IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 20, 2020, accepted May 12, 2020, date of publication May 18, 2020, date of current version June 8, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2995183

GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities

IMRAN ASHRAF[1], XIAOXUE MA[1], BO JIANG[2], AND W. K. CHAN[1]
[1]Department of Computer Science, City University of Hong Kong, Hong Kong
[2]School of Computer Science and Engineering, Beihang University, Beijing 100191, China

Corresponding author: W. K. Chan (wkchan@cityu.edu.hk)

This work was supported in part by the GRF of Hong Kong Research Grants Council under Project 11214116 and Project 11200015, in part by the NSFC of China under Projects 61772056 and 61690202, in part by the External Grant under Project 61400020404, in part by the HKSAR ITF under Project ITS/378/18, in part by the CityU MF_EXT under Project 9678180, and in part by the CityU SRG under Project 7004882, Project 7005216, and Project 7005122.

ABSTRACT Ethereum is a kind of blockchain platform where developers may develop and run programs called smart contracts. It inherently relies on gas consumption within a specified allowance to constrain code execution, making every instruction along an execution path to be a location for raising an exception. In this paper, we present GasFuzzer, the first work in exploring the effects of gas allowance manipulation to expose gas-oriented exception security vulnerabilities. GasFuzzer consists of two phases. The first phase introduces a gas-greedy strategy to favor transactions having higher gas consumption for mutation to obtain test transactions with different gas consumptions. The second phase introduces a novel notion of fractional gas consumption coverage and a novel gas-leveling strategy. It applies them to mutate the gas allowances of some of these transactions resulting in the highest gas consumptions produced in the first phase followed by applying these allowance-mutated transactions together with those which remained non-mutated to fuzz test the smart contract. We report an evaluation of GasFuzzer via an experiment on 3170 real-world smart contracts deployed on the public Ethereum Blockchain between October 2017 and July 2019. The findings show that GasFuzzer with gas-greedy strategy can detect more Exceptions Disorder kind of security vulnerabilities (7 more cases) than the previous state-of-the-art black-box fuzzer, and GasFuzzer with gas-leveling strategy and gas coverage criterion can detect 6 additional cases of Exceptions Disorder security vulnerabilities, which is significant.

INDEX TERMS Blockchain, Ethereum, smart contract, fuzzing, software testing, Fuzzer, security vulnerability, gas consumption, atomicity violation, vulnerability triggering.

I. INTRODUCTION

Since the inception of Bitcoin, the first cryptocurrency that took advantage of decentralization, both the industry and academia are taking interest in the blockchain technology as it was to reach a market capitalization of more than a quarter trillion USD [5] at the time of submission of this work. Ethereum is another major decentralized platform, which not only allows transactions with tokens but also offers storage and execution of the code, known as *smart contracts*. Smart contracts can be written in a high-level procedural language named *Solidity*. In addition to Solidity, other languages such as Serpent and LLL are also available, however these have not been as popular in comparison to solidity. In this work, we will use the term *smart contract* exclusively for an Ethereum smart contract (written in Solidity for illustration purpose) and blockchain for an Ethereum blockchain.

A key feature of Ethereum is that it uses a mechanism of gas allowance to constrain each *external call* to any smart contract (i.e., a *transaction* in Ethereum terminology) to execute within a given gas allowance, where each execution of any instruction consumes a certain amount of gas. This inherent reliance on gas consumption and allowance to execute code makes the execution of smart contracts different from the execution of traditional programs (e.g., Java programs) on traditional platforms in terms of control

The associate editor coordinating the review of this manuscript and approving it for publication was Lo'ai A. Tawalbeh.

flow, where in executing a smart contract, any executing instruction can be a code location to raise an exception.

In this paper, we investigate whether gas consumption and allowance may play an effective role in exposing exception-oriented security vulnerabilities in smart contracts. Transactions encoded with different input parameters may require different amounts of gas to be executed. Our insight is that by controlling the amount of gas allowance of a transaction and manipulating the parameter values of the function associated with a transaction when the transaction is issued, we can indirectly select not only a particular function but also a particular statement in the function (inside the function call sequence induced by the transaction) to raise the first *out-of-gas exception*. This allows a new kind of testing technique to be developed to (fuzz) test how well those gas-oriented exceptions are handled in selected statements and functions and to what extent the exceptions are back-propagated to the corresponding callers along the function call sequence.

In this paper, we present **GasFuzzer**, which is the *first* work to our knowledge that manipulates gas allowance of a transaction to exploit security vulnerabilities through the dimension of gas consumption and allowance. We also note that the idea of gas allowance manipulation is general, and it is orthogonal to techniques that impose no particular constraints on gas allowance and gas consumption.

