frequently. We investigated transactions made from 2:00 to 3:00 on May 1st, 2019, on Ethereum and found that 21 of them have out-of-gas errors. Gas waste caused by these failed transactions accounts for 5.64% of all gas consumption.

Extensive research has focused on estimating the gas limit of smart contract functions to avoid this error. For example, Solc [15], a widely-used compiler for smart contracts written in the Solidity language, provides a gas cost estimating tool to give a prediction on the gas consumption for each function. It statically analyzes each contract function and provides an estimation based on the opcodes contained in the function. Marescotti et al. [30] present a symbolic execution method to estimate gas cost. The inputs triggering the gas cost could be achieved by solving the equations generated by the constraints for each program path. Although those works solve the problem to some extent, in practice, their estimations are of limited practical usage, and users have to set the gas limit with manual evaluation. For instance, solc often under-estimates or estimates the gas consumption to be 'infinite' or underestimated values. The path conditions of symbolically based estimators are often complex to be solved, and hence many of the results are incorrect.

In fact, finding out how many gas units are used in a transaction is challenging before the transaction is executed. There are mainly two challenges. First, the gas consumption of a transaction comes from the gas charged by the opcodes and the gas used in data storage. It is usually not easy to estimate the data storage with static analysis and solve the constraints associated with the complex data structure with symbolic techniques. Furthermore, a function that contains many branches may behave differently under different block states or with different inputs. It is often infeasible to generate specific inputs manually, which could consume certain units of gas.

In this paper, we propose V-Gas, which automatically generates inputs that could lead to a high gas consumption of contract functions and reduce the underestimation ratio of existing works. V-Gas is designed based on feedback-directed fuzz testing. Specifically, V-Gas uses three steps to generate inputs: 1) Weighted control flow graph (W-CFG) generation, 2) Feedback-directed selection and mutation, 3) Contract execution. The first step generates a W-CFG for the contract, where each node in the CFG has a weight representing the gas cost of the node. The second step selects the seeds based on the feedback information, including the coverage and gas consumption, and applies several mutators to the selected seeds. The third step executes the functions with the mutated seeds and calculates the gas consumption of nodes and edges in W-CFG. In this way, V-Gas can make an estimation of gas cost in smart contract transactions.

For evaluation, we implemented V-Gas based on js-evm, one of the widely-used Ethereum Virtual Machine (EVM) for the execution of smart contracts. We conduct experiments on 1000 real-world transactions recorded on Ethereum, where contracts used by 736 transactions could be compiled successfully. Suppose we set the gas limit according to the limit estimated by solc.

In that case, 44.02% of the transactions will have out-of-gas errors, meaning that the recorded real gas consumption value is larger than the estimated value. Furthermore, among the remaining transactions, 33.43% are estimated to have an 'infinite' gas consumption, which may mislead the users to set a higher gas limit and drag the execution process of the transaction. While V-Gas could reduce the under-estimation ratio to 13.86%, and can also expose meaningful reference value for all the 'infinite' excessive gas estimation situations by solc. In order to evaluate the performance of the feedback-directed engine of V-Gas, we also implemented other well-known feedback-directed fuzzing engines to this problem, such as SlowFuzz [31] and PerFuzz [27]. The results show that V-Gas outperforms others. In particular, V-Gas generates the same or higher gas estimation on 97.8% of transactions with less time, usually within 5 minutes. More importantly, V-Gas found 25 previous unknown gas-related vulnerabilities in widely-used real-world contracts, which could lead to a Denial-of-Service attack, 6 of which have been assigned unique CVE identifiers in the US National Vulnerability Database (CVE-2019-8367, CVE-2019-8366, CVE-2019-8365, CVE-2019-8364, CVE-2019-7717, CVE-2020-18949), and the remaining 19 are pending approval. In general, this work makes the following contributions:

(1) We design and implement V-Gas, a feedback-directed fuzzing approach that generates inputs triggering a high-gas cost of smart contracts and reduces the underestimation ratio of existing works.
(2) We show two practical applications of V-Gas. First, V-Gas can guide on setting the gas limit to prevent potential out-of-gas errors. Second, V-Gas could be used to reveal gas-related vulnerabilities of existing contracts and settings. We have found 25 confirmed vulnerabilities, and 5 of which have been assigned with CVE IDs.

The rest of the paper is organized as follows. In Section 2, we introduce the background of blockchain and provide a high-level overview of V-Gas with a motivating example. We formally describe the design of V-Gas in Section 3. Section 4 demonstrates the evaluation results. In Section 5, we introduce some other related works. Section 6 makes a conclusion.

2 OVERVIEW

2.1 Background of Blockchain and Gas

Gas is a unit used to measure the workload of an operation in Ethereum. If an operation consumes more resources, it consumes more gas. This mechanism ensures that resources are not maliciously wasted. In the actual payment of gas costs, the number of tokens consumed will be determined by the gas unit cost and the current price of gas. During the actual execution of the transaction, users will set up a maximum gas amount to pay for the transaction, which is called the gas limit. If the gas consumed by the transaction is larger than the gas limit, the execution of the entire transaction will fail, which is called an out-of-gas exception.

Out-of-gas errors are undesirable because the miners will still charge the users for the gas though the transaction was reverted. However, it is also undesirable to set a very high gas limit. Due to the block gas limit limitation, the miners preferred transactions with a low gas limit. A high gas limit will prolong the completion of the transaction completing the transaction. Besides, a high gas limit setting, which allows more opcodes to be executed, may give attackers a chance to commit an attack.

2.2 A Motivating Example

In order to understand how our trace-based technique works, let us consider a contract deployed on Ethereum. The following code is intercepted from the contract [18] which is used to distribute tokens. The function we focus on is *distributeFixed*. This function transfers a fixed amount of ethers

```
 1  contract DistributeTokens is Ownable{
 2    ...
 3    function distributeFixed(address[] _addrs, uint _amoutToEach) onlyOwner{
 4    /* _addrs is an array of address input by the user */
 5      for(uint i = 0; i < _addrs.length; ++i){
 6      /* transfer some token to each address */
 7       tokenReward.transfer(_addrs[i],_amoutToEach);
 8      }
 9    }
10    ...
11  }
```

to each account that needs to be rewarded. Since this contract was deployed on Ethereum, 52 transactions have been made. Among these transactions, 80.77% of them are executed successfully, and 15.38% of the transactions failed because of out-of-gas errors. All the out-of-gas transactions called the function *distributeFixed*. The function is shown as below:

Suppose we use the gas estimation tool provided by solc to estimate the gas cost of this function. In that case, the result is 'infinite.' The primary source of gas consumption in *distributeFixed* is the transfer at line 8. The transfer function is implemented based on the opcode CALL. CALL has a multi-part gas cost: 1) the essential gas cost is 700 units; 2) If the transferring value is non-zero, another 9000 gas units are charged; 3) If the destination account does not exist, another 25000 gas units are charged. This is the gas consumption of opcodes. In an actual transaction, the memory used to store the transaction data on Ethereum also requires gas. We fetched three real transactions based on this function from Etherscan[24]. The inputs of the transactions and the results of gas used are shown in Fig. 1.