The basic idea of GasFuzzer is as follows: Like typical fuzzers [1], [28], GasFuzzer starts with a pool of seed transactions. It assigns the same amount of energy to every such transaction. Given a transaction in a seed pool of transactions, GasFuzzer mutates it with two original strategies: *gas-greedy* and *gas-leveling*.

In the **gas-greedy strategy**, the input parameters of the given transaction are mutated to produce a mutated transaction. The gas consumption of the mutated transaction is collected. If the mutated transaction consumes more gas than the given transaction, it is placed into the seed pool for potential further mutation and the energy of the given transaction is reduced according to a power law.

In the **gas-leveling strategy**, the seed pool is firstly filled with a small number of transactions consuming most gas generated by the gas-greedy strategy. They form a sequence of transactions. From the sequence, transactions are randomly picked for mutation. For every picked transaction, GasFuzzer divides the gas consumption of the picked transaction into a number of intervals, mutates the gas allowance of the picked transaction to fall within a randomly selected interval, and substitutes the original picked transaction by the mutated transaction. It applies the resultant sequence of transactions to test the corresponding smart contract. GasFuzzer also includes a novel coverage-based test data adequacy criterion (referred to as the **gas coverage criterion**) to terminate fuzz testing: It repeats the process until the overall coverage on the set of the above-mentioned gas allowance intervals of these picked transactions has reached a predefined threshold.

In the experiment, we collected the a set of 3170 real-world smart contracts from Etherscan [9] deployed between Oct 2017 and Jul 2019 as our smart contract dataset. The empirical results show that, the gas-greedy strategy detects 28% more *Exceptions Disorder* security vulnerabilities than ContractFuzzer [17] (the current state-of-the-art in black-box fuzzing). It was also found that both techniques had similar security vulnerability detection ability on other kinds of security vulnerabilities detectable by the full set of ContractFuzzer's test oracles in the experiment. Furthermore, the gas-leveling strategy detected 6 additional smart contracts incurring *Exceptions Disorder* vulnerability (additional increase of 24% over ContractFuzzer), which is significant because they are real bugs. We also observe that some gas-related security vulnerabilities can only be detected under certain pre-conditions, which we will report in Section V.

This work makes the following contributions:

1. This paper is the first work that proposes gas allowance and consumption as a guiding dimension to provide feedback for smart contract fuzzing.
2. It presents a novel technique, called GasFuzzer, to realize the above proposal and shows its feasibility by implementing it as a tool. GasFuzzer also includes a novel gas-leveling strategy and a novel coverage-based test data adequacy criterion.
3. It presents the first empirical study that compares black-box fuzzing (ContractFuzzer) and gas consumption driven fuzzing (GasFuzzer) for security vulnerability detection on Ethereum smart contracts. It shows the effectiveness of GasFuzzer.

The rest of this paper is organized as follows. Section II introduces a running example while section III provides a background on EVM, smart contracts, gas architecture and fuzz testing. Section IV is an overview of ContractFuzzer. Section V presents our proposed method, GasFuzzer which is built on top of ContractFuzzer. We present our experiments and results in Section VI. Related work is discussed in Section VII and finally, Section VIII concludes this work.

II. RUNNING EXAMPLE

A simplified scenario of interactions between two smart contracts tokenHolder and txManager has been presented in Fig. 1. The function receiveToken() receives an unsigned integer t as input and updates a storage variable token. receiveToken() also uses the address m of contract txManager to call the function manageTx() that increments the storage variable tx by 1 to maintain its transaction count. Unlike a variable labeled as ''memory'', a storage variable in Ethereum will cause the storage area of the blockchain to permanently keep the value assigned to that variable, provided that the transaction is executed successfully. In Ethereum, owing to the need for blockchain space to keep a value for a storage variable in a smart contract, the gas consumptions to keep different values may not be identical. For instance, suppose that the smart

TABLE 1. Gas consumption of tokenHolder.receiveToken(t).