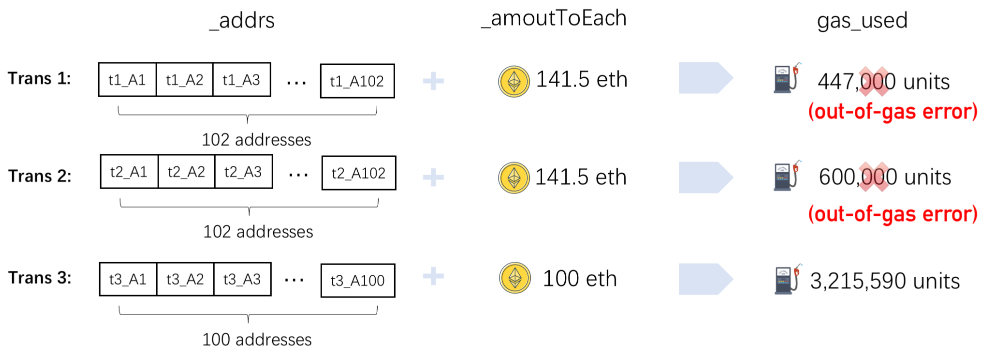


Fig. 1. Inputs for distributeFixed

The first transaction has 102 transfers. The gas limit set by users is 447,000 units. This transaction failed because of an out-of-gas error. The second transaction also has 102 ERC-20 transfers. The inputs of this transaction are the same as the first transaction. The user would like to try a higher gas limit for the transaction but significantly under-approximate the gas consumption, and, as a result, the transaction also failed. The third transaction has 100 ERC-20 transfers from the contract account to 100 other accounts. The amount in the transaction is 100 gwei. The gas cost of the transaction with this set of input is 3,215,590 units. This transaction is successfully executed. The first two transactions cost 1,047,000 units of gas in total. According to the gas price setting in these two transactions, 31,857,000 gwei is charged. **That means nearly six dollars are wasted after these transactions. These out-of-gas errors will not occur if the user knows in advance how much gas will be consumed by this function and the inputs to trigger the gas consumption. V-Gas addresses precisely this concern.**

In addition, there is also a vulnerability in this example contract. If the array '_addrs' has a very large length, the 'for' loop at line 5 will be executed many times. As a result, a large amount of gas will be cost, and the traction will eventually fail due to out-of-gas error. This is named as 'Unbounded Mass Operations' by the authors of MadMax [22]. This vulnerability has been assigned with a CVE identifier: CVE-2020-18949. According to the priory work, [16, 22], a safe way to make token transfers is shown as the following contract. For each 'for' loop round, one needs to check the left gas for now.

```
contract DistributeTokens is Ownable{
  ...
  function distributeFixed(address[] _addrs, uint _amoutToEach) onlyOwner{
    for(uint i = 0; i < _addrs.length && msg.gas >100000; ++i){...}
  }
  ...
}
```

2.3 V-Gas Idea

Figure 2 shows the CFG of this case. Let us assume that the initial inputs of this function are the same as the third transaction in Figure 1. V-Gas would set up an initial environment and execute the function with the initial inputs. The initial inputs would execute edge E1, E2, and E4 81 times. V-Gas will record the gas cost on each edge of the CFG and the total gas cost of the transaction. The initial inputs are reserved in the seeds pool for further mutation. V-Gas picks an input from the seeds pool at each iteration and randomly mutates it with several mutators.

As Table 1 shows, let us assume that the first mutated seed is *Input2*. V-Gas will use the new input to run the function. This input only executes the edge E1, E2, and E4 32 times. The gas costs on these edges are not increased with the new input, and the total gas cost is not increased either, so V-Gas will abandon it. Let us further assume that the following input is *Input3*. With this input, the edges E1, E2, and E4 are executed 100 times. The gas cost of these edges is increased, and this input will be reserved in the seeds pool. The gas cost for each edge and the total gas cost are updated. As for *Input 4*, the edges are executed 100 times as well, but if the total gas cost is increased, V-Gas will keep it in the seeds pool as well. All of the reserved inputs will be mutated to generate new inputs. V-Gas will choose several of the alternative inputs with the most gas cost of the whole function as the seeds to be mutated. In this way, V-Gas always keeps the inputs that could trigger a local higher gas cost or a global higher gas cost and makes mutations based on these inputs.

In this contract, if the length of the address array is larger, the transaction will make more transfer operations and consume more gas, eventually exceed the gas limit of a block which is 80,039,143 units [14]. To detect the vulnerability, V-Gas uses the WCFG to record the gas consumption of each seed in each fuzzing round. A seed with a large array of addresses will be stored for mutating. After several rounds, V-Gas could successfully generate inputs that trigger a high gas cost that exceeds the gas limit and detect the bug. A more thorough analysis of V-Gas's performance on various smart contracts will be presented in Section IV.

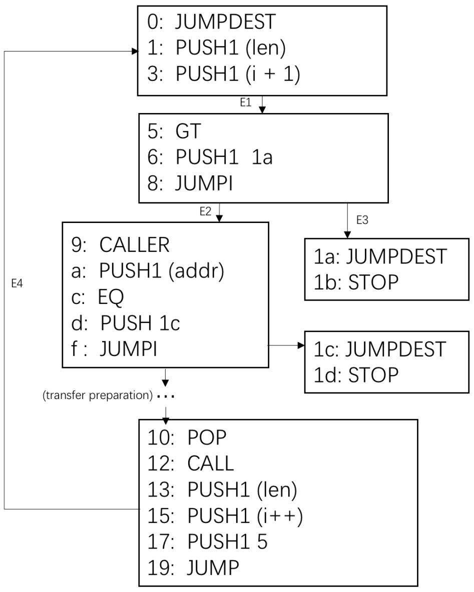


Fig. 2. The Control Flow Graph for function distibuteFixed

Table 1. Overview of the input mutation for V-Gas

Input_Num	_addr.length	_amount	edge_exe_times
Input1	81	100 eth	81
Input2	32	231 eth	32
Input3	100	109 eth	100
Input4	100	31 eth	100

3 V-GAS DESIGN

V-Gas uses three steps to generate inputs to maximize the gas cost: 1) Weighted control flow graph (W-CFG) generation, 2) Feedback-directed mutation and selection, 3) Contract execution. The architecture of V-Gas is shown in Fig 3. V-Gas starts the fuzzing process with a binary file (.bin) and application binary interface file (.abi) of the contract. V-Gas extracts the types of parameters and randomly generates some initial inputs for contracts, and generates some initial environment variables. Besides, V-Gas deploys the contract and prepares some function runners for execution. V-Gas uses the first input to run and calculates the gas cost information for each edge in W-CFG. After the execution, the seed and feedback information is passed to the feedback-directed selection and mutation part. The feedback information determines whether the seed is saved or not. All the saved seeds are mutated according to the V-Gas mutation strategies. After the mutation, V-Gas selects one seed from the queue and passes it to the function runners. This is a cyclic process, and V-Gas only stops when the user forces it to stop with time or value thresholds.

3.1 W-CFG Generation

V-Gas defines a weighted control flow graph that gives a gas consumption weight to each node contained in the traditional CFG. Each node of the CFG is a sequential non-branching block of

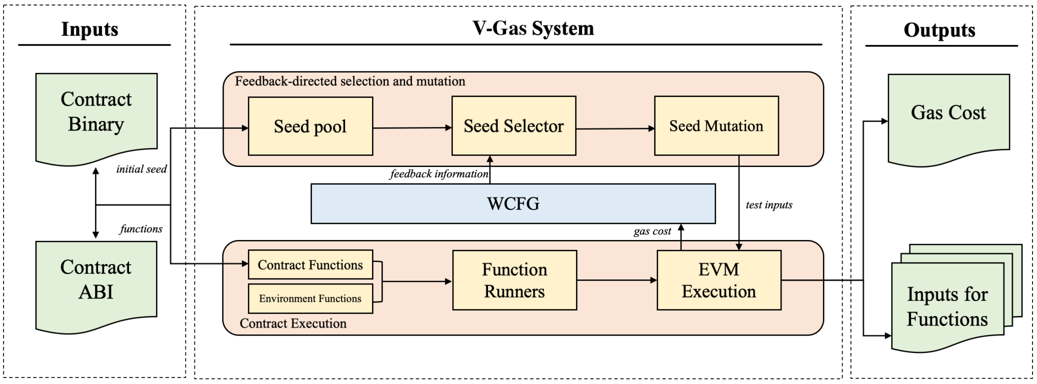


Fig. 3. V-Gas architecture, which consists of: W-CFG Generation, Mutation and Environment Setup. The inputs of V-Gas are the smart contract, while the outputs are the value of gas cost and the inputs that could trigger the cost.

opcodes. If there is a JUMP opcode or JUMPI opcode, a corresponding branch is introduced in the CFG. The weight of the node represents the gas cost of the node. As for the gas cost of some opcodes determined by the operands, a node may have an uncertain gas cost. If the node has an uncertain gas cost, the weight of this node is defined as gas consumption values excluding uncertainties. A simple W-CFG generated by V-Gas is presented in Fig. 4.