Index	Transaction	Gas Allowance	Transaction Cost	Transaction Status	Value of tokenHolder.token		Value of txManager.tx		Vulnerability Triggered
					Expected	Actual	Expected	Actual	
t_1	receiveToken (0)	80,000	45,992	Successful	0	0	1	1	No
t_2	receiveToken (7)	80,000	50,256	Successful	7	7	2	2	No
t_3	receiveToken (6)	80,000	35,256	Successful	6	6	3	3	No
t_4	receiveToken (0)	80,000	20,192	Successful	0	0	4	4	No
t_5	receiveToken (4)	80,000	50,256	Successful	4	4	5	5	No
t_6	receiveToken (7)	35,000	**34,993**	Successful	7	7	**6**	**5**	**Yes**
t_7	receiveToken (3)	20,000	**20,000**	**Failed**	**3**	**7**	**6**	**5**	No

```
contract tokenHolder {
  uint token;

  function receiveToken (uint t) public {
    token = t;
    address m = 0x1c..A4      //address of txManager
    m.call(abi.encodeWithSignature("manageTx()"));
  }
}

contract txManager {
  uint tx;

  function manageTx () public {
    tx = tx + 1;
  }
}
```

FIGURE 1. An exemplified smart contract written in Solidity.

contracts shown in Fig. 1 are newly deployed on a blockchain. Table 1 shows the gas consumption (i.e., transaction cost) of each transaction (i.e., each call) in the calling sequence to receiveToken(t) with t = 0, 7, 6, 0, 4, 7 and 3. The gas allowance provided for each transaction is listed in the second column of Table 1. Considering the first five transactions, by calling receiveToken(0), token is set to 0 in the 1st and 4th calls, and the gas consumptions of these function calls are considerably less than invoking the same function with a non-zero parameter. On the other hand, updating the value of the storage variable from zero to non-zero consumes more gas than all the other cases, e.g., receiveToken(7) and receiveToken (4).

When a transaction is issued to a blockchain for execution, the transaction must come with a gas allowance that implicitly constrains the total number of execution steps allowed to complete the function call. The transaction can only be completed before exhausting this gas allowance.

The last two rows in Table 1 summarize transactions where their gas allowances are less than those of the first five transactions. receiveToken(7) and receiveToken(3) are allowed with gas limits of 35,000 and 20,000 respectively. Similar to the transaction receiveToken(6), the full execution cost for receiveToken(7) should also be 35,256. However, this transaction was only allowed to spend 35,000 units of gas to execute. As such, not every instruction can be completed: the value of token variable in tokenHolder is updated from 4 to 7, but the value of tx remains 5 without any exception being reported. If the gas allowance is further reduced to 20,000 units with transaction receiveToken(3), no variable in these two contracts is updated.

The reason for the first case in the last paragraph is that the statement tx = tx + 1; has an insufficient gas allowance to be executed, and thus, the function call on manageTx() aborts the update of tx and results in an exception. Nonetheless, its calling function (i.e., receiveToken()) neither catches the exception nor determines whether the effect of calling manageTx() has been properly in place. It goes on completing its execution. Therefore, the state of the smart contract becomes inconsistent and the transaction is marked as successful. In the second case in the last paragraph, the amount of gas allowance is small enough that it even cannot complete the update of the variable token. This type of bug can also be viewed as a breach of the atomicity region (i.e., an atomicity violation that raises through an exception which corrupts the memory state). We will use this running example in illustrating GasFuzzer.

III. BACKGROUND

In this section, we present an overview of Ethereum. After that, the Ethereum Virtual Machine (EVM), smart contracts, and gas architecture of Ethereum are described. In the last subsection, an overview of typical fuzz testing is provided.

A. ETHEREUM VIRTUAL MACHINE

The Ethereum Virtual Machine (EVM) is the platform for all smart contracts to be deployed, maintained and executed in a decentralized architecture. It is the only execution environment for Ethereum smart contracts to carry out their operations.

EVM is a clean stack-based implementation and a light-weight execution environment. Each element on the stack consists of 256 bits and is also referred to as a word. EVM is responsible to handle all the state changes that happen to the blockchain in accordance with the predetermined execution phases and environment, e.g. exception handling, transaction reversion and verification of jump target locations.

EVM performs operations on the bytecode of a smart contract after compilation and deployment on a blockchain. It handles tasks such as running the bytecode, computing and keeping a record of the amount of gas consumed/remaining and halting the execution once all the gas offered has been consumed (including but not limited to throwing an out-of-gas exception). In case of a successful state

transaction, all the remaining gas is returned to the caller of the transaction. Further details can be found in [8].

B. SMART CONTRACTS

Smart contracts are programs. A simplified source code listing for a smart contract is shown in Fig. 1. Once compiled successfully, a smart contract's bytecode can be deployed on a blockchain. After a successful deployment, the bytecode of the smart contract is visible publicly and the functions can be invoked. Similarly, every transaction calling the public function of the smart contract is also publicly visible.

If somehow, a malicious user manages to execute the public functions of a deployed smart contract in such a manner that renders the smart contract in a state that it is not designed to handle, unintended consequences can arise. The malicious user may exploit such loopholes in the code to carry out attacks such as locking all the digital tokens (e.g., ether or other user-defined cryptocurrency, or data) inside the contract or getting the tokens from the smart contracts which should not happen according to the intention of the smart contract developer.

Any smart contract deployed on a Ethereum blockchain is immutable and its address cannot be allocated to another smart contract. Hence, if a smart contract has to upgrade its version to handle some issues (such as logical bugs or security vulnerabilities), additional deployments need to take place which increase execution costs. Forwarding the transaction from an older version to a newer version on each transaction received by the old smart contract, or transferring the data from the older version to the newer version are the remedies that are often used in such scenarios. But even after the application of such remedies, transferring of data from older versions to newer version remains a problem.

Effective and efficient techniques to find security vulnerabilities before the deployment of new smart contracts is highly desirable. In this sense, even for fuzz testing, one should not target at having a one-size-fit-all technique to expose higher average security vulnerability instances (or bug locations). In Section V, we will present GasFuzzer which targets detecting gas-related exception security vulnerabilities.

C. THE GAS ARCHITECTURE

When initiating a transaction, the initiator (i.e., the caller of the transaction) has to define a gas allowance to pay for running the transaction. The EVM will deduct a certain amount of gas from this given amount of gas allowance after every execution step. Any transaction that exceeds the given gas allowance will be reverted, and all the gas consumed is transferred to a miner (if using proof-of-work consensus protocol). A miner may choose to include or exclude a transaction in its computational task of the required consensus protocol. In the current state of the practice, a transaction with a higher unit gas price has an advantage over a transaction with a lower unit gas price to be processed earlier, and a transaction with a very low unit gas price may never be processed.

D. FUZZ TESTING

There are many different kinds of fuzzing techniques used in the past. For blackbox fuzzing, random inputs are generated to test applications without any knowledge of the implementation of the system. In general, fuzzing is often started by providing a small set of seed inputs and then incrementally and randomly mutating them to generate new inputs without referring to the application details. An example is ContractFuzzer [17], which we will review in Section IV.

To make the fuzz testing process program-aware, grey-box fuzzing such as AFL [28] and AFL Fast [1] have been proposed. Grey-box fuzzers generally follow the methodology depicted in Fig. 2. A program is provided to the fuzzer along with some seed inputs. Such an input is executed on an instrumented version of the program to mutate inputs from feedback such as whether new code-based artifacts (e.g., new branch or new branch subsequences) have been discovered by the applied input. These mutations can be random or guided by heuristics such as the frequency that an input has previously been used for mutation. The program under test is executed on a mutated input, and if there is an increase in path coverage, the mutated input is added to the original input queue making it eligible for further mutation. Harvey [27] is an example of a grey-box smart contract fuzzer that uses input prediction to improve coverage on program paths.

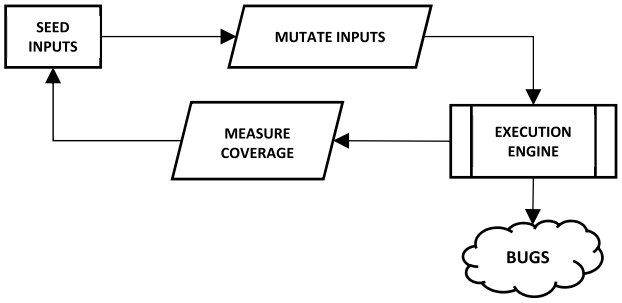

FIGURE 2. Typical workflow of grey-box fuzzer.

IV. CONTRACTFUZZER

ContractFuzzer [17] is the state-of-the-art in blackbox fuzzing for smart contracts. It generates test inputs based on the Application Binary Interface (ABI) of a smart contract under test and formulates a suite of seven test oracles to detect Gasless send, Exceptions disorder, Re-entrancy, Timestamp dependency, Block Number Dependency, Dangerous Delegateccall and Freezing Ether security vulnerabilities. An instrumented version of the EVM is used to keep track of the behavior of smart contracts over randomly generated transactions. To support some test oracles, ContractFuzzer also generates additional contracts when fuzz testing smart contracts.