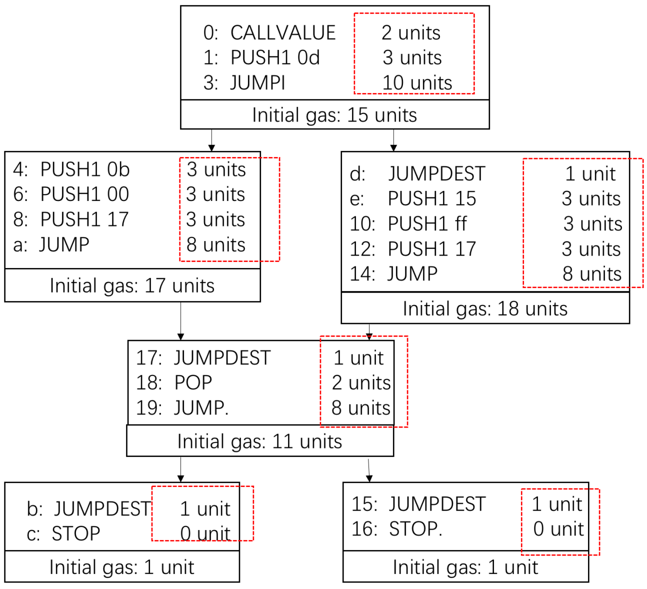


Fig. 4. W-CFG, each block has an initial gas costs. When there is a JUMP or JUMPI opcode, there is a new branch.

As we could see from the W-CFG presented above, each node has some opcodes and an initial gas consumption value. If the gas consumption of a node is not fixed, the initial gas consumption of the node will be the minimum value of the node. The W-CFG has a new branch when encountering a JUMP or JUMPI opcode. A JUMP opcode takes one parameter from the stack as the destination, whereas a JUMPI opcode takes two parameters. The first parameter is the destination, and the second one is the condition. As for this example, the destination of the JUMPI in the first node is ‘0d’, which is pushed into the stack in the first PUSH opcode. The condition of this JUMPI is the msg.value which is pushed by CALLVALUE opcode. If msg.value is 0, JUMPI will add the program counter by 1. If msg.value is not 0, JUMPI will jump to address 17. Each edge in the W-CFG has an initial gas consumption of 0. The gas consumption of an edge is defined as the total units of gas cost of the root node of this edge in the execution process.

3.2 Feedback-directed selection and mutation

Feedback-directed fuzzing is a core component of V-Gas. It collects feedback during test case execution and uses it to optimize the fuzzing process by selecting the right seeds for generating new test cases. Seeds selection determines whether a seed is a good one, and seeds mutation determines how the selected seeds mutate to generate new test cases. Algorithm 1 outlines the feedback-directed selection and mutation for V-Gas. V-Gas keeps running until the time limit set by the user is up, or the user would like to stop the process.

Algorithm 1: Feedback-directed selection and mutation

```
   Input   : initial_inputs
1  select_seed = initial_inputs;
2  old_feedBack = function.execute(select_inputs);
3  SeedsPool.push(initial_inputs);
4  repeat
5      foreach strategy of mutation_strategies do
6          newSeed = mutate(original_seed, strategy);
7          newSeed.change_Array();
8          new_seeds.push(newSeed);
9      end
10     foreach seed of new_seeds do
11         feedBack = function.execute(seed);
12         if feedBack.isInteresting(old_feedfack) then
13             SeedsPool.push(seed);
14             old_feedback = feedBack;
15     end
16     select_seed = select(SeedsPool);
17 until Duration_timeIsUp() || UserStop();
   Output: Inputs that lead to a high gas cost or Warnings
```

Seeds Selection. V-Gas automatically generates an initial input according to the types of parameters in smart contracts’ functions. The initial inputs are fed into the contract functions for executing the function for the first time. The feedback information of the execution, which contains gas cost and the number of times each edge is exercised, and the total gas cost, is stored by V-Gas. The initial seed is mutated then using several mutation operators. The mutated inputs are used to invoke the function. If the inputs consume more gas or visit certain edges in the W-CFG more times than existing test cases, the inputs are defined as interesting seeds and reserved. The selection process could be represented as the following:

$$\{s|S, total(s) > totalcur\}\cup$$
$$\{s|S, \exists e : E\ cost(s, e) > costcur(e)\}$$

where E represents the set of edges in the W-CFG, s represents seeds, and *total* represents the total gas cost of the transaction. All interesting seeds are stored in a priority queue called seeds pool. Seeds that consume more gas are assigned with higher priority. Each iteration, V-Gas picks a seed from the queue to mutate and puts the mutated seeds into the function again. V-Gas repeats the above process iteratively.

The approach might suffer from the optimum local problem, which occurs when the gas consumption does not grow, which hinders the execution of its subsequent nodes, which may lead to high gas consumption. In order to tackle this problem, V-Gas uses a sampling algorithm similar to MCMC [36] in the seed selector, which is implemented in the function *isInteresting* in the algorithm above. In this way, V-Gas saves those none-interesting seeds with a probability and likely avoids the optimal local problems.

Seeds Mutation. Seeds mutation operators of V-Gas could be divided into two groups: 1) Traditional mutations. For mutating common inputs of functions, we defined several mutators which are adopted from AFL [21]. Each selected seed is mutated using the mutation operators. The mutators we adopt include bit flipping, byte flipping, arithmetic increment, and decrement of integer values, replacing bytes with interesting integer values (0, MAX_INT), etc. 2) Gas-cost-specific mutations. The second group of mutation operators is used for mutating some special types of variables such as arrays. The length of an array variable may affect the execution times of some loops in a function. More loops mean more gas consumption usually. A new array replaces an array variable with a different length and different elements. The array elements are generated randomly according to the type of the parameters, and the length is randomly selected.

A seed in V-Gas is represented as a gene chain containing all bytes of the function inputs in one contract. V-Gas defines a gene_map to match the inputs for each parameter of each function. Besides, smart contracts need some state information to run. V-Gas defines state variables as new functions and connects them with the original functions in the contract. The inputs of the functions are state variablesa. V-Gas will generate new state variables with the same strategies as other inputs and store them in the gene chain.

3.3 Contract Execution

The input of V-Gas is the ABI file and binary file of a smart contract. The construction of an environment and several runners for a contract to run is essential. The environment of V-Gas is a simulator that emulates the same features as the Ethereum environment. For each contract, the first step is to deploy the contract, that is, to execute the construct function of the contract. The construction process is done by executing the binary code compiled from the contract source code on the Ethereum virtual machine. After one contract has been deployed, V-Gas generates a runner for each function. This runner contains the signature of the function and each input for the function. With the runner's help, V-Gas finds the matching piece of bytecode from the bin file according to the signature of the function. The function's signature is a string that is only related to the name and the inputs of a function. The number of inputs and their types are extracted by V-Gas from the ABI file and recorded in the runner of function. Ethereum virtual machine could run a transaction easily with the parameters provided by these drivers. To startup the fuzzing process, an initial seed is also needed. The initial seed is generated according to the ABI file of the contract. A map is used to record the mapping relation between the formal parameter and the actual parameter stored in the seed.

V-Gas is based on js-evm, and relies on several open-source programs such as solidity-types[11].

The reasons why we choose js-evm lie in such aspects: 1) Js-evm is a more light-weight EVM compared with geth. Though geth is used more widely than js-evm. However, we choose not to use it because it is not only an EVM but a client node. That means it contains lots of functions more than just a virtual machine. Using geth as the basic EVM will import more complexity to our tool. 2) 2) Js-evm is a widely used EVM whose repository on Github [17] has a star of 1.1k (Parity with 6.4k and py-evm with 1.3k). In the following, we mainly describe some implementation details for mutation part and environment variables definition part.

The seed is represented as a variable called gene in our implementation. A gene stores all of the inputs of the functions in one contract. Another structure named gene_map is used to fetch seeds for each input parameter of a function. For each input of the function defined in the contract, V-Gas fetches the name of the function, the length of the inputs, the name of the input and the type of the input as the components to generate a key. The key is used to get the seed from the gene_map. For each input, we use an open-source tool called 'solidity-types' to generate a random value according to the type of the parameter. The value is represented as a buffer and connected to the variable gene. The starting position and the ending position of the value in the gene are recorded in the gene_map. In this way, we generate random values for the inputs and maintain a structure to effectively get the values. As for the environment variables, we define the following variables in V-Gas: the coinbase of a block as an address variable; the difficulty of a block as an integer; the block number as an integer; the block timestamp as an integer; the sender as an address; the original address of a transaction named tx.origin as an address.

4 EVALUATION

In this section, we evaluate V-Gas to answer the following four questions: **a) Is out-of-gas a real threat? b) Is V-Gas capable of generating inputs that approximate the gas limit for a given smart contract function? c) What advantages does V-Gas have compared with other feedback-directed fuzzing tools aiming at maximizing the resource utilization? d) Does V-Gas have other application uses besides gas estimation?**

All our experiments were performed on a machine with 126GB memory, equipped with an Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz and running a 64-bit Linux operating system. All of the 255 actual world contracts used in our experiment are compiled by solc version 0.4.26. We took 1000 recorded transactions randomly from Etherscan [24] for all of the following experiments. We used solc to compile all these contracts, and contracts used by 736 transactions could be compiled successfully. The other contracts could not be compiled due to a mismatch between the contract and the compiler version. It is necessary to mention that some of the tools referred to in related work are not available, so we will not compare V-Gas with them in the following experiments. We do compare with the most widely used solc and reward-directed fuzzing such as SlowFuzz and PerfFuzz.

4.1 Risks of out-of-gas error

As many smart contract users use the estimation tool solc to set the gas limit when they made a transaction, we use the result of solc to investigate the current level of out-of-gas risk.

Fig.5 shows a comparison between the gas used of these transactions and the gas estimation results of the tool in solc. We use 'diff' to express this contrast. As shown in the following formula:

$$diff = solc_{est}(function) - real\_cost(trans)$$

where 'diff' represents the difference between the gas estimation of solc and the gas units actually consumed by the recorded transaction. In Fig 5, the X-axis represents the interval where 'diff' is

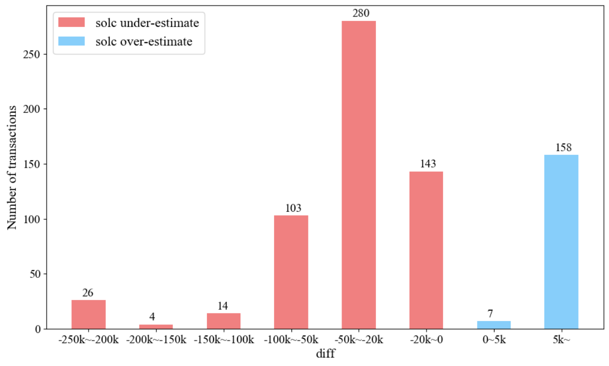


Fig. 5. Risks of out-of-gas error with the results of solc

.

located, and the Y-axis represents the number of transactions in that interval. The red bar represents the case where the 'diff' is negative, and the blue bar represents the case where the 'diff' is positive. We use 0 to represent the 'infinite' estimation given by solc. **As the result shows, if the users use the estimation results of solc to set the gas limit, an out-of-gas error is likely to happen. In particular, 44.02% of the transactions will be out-of-gas, and 33.43% of the transactions will evaluate to be 'infinite' without meaningful guidance.**

The underestimation is mainly because the gas estimation tool in solc uses static analysis to calculate the gas consumption of all opcodes. solc does not consider the gas consumption caused by data storage during the execution of transactions, which results in inaccurate estimation. Furthermore, if the function has some complex loops that depend on the function's inputs, the execution times of each opcode are uncertain. In this case, solc will estimate 'infinite,' and users have to set the gas limit based on manual analysis, which is highly non-trivial.

Answer to RQ-a: The results of current gas estimation tools can lead to a large number of out-of-gas errors. Out-of-gas error is indeed a potential threat.

4.2 V-Gas Performance

In order to answer the second question, we evaluated V-Gas with those functions and ran each function for 5 minutes. The results are shown in Figure 6. According to the relationship between the results of V-Gas and solc and real gas cost, we divide the transactions into three intervals. As shown in Fig. 6, the first interval represents the results of V-Gas are smaller than solc and real gas cost, the second interval represents the results of V-Gas are between solc's result and real gas cost, and the third one represents the results of V-Gas are larger than solc and real gas cost. The red bars in the figure represent that, within this interval, solc's result is smaller than the real gas consumption and would definitely result in out-of-gas error. The blue bar represents that the solc's result is larger than real consumption but would still be potentially attacked by the inputs generated by V-Gas, especially in the third interval.

The figure shows that V-Gas can exceed the actual gas cost in 634 transactions (493+17+124) of the 225 contracts, which counts for 86.14% of all transactions. In these transactions, if user sets gas limit according to the results of V-Gas instead of those of solc, out-of-gas problems might

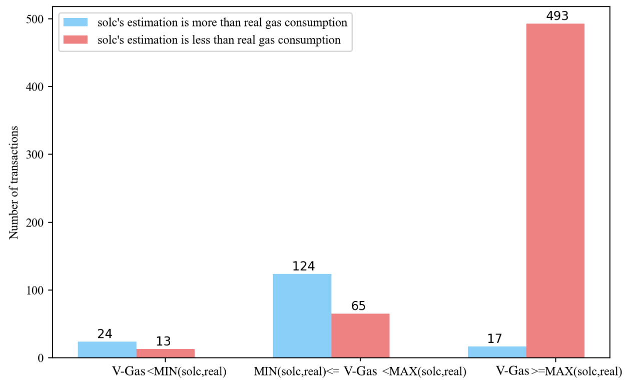


Fig. 6. V-Gas results compared with results of solc and the real gas consumption in the transaction

be significantly reduced. As we illustrated in the formal or previous section, the results of solc exceed the real gas consumption on only 22.55% of the transactions. **If V-Gas is used for gas limit settings, the risk of out-of-gas occurring in these transactions will be greatly reduced, from 44.02% to 13.86%.**

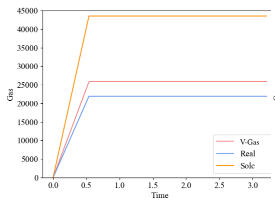
We use the 8 illustrative examples to observe the effectiveness of V-Gas. The results are shown in Table 2. Column V-Gas represents the results of maximum gas given by V-Gas in 5 minutes. Column Time shows how long it takes for V-Gas to find the maximum gas cost. The results in the last two columns show the increase ratio of gas cost given by V-Gas compared with solc and the real situation.