GasFuzzer to be presented in Section V is built on ContractFuzzer. Therefore, we review ContractFuzzer in greater detail. ContractFuzzer consists of two

Algorithm 1 ContractFuzzer Algorithm

Input:
list of smart contracts to test ($c_i \in C$)
Output:
tuple $M < c_i$, vulnerability $v_i >$

```
1 M = ∅
2 for each c_i in C
3   while ¬ (timeout)
4     Generate a Random transaction (t_j)
5     c_i.Execute(t_j)
6     if(vulnerability)
7       add < c_i, v_i > to M
8     end if
9   end while
10 end for
```

sub-components. One is an offline instrumented EVM and the other is an online fuzzer. The EVM instrumentation component is responsible for instrumenting the EVM for enabling the fuzzer to be able to examine the execution of smart contracts and retrieve data for the discovery of bugs. The fuzzing process begins with exploring the bytecode of a smart contract using static analysis and an ABI analysis. In this phase, the data types of ABI parameters, the addresses of the smart contracts, and signatures of functions in these smart contracts are obtained. ContractFuzzer analyzes the ABI signatures of the deployed smart contracts from the blockchain. After performing these two tasks, a phase of random input data generation begins. These generated inputs conform to the ABI specifications. The fuzzing process of ContractFuzzer has been shown in Algorithm 1.

What ContractFuzzer does is: For each contract c in the pool of contracts needed to test, ContractFuzzer extracts a set F_c of all the public functions $\{f_1, f_2, f_3 \ldots f_m\} \in F_c$ along with the data type of each input parameter that is required by each function. To achieve this, ContractFuzzer utilizes the ABI of each smart contract from where all the public functions and their input types can be identified, i.e., $\{i_1, i_2, i_3 \ldots i_n\} \in f$ where $f \in F_c$. Once all the necessary information to generate a transaction sequence is obtained, ContractFuzzer randomly assigns a value to each input parameter i (for $i = 1 \ldots n$) of f and constructs a transaction to represent an invocation of f with these parameter values. A chain of such transactions is then applied to the blockchain to test the set of deployed smart contracts in the blockchain.

The instrumented version of EVM analyzes the execution traces of all the invoked smart contracts through its implemented test oracles. For brevity, we do not review the set of test oracles and how they are formulated in ContractFuzzer. Interested readers may refer to the work of Jiang *et al.* [17].

V. OUR PROPOSAL: GAS-AWARE FUZZING

In this section, we present GasFuzzer. It includes two strategies to increase the effectiveness of security vulnerability detection. In the gas-greedy strategy, GasFuzzer tends to prioritize transactions that consume more gas than others for input parameter mutation. The insight behind this strategy is that if a transaction consumes more gas, it indicates that more opcodes are likely to have been exercised or more important blockchain-related operations have been performed. In the gas-leveling strategy, the gas allowance for gas-expensive transaction is mutated with the aim of assessing whether exceptions generated due to gas unavailability have not been properly back-propagated to preceding function calls in the call chain. To the best of our knowledge, we are not aware of similar strategy as our gas-leveling strategy in the literature. Moreover, the gas coverage criterion is the first black-box coverage criterion proposed to facilitate gas-leveling strategy in testing smart contracts. The remainder of this section describes these strategies in detail.

A. GAS-GREEDY STRATEGY

An overview of how GasFuzzer fuzz-tests smart contracts with the gas-greedy strategy is depicted in Fig. 3. In this strategy, GasFuzzer initially generates transactions randomly, similar to ContractFuzzer, and executes the smart contracts under test with these transactions. These transactions are then mutated and sent to the blockchain for execution. If these newly generated transactions consume more gas, they are added to the inputs queue for further possible mutations. This process is allowed to take place until all the testing time has been used. Finally, the execution logs are analyzed for security vulnerability detection.

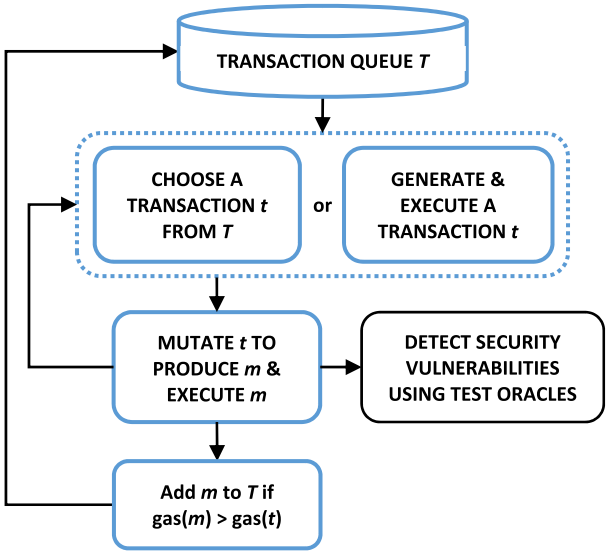

FIGURE 3. An overview of GasFuzzer's Gas-Greedy strategy.