Table 2. V-Gas results on the 8 functions

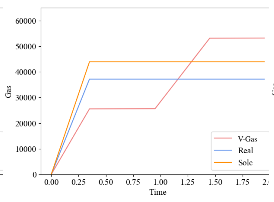
Contract	V-Gas	Time	/solc	/real
BAToken	25879	0.542	-40.5%	18.08%
DSTokenBase	53179	1.949	21.08%	43.05%
Dogg	55072	0.583	21.80%	4.23%
AccessoryData	49601	0.224	18.32%	-0.43%
ENIGMA	26949	8.496	/	4.90%
KittyItemMarket	116324	0.417	/	2.18%
CommunityChest	32884	0.173	/	1.11%
Future1Exchange	32991	0.389	/	4.79%

We could find that V-Gas could generate inputs to trigger a gas higher than the estimation result of solc by 20% on average. Except for the contract ‘AccessorData’, V-Gas’s results are higher than the maximum value of real transactions. For the contract ‘BAToken’, ‘V-Gas’ gives a lower result than solc, the reason is as follows. For the contract ‘AccessoryData’, the gas consumption is lower than the real value by only 0.43% which means V-Gas’s result is very closed to the real value. If the user sets the gas limit with the estimation result by solc, 7 of the contracts are in danger of out-of-gas error. However, with V-Gas’s result, only one contract may occur out-of-gas error.

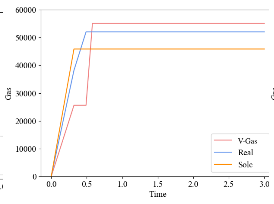
Besides the actual out-of-gas situations in these transactions, some of the transactions face potential out-of-gas errors. A potential out-of-gas means the limit of some successfully executed transactions is not correct in some other situations. Figure 7 shows the relationship between the results of V-Gas on the eight functions and fuzzing time. As the figure shows, V-Gas could approach or even exceed the real gas used in the transaction, which means some functions have potential our-of-gas error. With the inputs generated by V-Gas, out-of-gas errors may occur. We will use three functions to illustrate the three types of inputs that V-Gas generated within 5 minutes below.



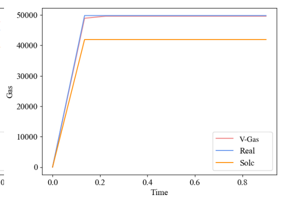
(a) V-Gas on BAToken.sol



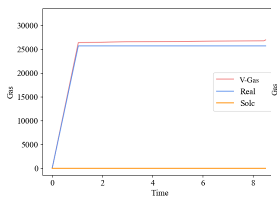
(b) V-Gas on DSTokenBase.sol



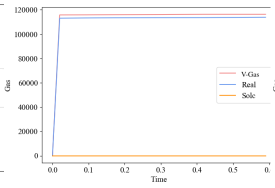
(c) V-Gas on Dogg.sol



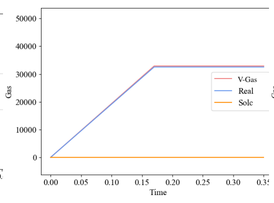
(d) V-Gas on AccessoryData.sol



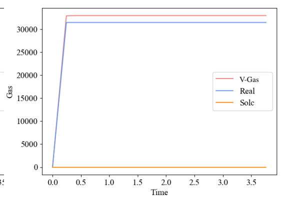
(e) V-Gas on ENIGMA.sol



(f) V-Gas on KittyItemMarket.sol



(g) V-Gas on CommunityChest.sol



(h) V-Gas on Future1Exchange.sol

Fig. 7. V-Gas on 8 functions. V-Gas exceeds the gas estimation results given by solc on 7 functions. V-Gas also exceeds the real maximum gas cost on 7 functions. This means with the input generated by V-Gas an out-of-gas error may occur in these transactions.

V-Gas exceeds the results of solc and the real gas cost in the transaction. The first function we focused on is *transfer* taken from the contract Dogg. This function is used to transfer some ether to some other addresses. This function needs two parameters: *to* and *value*. The first is the destination address, and the second is the number of tokens to be transferred. The function is shown as follows:

```
function transfer(address _to, uint256 _value) public returns (bool success)
{
    _transfer(msg.sender, _to, _value);
    return true;
}
```

'msg.sender' in function' _transfer' represents the caller of the transaction. This is a state variable used in Ethereum. V-Gas finds three different inputs for this function. At 0.32 seconds, V-Gas generates an input that costs 25603 gas units. At 0.49 seconds, another input is generated, costing 25607 units of gas. After 0.58 seconds, V-Gas finds an input costing 55072 units of gas. This result is higher than the real gas cost in the transaction based on this function, which is 52835 units. This means there may be potential out-of-gas errors in these transactions when given the inputs generated by V-Gas.

The same scenario occurred in 28 of the 30 transactions based on eight contracts mentioned above. Overall, it occurs in 69.29% of all the transactions in our experiment. In this situation, if the user used solc to set the gas limit of the transaction,there is a chance that out-of-gas error could occur. In this situation, a user could just set the gas limit as a value that is a little larger than the result of V-Gas.

V-Gas exceeds the results of solc but is lower than the real gas cost in the transaction. The second function we focused on is ‘addSERAPHIM’ from the contract ’AccessoryData’. The function is shown below. The function is used to add new seraphim to the system. The ’onlyCREATOR’ is a modifier used to limit the scope of the executor of the function. The function first checks whether the new seraphim is a new one, then it adds the number of the seraphims by one.

```
function addSERAPHIM(address _newSeraphim) onlyCREATOR public {
        if (seraphims[_newSeraphim] == false) {
            seraphims[_newSeraphim] = true;
            totalSeraphims += 1;
        }
}
```

V-Gas generates two gas consumption results for this function. At 0.135 seconds, V-Gas found a set of inputs that cost 48937 units of gas. At 0.224 seconds, V-Gas gave another gas consumption result of 49601 units. Solc gives a result of 41921 gas units for this function. However, in the real transaction, 49816 units of gas have been consumed. V-Gas finds a result better than solc but lower than the real situation. The reason that V-Gas fails to find the worst case for gas consumption may be the time for fuzzing. That is, with more time, more accurate estimation is typically identified. However, this result is higher than the one of solc, which means the risk of out-of-gas error is decreased with the help of V-Gas.

V-Gas exceeds the real gas cost but is lower than the estimation result by solc. The last function we focused on is *transfer* in BAToken.sol. This function is listed as follows.

```
function transfer
(address _to, uint256 _value) returns (bool success)
{
  if (balances[msg.sender] >= _value && _value > 0)
  {
    ...
    Transfer(msg.sender, _to, _value);
    return true;
  }
  else
    return false;
}
```

Solc predicts that 43517 gas units would be charged while V-Gas gives a result of 25879 gas units which is lower than the estimation of solc. To find out the reason for the underestimation, we conducted a thorough investigation of this result. The following codes are extracted from the opcode sequences of the execution.

```
{"pc":1420,"op":85,"gas":"0xffffaa6c","gasCost":"0x0",
"stack":["0xd356a28b","0x1f8","0x0","0x1","0x1000000000000
0000000000000000000000000000","0x0"],
"depth":0,"opName":"SSTORE"}
```

’SSTORE’ is an opcode used to store data in the memory. The opcode takes two parameters from the stack. The first element is the value that needs storing. The second element is the address that the value is stored at. As for this example, the value ’0x10000000000000000000000000000000000000000’ is the address and the value '0x0' is the value stored at this address. As stated in the Ethereum yellow paper[13], the SSTORE opcode will give a gas refund if it changes some non-zero value to a zero value in some address. The refund gas is half the gas used before, which could not exceed 15,000. In our example, the gas used is 43,517. So the refund gas is 15,000. Solc did not consider the refunding, which leads to a higher gas estimation. In this situation, V-Gas gives a more accurate result than solc, which is also higher than the real transaction costs. As mentioned in Section II, fewer gas limit settings often imply faster execution. Besides, excessive gas estimates are also dangerous because attackers can exploit the gas loss.