Algorithm 2 presents the gas-greedy strategy of GasFuzzer. A list of smart contracts $c_i \in C$ under test is provided along with a set of seed transactions $t_i \in T$ for each function with *priority* 1 in each smart contract c_i in C. Seed prioritization factor θ (which is in the range of 0 to 1) and a threshold ρ to choose between generating a new transaction or selecting one from the queue are also provided.

Algorithm 2 GasFuzzer Algorithm for Gas-Greedy Strategy

Input:
list of smart contracts to test ($c_i \in C$)
list of seed transactions: $\langle c_i, t_j, gas_j, priority{:}1\rangle \in Q$
seed prioritization factor: θ
new transaction generation threshold: ρ
Output:
tuple $M\langle c_i$, vulnerability $v_i\rangle$
list of transactions and gas: $\langle c_i, t_j, gas_j\rangle \in Q_{out}$

1 $M = \emptyset$
2 $Q_{out} = \emptyset$
3 **for** each contract c in C
4 $Q_c = \{q \in Q | q.c_i = c\}$
5 **while** ¬ (timeout)
6 f_i = select a function from c
7 **if** random() $< \rho$
8 t = Generate a random transaction for f_i
9 $gas = c$.Execute(t)
10 **else** t = choose a transaction for f_i from Q_c
11 **end if**
12 t_m = mutate(t)
13 $gas_m = c$.Execute(t_m)
14 **if** ($gas_m > gas$)
15 $Q_c = Q_c \bigcup \langle c, t_m, gas_m, 1\rangle$
16 $Q_c(t).priority = Q_c(t).priority * \theta$
17 **end if**
18 **end while**
19 **for** each Security Vulnerability detected
20 $M = M \bigcup \langle c, v_i\rangle$
21 **end for**
22 $Q_{out} = Q_{out} \bigcup Q_c\ (c, t_j, gas_j)$
23 **end for**

The output is a list of tuples $\langle c_i, v_i\rangle$ that contains a detected security vulnerability v_i for the vulnerable smart contract c_i. (We note that seed transactions can be obtained from ContractFuzzer.) For each public or external (in the context of smart contracts written in Solidity) function in c_i, a list of input parameters is obtained randomly and then represented as a transaction. Two sets Q_{out} and M are initialized as empty sets (lines 1-2). In lines 3-4, a smart contract c_i is chosen from C and transactions relevant to c_i are extracted from Q into Q_c. Then, a loop is initiated which selects a function f of c (lines 5-6) and in each iteration, a transaction t_j is either obtained from Q_c or generated randomly based on the parameter ρ (lines 7-11). If a new transaction is generated, its gas consumption (gas_j) is recorded after executing it through the blockchain (line 9). After that, t_j is mutated to obtain a mutated transaction t_m (line 12), which is then executed to get its gas consumption gas_m (line 13). In case that the gas consumption of t_m is greater than that of t_j, the mutated transaction t_m is added to Q_c along with its gas consumption gas_m and *priority* of 1 (lines 14-15). The *priority* of the chosen transaction t_j is reduced for further selection by multiplying it with θ (line 16). Any security vulnerabilities found by the test oracles are recorded in set M (line 20). In the end, all the transactions in Q_c are added to Q_{out} for usage in the next phase of gas-leveling.

In comparison to ContractFuzzer, GasFuzzer, with this strategy, keeps a record of gas consumption for each selected transaction. It mutates transactions in the inputs queue to change input parameters in a manner that tends to increase gas consumption and employs mutation operators similar to those used in [28]. However, unlike traditional gas-unaware fuzzers, mutations in GasFuzzer are only applied to transactions that call functions requiring input parameters to be provided for successful execution. Moreover, as presented in [20], seed prioritization is important in making fuzz-testing increasingly cost-effective. A seed prioritization scheme to improve the diversity of input transactions has also been adopted in GasFuzzer. We further explain these strategies in Section VI.

Consider the exemplified smart contract presented in Fig. 1 in the context of Algorithm 2. For the tokenHolder smart contract, transaction at index t_1 in Table 1 is considered as the seed transaction with priority 1 in Q_c (line 4). This transaction is picked up as t for mutation (line 10) and input 0 is mutated to 7 (i.e., t_2 from Table 1) producing t_m (line 12). The transaction receiveToken(7) increases gas consumption from 45,992 to 50,256 and thus according to line 14, t_m is added to Q_c with priority 1 (line 15) while reducing the priority of t by a factor of θ (line 16). In the next iteration, a random transaction (t_3) is generated (line 8) which is receiveToken(6) and its gas consumption is recorded to be 35,256 (line 9). This transaction is also mutated and input 0 is generated which reduces the gas consumption and does not get included in Q_c. Security vulnerability analysis is performed in the end using test oracles to analyse the execution logs.

B. GAS-LEVELING STRATEGY

In the gas-leveling strategy, as depicted in Fig. 4, the gas allowances of gas-expensive transactions are manipulated to find out if an insufficient gas allowance leads to storage changes in the blockchain that should not be taking place, as described through our example in Fig. 1. It is also important to point out here that only successful transactions, the transactions that do not revert or discover a security vulnerability, from gas-greedy strategy were considered for the gas-leveling strategy to extract the gas-expensive transactions in the experiment. If all transactions were to be considered, then transactions that have already triggered a security vulnerability in the first strategy could result in producing duplicate results.

Algorithm 3 explains the gas-leveling strategy of GasFuzzer. The set of transactions from the gas-greedy fuzzing (Q_{out}) are provided for each contract along with values γ, k, m and ε which are the number of expensive transactions to extract, number of sections to divide gas consumption, number of transaction for gas allowance

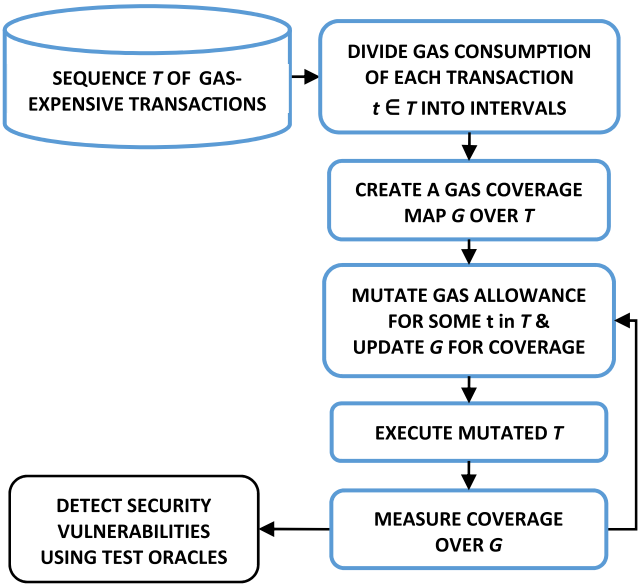

FIGURE 4. An overview of GasFuzzer's Gas-Leveling strategy.

manipulation and coverage threshold respectively. In the beginning, an empty set M of security vulnerabilities is initialized (line 1). Then a contract c is chosen for testing and relevant transactions for c are extracted from Q_{out} to Q_c (lines 2-3). After this, a queue of gas-expensive transactions Q_e is initialized and top γ gas-expensive transactions for each function f are extracted into it (lines 4-9). For this purpose, an empty queue is initialized (line 4), followed by a function-wise iteration over the contract c (line 5). In each iteration, transaction for a function f are extracted into Q_f (line 6) and a *cutoff* point for gas-consumption is decided by first choosing the top γ gas-expensive transactions and then taking the minimum gas allowance among them (line 7). Each transaction in Q_f that has a gas-consumption greater than *cutoff* is extracted into Q_e (line 8).

As stated in Section I, GasFuzzer includes a novel coverage-based test data adequacy criterion (referred to as the gas-coverage), which is as follows: A gas coverage map G is created for each transaction in the sequence Q_e over k gas intervals (lines 10-11). Lines 13-20 iterate over the transaction sequence Q_e by mutating allowances for m transactions in each iteration which transforms Q_e into Q_m. G is updated in each iteration for gas interval coverage (line 17) and the mutated transaction sequence Q_m is executed (lines 19-20). The loop executes until a coverage of ε is achieved over G. At the end, the execution log is analyzed for detecting security vulnerabilities which are added to M if detected.

Consider the same exemplified smart contract from Fig. 1 but this time in the context of Algorithm 3. Suppose that Q_{out} in Algorithm 3 contains all the transactions from gas-greedy strategy. The transaction sequence Q_c contains transactions t_1 to t_5 from Table 1., 2 transactions (value of γ), in the context of function **receiveToken(t)**, that consume the most gas are used to form a sequence Q_e containing transactions t_2 and t_5 from Table 1 (i.e., $\langle \ldots, (t_2, 50256), (t_5, 50256), \ldots \rangle$). It needs to be pointed out that Q_e will contain transactions for various functions but we are focusing only at one function for the sake of brevity here. Next, a gas coverage map G is created. To fill G, each g_i is divided into 3 intervals (value of k). The gas map G is a 2-dimentioanl array with each entry consisting of the input data and 2 gas values (corresponding to the interval with a high and low value) as the key and a boolean (cov) to be the value that keeps a record if this particular interval has been covered or not. For instance, the entries corresponding to $(t_2, 50256)$ in Q_e will be of the form $\langle (t_2, 0, 16752)=\text{✖}, (t_2, 16752, 33504)=\text{✖}, (t_2, 33504, 50256)=\text{✖} \rangle$. The ✖ symbol signifies that a particular section has not been covered and its value is false. Next, Q_e is mutated into Q_m by modifying the gas allowance for some (value of m) transactions. For each transaction in Q_m that