Answer to RQ-b: The out-of-gas errors could be effectively avoid with the help of V-Gas. V-Gas is capable of generating inputs that approximate or exceed the real gas cost of the transaction in smart contract, and reduce the under estimation of solc dramatically.

4.3 Feedback-Directed engine in V-Gas

V-Gas is a feedback-directed fuzzing method. However, some other feedback-directed fuzzing tools could be potentially used for this problem. In order to evaluate how effective the feedback-directed engine in V-Gas is, we compared V-Gas with two other fuzzing tools: SlowFuzz [31], and PerfFuzz [27]. Slowfuzz and PerfFuzz both use feed-back directed fuzzing to maximize resource utilization in a program. SlowFuzz provides a fuzzing method to detect algorithmic complexity vulnerabilities for C/C++ programs. SlowFuzz uses resource-usage-guided evolutionary search techniques to automatically find inputs that maximize computational resource utilization for a given application. Similar to SlowFuzz, PerfFuzz generates inputs via feedback-directed mutational fuzzing. PerfFuzz uses multi-dimensional feedback and independently maximizes execution counts for all program locations.

4.3.1 Implementations for Gas Used Problem. In order to apply the fuzzing tools mentioned above, based on their open-source version, we customize their feedback mechanisms for fuzzing smart contracts. Generally, the challenges that need to be solved to implement these tools on smart contracts lie in such aspects: 1) The construction of smart contract CFG is different from traditional programs. SlowFuzz and PerfFuzz both depend on the CFG of the programs. However, in a smart contract CFG, a block contains Ethereum opcodes, and an edge is generated when there is a 'JUMP' or 'JUMPI' opcode. In order to apply the tools on smart contract CFG, we supplemented the W-CFG defined in V-Gas with an edge counter and a path counter. The edge counter is used to count the execution times of each edge, and the path counter records the length of the execution path. 2) The mutation strategies of SlowFuzz and PerfFuzz are not suitable for smart contracts. In traditional programs, a seed means a set of inputs. However, in smart contracts, the order of the transactions(functions) may also affect the execution paths of a contract. A seed in smart contracts should contain the function order too. Thus, the mutation strategies used in SlowFuzz and PerfFuzz cannot be used on smart contract seeds. In order to solve this problem, we extracted the feedback-directed strategies of SlowFuzz and PerfFuzz and connected the mutation strategies of V-Gas with them. SlowFuzz retains the seeds that could maximize the total path of program execution. While PerfFuzz retains seeds that could lead to more execution of an edge in the program CFG. The seeds that are selected by two tools will be mutated under V-Gas's strategy.

In our customization, we use the same CFG as V-Gas. For SlowFuzz, we calculated the length of the paths in the CFG during execution with the path counter. The total length of a path is represented as the sum of execution times for all edges. As for PerfFuzz, we record the maximum execution times for each edge in the CFG with the edge counter. If a seed could execute some edges more than the times recorded as the maximum value, it is selected as a seed for generating more test cases.

4.3.2 Comparison among Different Fuzzing engines. We compared V-Gas with the above-mentioned fuzzing tools as well as a random fuzzing engine. The random engine saves all the seeds randomly without any feedback information. The comparison is committed on the 736 transactions we took. The result is shown in Table3.

Table 3. The number of transactions that each fuzzer can maximize the gas estimation compared with other engines

	PerfFuzz	RnadomFuzz	SlowFuzz	V-Gas
total	628	644	623	**722**
percentage	85.1%	87.3%	84.4%	**97.8%**

Table 4. Performance of each fuzzer on 8 smart contracts

	PerfFuzz			RandomFuzz			SlowFuzz			V-Gas		
	Result(unit)	Time(s)	GasRate	Result(unit)	Time(s)	GasRate	Result(unit)	Time(s)	GasRate	Result(unit)	Time(s)	GasRate
BAToken	25879	0.712	36347	25879	3.012	8592	25879	2.166	11948	25879	0.542	**47747**
DSTokenBase	53179	2.412	22048	53179	1.904	27930	53179	0.579	**91846**	53179	1.949	27285
Dogg	55072	4.457	12356	25667	2.287	11223	55008	2.549	21580	55072	0.583	**94463**
AccessoryData	49601	0.704	70456	49601	2.315	21426	49601	1.467	33811	49601	0.224	**221433**
ENIGMA	26949	52.246	516	26693	60.543	441	26565	3.751	**7082**	26949	8.496	3172
KittyItemMarket	116196	2.522	46073	116196	0.546	212813	116324	1.681	69199	116324	0.417	**278954**
CommunityChest	32884	0.426	77192	32884	158.368	208	32820	0.651	50415	32884	0.173	**190081**
Future1Exchange	32991	1.234	26735	32991	1.601	20606	32991	0.767	43013	32991	0.389	**84810**

V-Gas performs the best in 97.8% of the transactions compared with other fuzzing engines, which is the best among all of the engines. RandomFuzz performs better than the other two fuzzing engines with a ratio of 87.3%. The reason for this result may be that the directed strategies used in PerfFuzz and SlowFuzz are not suitable for this problem. Maximizing the execution times for each edge or the total length of the execution path may lead to the highest gas cost due to the opcodes. However, the gas consumption of a transaction also contains the gas cost due to data storage. For this reason, the fuzzing engines in PerfFuzz and SlowFuzz are not ideal for this problem. In order to observe the difference among the four engines more specifically, we count the fuzz results of the four engines on the eight contracts as examples. The results are shown in Table4. In the table, we introduce a new concept called 'GasRate', which is defined as the ratio of gas result to time.

This variable is used to measure the effectiveness of the fuzz engine. If this tool can find higher gas consumption in less time, which means a higher GasRate, we believe the effectiveness of this engine is better than others. As the table shows, V-Gas has the highest GasRate on six contracts. V-Gas finds the highest gas cost among four engines on all of the eight contracts. V-Gas can find the best results in a relatively short period of time. We now use the contract 'ENIGMA' to show the fuzzing process of these fuzzing engines. The results of 4 fuzzing engines are shown in Figure 8.

As we could see from the figure, V-Gas could reach a high result in a short time which is 8.5 seconds to be precisely. PerfFuzz gives the same result as V-Gas at 52.24 seconds. SlowFuzz and RandomFuzz performs worse than the other two engines even with more than 5 minutes.

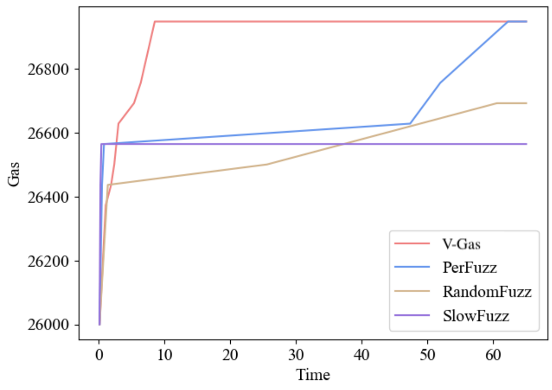


Fig. 8. Trend of results given by 4 fuzzing engines on function 'Activate' from contract 'ENIGMA'

Answer to RQ-c: V-Gas performs better than some well used feedback-directed fuzzing engines in this problem, where the gas weighted control flow graph and the gas directed seed selection and mutation strategies contribute to the improvements.

4.4 Out-of-gas related Vulnerabilities Detection

V-Gas aims at guiding the setting of the gas limit. However, another application of V-Gas is to detect gas related vulnerabilities. A transaction with a gas cost that exceeds the gas limit will be reverted. Malicious attackers may exposure a set of inputs to trigger a loop to consume excessive gas. As a result, the service provided by the contract is stopped. Such vulnerabilities may cause a loss of money. The following code is fetched from a real-world smart contract.

```
1 function transfer(address[] _address, uint256[] _values) onlyOwner public {
2  require(_address.length == _values.length);
3  for (uint i = 0; i < _address.length; i += 1) {
4         token.transfer(_address[i],_values[i]); }
5  AirDroppedTokens(_address.length);
6 }
```

The function 'transfer' is used to transfer tokens to a series of addresses which is stored in the array '_address'. However, if the length of '_address' is too big, the 'for' loop at line 3 will be executed many times. Each round will cost lots of gas. As a result, if the gas consumption is bigger than the gas limit, an out-of-gas error will occur, and the transaction will revert. That means all the token transfer operations will not succeed.

The severity of the bug is reflected in the fact that it can make the normal transfer process fail. One possible attack scenario will be like this: an attacker created a bunch of addresses and added them into the array _address. This can be done easily by just investing a very small number of tokens. After that, when several normal users need to get some rewards, the owner of this contract will call the function 'transfer' to send tokens to them. However, as the attacker created a large number of garbage addresses, the function will cost lots of gas and exceeded the gas limits. The token transfer to normal users will fail.

The attack scenarios for these vulnerabilities are built on a zero-trust basis. Thus, V-Gas is a tool that can secure the blockchain system under a zero-trust scenario. In recent years, blockchain is widely used in IoT systems. For example, a recent work [33] provides an IoT management framework that embraces blockchain technology to help companies to form a supply chain effectively. The smart contracts used in the blockchain of IoT systems need to be secured. V-Gas can secure smart contracts from suffering out-of-gas errors. In terms of this, the IoT systems using blockchain can be protected by V-Gas.

In order to avoid this kind of bug, Ethereum wiki for safety[16] suggests writing contracts like the following. In each round of the 'for' loop, one needs to check how much gas is left for now.

```
function transfer(address[] _address, uint256[] _values) onlyOwner public {...
 for (uint i = 0; i < _address.length && msg.gas > 10000; i += 1) {... }
}
```

V-Gas could generate a set of inputs that identifies this problem. This previously unknown issue has been assigned with a CVE ID. We have already detected 25 confirmed out-of-gas vulnerabilities with the help of V-Gas, 6 of which have been assigned unique CVE identifiers in the US National Vulnerability Database, and the other 19 are appended for approval. We also applied MadMax on all the 25 vulnerable contracts found by V-Gas. The result shows that MadMax can find all of the 25 vulnerabilities. As for time consumption, MadMax costs 5.4 seconds on average to analyze these contracts, while V-Gas costs only 2.2 seconds on average. The reason is that MadMax leverages the datalog analysis to detect vulnerabilities in smart contracts. The datalog engine used by MadMax is Souffle. It needs to decompile a contract from bytecode into an IR. Thus, MadMax usually costs more time to analyze a contract than V-Gas.

In addition, V-Gas can enhance MadMax in two aspects: 1) First, MadMax can only detect whether there is a vulnerability in a contract. However, it cannot give out a concrete input that can trigger such vulnerability. V-Gas leverages different seeds to get high gas consumption. Thus, V-Gas can give out the inputs to reproduce the bug found by MadMax. 2) V-Gas can reduce the false positives reported by MadMax. As a static analysis tool, MadMax can report many false positives. V-Gas can assist MadMax by checking whether the vulnerability can be triggered by a concrete input. Thus, some false positives can be eliminated.

V-Gas mainly focuses on generating inputs to trigger a high gas consumption. Detecting gas-related vulnerabilities is just one of the applications of V-Gas. V-Gas can also be used to make guidance for the set of gas limit before making transactions and help Ethereum users avoid out-of-gas errors and save more money.

Answer to RQ-d: V-Gas has practical values and is really helpful in gas estimation tasks as well as in looking for gas-related vulnerabilities.

4.5 Threats to Validity

Mutation and selection strategies limitations. The mutation strategies of V-Gas are hybrid for the current version. They are designed based on the mutation strategies in AFL. V-Gas uses some global functions to represent the environment variables. For mutations, these variables are given different values randomly on the basis of their types. This mutation strategy performed effectively for V-Gas. However, more efficient mutation strategies for environment variables could be adapted to V-Gas to build a more complex environment for the contract to run. A more complex environment means more accounts and more state information, which may lead to some different behaviors of smart contracts. Second, even we have implemented MCMC in the seed selection, we will still face the local optimization problem, which needs to be solved with more sophisticated algorithms.

Smart contracts support limitations. The current prototype of V-Gas could not deal with the smart contract functions that call another contract deployed on some address in Ethereum. The gas cost of these functions is determined by the contract, which V-Gas could not calculate for now. This problem may be solved by fetching the called contract. We will try to extend V-Gas to support these kinds of contracts in our future work.

Data collection limitations. One of the subjective threats is data collection. The experimental process contains a large amount of data, and there are often some errors in the statistics and analysis phases. In order to solve this problem, we use fully automated script files to statistics and analyze the data. To a certain extent, script files could avoid the impact of the data collection process on the experimental results. Another possible subjective threat of this experiment is the time of the fuzzing process. The result of the fuzzing technique depends on the duration time sometimes.

5 RELATED WORK

Gas cost estimation. Solc [15] provides an API to estimate the gas cost of the functions of contracts as shown in Section 5. However, the estimation way of solc has many constraints, leading to many 'infinite' estimations and underestimations. GASTAP [2] infers sound gas upper bounds for all public functions in smart contracts with complex transformation and analysis processes on the code. Matteo Marescotti [30] have presented a solution to the problem of estimating the gas consumption for Ethereum smart contracts based on techniques inspired by bounded model-checking techniques. These works contribute lots of premiere ideas for further research, but none are publicity available except for Solc.

Worst Case Execution Time. Kirner et al.[26] present a hybrid WCET analysis framework using runtime measurements together with static program analysis. Stefan et al.[32] give an overview on available options to extract the traces which are used to estimate the worst-case execution time and highlights the advantages and disadvantages of these options. Another work[37] generates as few tests as possible for paths whilst ensuring that the WCET is exhibited by at least one case in the set.

Security of Ethereum and Smart Contracts. Security of Ethereum and smart contracts is essential to the development of blockchain techniques. Some works make a survey on the attacks of Ethereum and smart contracts [3]. Plenty of research focuses on Ethereum and smart contract security. For example, echidna [9] provides a fuzzing framework for developers to check if there is any problem in their contracts. It takes a list of invariants (properties that should always remain true) as input. Each invariant generates random sequences of calls to the contract and checks if the invariant holds. ContractFuzzer [25] defines some oracles to detect vulnerabilities in smart contracts, such as dangerous delegate bugs. Vandal [5] presents a security analysis framework for Ethereum smart contracts. Vandal consists of an analysis pipeline that converts low-level Ethereum Virtual Machine (EVM) bytecode to semantic logic relations. Another recent work[23] gives the first complete small-step semantics of EVM bytecode. This formalization relies on a combination of hyper and safety properties. MadMax [22] classifies and identifies gas-focused vulnerabilities and presents a static program analysis technique to detect gas-focused vulnerabilities with very high confidence automatically. EVMFuzz [20] uses differential testing to fuzz different versions of EVM and try to find out the vulnerabilities in EVM. Another work, EVM* provides a framework to make a reinforcement on EVM [29] to test whether there is a dangerous operation during transaction execution and stop the dangerous transaction in time. Another recent work [34] presented a security analyzer for Ethereum smart contracts that is scalable, fully automated, and able to prove contract behaviors as safe/unsafe concerning a given property.

Performance problems detection. Much work has been done to detect the waste of gas and the optimization pattern for gas consumption. For example, [7] identifies 24 anti-patterns from the execution traces of real smart contracts. Besides, they design GasReducer, the first tool to automatically detect all these anti-patterns from the bytecode of smart contracts and replace them with efficient code through bytecode-to-bytecode optimization. Another work [6] proposes an emulation-based framework to measure the resource consumptions of EVM operations automatically.

Fuzzing tools. Traditional fuzzing techniques aim at finding memory safety problems. Fuzzing testing could be divided into mutation-based fuzzing and generation-based fuzzing. Mutation-based fuzzing such as AFL[21] mutates seeds with mutation strategies for each fuzzing process. Generation-based fuzzing generates inputs based on the domain knowledge of the program. Some works focus on the improvement of AFL to increase the fuzzing performance. Enfuzz[8] proposes a method to combine several fuzzers with working together and sharing the seeds. Pafl[28] utilizes efficient guiding information synchronization and task division to extend those existing fuzzing optimizations of single-mode to industrial parallel mode. Safl[35] focuses on the combination of symbolic execution and fuzzing testing tools to increase the coverage. Some other works focus on the techniques to find the weakness of fuzzing techniques, such as Anti-fuzz[12]. Their results show that it is relatively easy to implement and apply anti-fuzzing techniques that are able to completely mask crashes and, by extension, vulnerabilities from fuzzers. However, this is hard to implement in smart contracts deployed on Ethereum because the contracts on Ethereum could not be changed since it is deployed.

Main difference. For the gas estimation tools, V-Gas focuses on generating inputs to trigger a high gas consumption. Besides, in section V, we demonstrated V-Gas could perform better than the available related tools. For WCET estimation works, the gas consumption in Ethereum is quite different from the execution time in traditional programs. Gas cost is related to the inputs of the contract as well as the state of the blockchain. V-Gas provides a domain-specific method for estimating the worst case of gas consumption. V-Gas uses different mutation strategies and detects different vulnerabilities for the traditional fuzzing tools aimed at smart contracts. Echidna and ContractFuzzer randomly mutate the inputs for each generation, while V-Gas combines the strategies of AFL and the random methods for environment variables. Besides, V-Gas aims to generate inputs that trigger a high gas cost but not detect memory safety bugs. As for the performance bug detection tools, V-Gas could automatically find the bottleneck of gas cost for a smart contract, either for gas limit reference or vulnerability detection.

6 CONCLUSION

In this paper, we designed V-Gas, a fuzzing approach to generate inputs that could lead to high gas consumption, which can reduce the underestimation ratio of existing tools and detect previously unknown out-of-gas vulnerabilities. First, V-Gas will generate a W-CFG for the function. Second, V-Gas uses feedback-directed selection and mutation strategies. Finally, V-Gas provides an environment for contract transactions. The result demonstrates that V-Gas could successfully generate some inputs to trigger the high gas cost in a short time. In 86.14% of the transactions, the gas cost given by V-Gas is higher than the gas cost of real-world transactions or values estimated by solc, which could successfully decrease the risk of out-of-gas error from 44.02% to 13.86% of all transactions. For the practical application of V-Gas, we evaluate V-Gas on more real-world contracts and detect 25 previously unknown gas-related vulnerabilities, and 6 of which are assigned with CVE Ids.

Our future work will focus on the optimization of V-Gas. We will design some gas-related mutators, accelerate the fuzzing procedure with some static analysis such as the result of solc, and try to detect more gas-related vulnerabilities in smart contracts with V-Gas. Besides, we would like to find some gas hot spots with the help of V-Gas and try to design some patterns to optimize the gas cost of these spots.

7 ACKNOWLEDGMENTS

This research is sponsored in part by the NSFC Program (No. 62022046, No.92167101, U1911401, 62021002, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706203), and the Tsinghua-Webank Scholar Project (No. 20212001829).

REFERENCES

[1] 2019. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/known_attacks/gas-limit-dos-on-the-network-via-block-stuffing.

[2] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2018. GASTAP: A Gas Analyzer for Smart Contracts. *CoRR* abs/1811.10403 (2018).

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. A survey of attacks on Ethereum smart contracts. *IACR Cryptology ePrint Archive* 2016 (2016), 1007.

[4] BlockGeeks. 2019. Ethereum-gas. https://blockgeeks.com/guides/ethereum-gas/.

[5] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[6] 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.

[7] 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.

[8] Yuanliang Chen, Yu Jiang, Jie Liang, Mingzhe Wang, and Xun Jiao. 2018. Enfuzz: From ensemble learning to ensemble fuzzing. *arXiv preprint arXiv:1807.00182* (2018).

[9] crytic. 2019. echidna. https://github.com/crytic/echidna/.

[10] Chris Dannen. 2017. *Introducing Ethereum and Solidity*. Springer.

[11] duytai. 2019. solidity-typesp. https://github.com/duytai/solidity-types.

[12] Emil Edholm and David Göransson. 2016. Escaping the Fuzz - Evaluating Fuzzing Techniques and Fooling them with Anti-Fuzzing.

[13] Ethereum. 2019. Ethereum. https://ethereum.github.io/yellowpaper/paper.pdf.

[14] Ethereum. 2019. gas_limit. https://ethstats.net/.

[15] Ethereum. 2019. Solc. https://github.com/ethereum/solidity.

[16] Ethereum. 2020. solidity-safety. https://github.com/ethereum/wiki/wiki/Safety.

[17] Ethereumjs. 2021. ethereumjs evm. https://github.com/ethereumjs/ethereumjs-monorepo.

[18] etherscan. 2019. contract. https://etherscan.io/address/0x2359ffa8e5b7a7f3d44f4aa9fbe03d061a4b5d0c.

[19] Ethos. 2019. What is ethereum gas. https://www.ethos.io/what-is-ethereum-gas/.

[20] Ying Fu, Meng Ren, Fuchen Ma, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2019. EVMFuzz: Differential Fuzz Testing of Ethereum Virtual Machine. *arXiv preprint arXiv:1903.08483* (2019).

[21] google. 2019. American fuzzing lop. http://lcamtuf.coredump.cx/afl/.

[22] 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), 116.

[23] 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.

[24] https://etherscan.io/. 2019. Etherscan. https://etherscan.io/.

[25] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. (2018).

[26] Raimund Kirner, Peter Puschner, and Ingomar Wenzel. 2004. Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation.

[27] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Xiaodong Song. 2018. PerfFuzz: automatically generating pathological inputs. In *ISSTA*.

[28] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 809–814.

[29] Fuchen Ma, Ying Fu, M Ren, Mingzhe Wang, Yu Jiang, Kaixiang Zhang, Huizhong Li, and Xiang Shi. 2019. EVM*: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), 554–558.

[30] Matteo Marescotti, Martin Blicha, Antti EJ Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 450–465.

[31] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. (2017).

[32] Stefan M. Petters. 2003. Comparison of Trace Generation Methods for Measurement Based WCET Analysis. In *WCET*.

[33] Qun Song, Yuhao Chen, Yan Zhong, Kun Lan, S. Fong, and Rui Tang. 2021. A Supply-chain System Framework Based on Internet of Things Using Blockchain Technology. *ACM Transactions on Internet Technology (TOIT)* 21 (2021), 1 – 24.

[34] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *ACM Conference on Computer and Communications Security*.

[35] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 61–64.

[36] Wiki. 2019. MCMC. https://en.wikipedia.org/wiki/Markov_chain_Monte_Carlo.

[37] Nicky Williams and Muriel Roger. 2009. Test generation strategies to measure worst-case execution time. *2009 ICSE Workshop on Automation of Software Test* (2009), 88–96.