Algorithm 3 GasFuzzer Algorithm for Gas-Leveling Strategy

Input:
list of smart contracts to test ($c_i \in C$)
list of transactions: $\langle c_i, t_j, gas_j \rangle \in Q_{out}$
number of gas expensive transaction for each function: γ
number of section to divide gas consumption: k
number of transactions for gas manipulation: m
coverage threshold: ε
Output:
tuple $M\langle c_i$, vulnerability $v_i \rangle$

1 $M = \emptyset$
2 **for** each contract c in C
3 $Q_c = \{q \in Q_{out} | q.c_i = c\}$
4 $Q_e = \emptyset$
5 **for** each function f in c
6 $Q_f = \{x \in Q_c | x.c.f_i = f\}$
7 cutoff = minimum(top $\gamma(Q_f.gas_j)$)
8 $Q_e = Q_e \bigcup \{x \in Q_f | x.gas_j \geq$ cutoff$\}$
9 **end for**
10 **for** $i = 0$: k-1
11 $\forall q \in Q_e, G[q.t, l = \lfloor (x.gas_j)/k \rfloor * i,$
$h = \lfloor (x.gas_j)/k \rfloor * (i+1)] = \langle cov$: ✖$\rangle$
12 **while** ¬(coverage over $G > \varepsilon$)
13 $Q_{mut} = Q_e$
14 μ = Choose m indices randomly from 1:$|Q_e|$
15 **for** i =1:m
16 $Q_{mut}[\mu[i]].gas$= mutate($Q_{mut}[\mu[i]].gas$)
17 $G[Q_{mut}[\mu[i]].t, h \geq Q_{mut}[\mu[i]].gas,$
$l < Q_{mut}[\mu[i]].gas] = \langle cov$:✔ $\rangle$
18 **end for**
19 **for** $i = 1$:$|Q_{mut}|$
20 c.Execute($Q_{mut}[i]$)
21 **end while**
22 **for** each Security Vulnerability detected
23 $M = M \bigcup \langle c, v_i \rangle$
24 **end for**
25 **end for**

gets modified, for example (t_2, *50256*) to (t_2, *35000*), the corresponding entry in G for t_i, where g_m falls within the interval is updated and boolean *cov* is set to ✔ (signifying that the value is true). This corresponds to the last interval that we stated above which now will look like (t_i, 33504, 50256)= ✔. Then Q_m is executed by GasFuzzer and execution logs are analyzed in the end where the *Exceptions Disorder* security vulnerability in t_6 is successfully detected. This process is repeated until a coverage of ε is achieved over G.

VI. EVALUATION

In this section, we present an evaluation of GasFuzzer. We first describe our dataset, experimental setup and procedure. Then in the experiment, we aim to answer the following three research questions:

RQ1: Is the gas-greedy strategy of GasFuzzer more effective than ContractFuzzer in exposing Exceptions Disorder Security Vulnerability?

RQ2: Is the effectiveness of the gas-greedy strategy of GasFuzzer at par with ContractFuzzer in exposing other types of security vulnerabilities?

RQ3: Can *Exceptions Disorder* vulnerability detection be further improved by GasFuzzer through its gas-leveling strategy with the use of gas-coverage criterion?

In the end, we discuss the threats to validity and future work.

A. DATASET

To set up our experiment, we obtained a dataset of 3170 smart contracts from Etherscan.io [9] deployed on Ethereum Main-Net between October 2017 and July 2019. The standard of smart contracts has evolved quickly. The dataset of smart contracts has been made available online.[1]

In recent two years, a limit had been imposed by Etherscan.io[2] on the number of smart contracts that can be downloaded. To operate under this constraint, we searched for verified contracts [10] with popular keywords in their names and then extracted their Solidity source code, bytecode, ABI and constructor arguments to be used in our experiments.

From Table 2, on average, each contract in our dataset consists of around 158 lines of code with the median being 134. Lower and upper quartiles are at 53 and 196 with the largest of contracts reaching almost 1500 lines of code. Each contract on average has 23 functions where upper and lower quartiles are 8 and 41 respectively. The number of *pure* [3] functions is much lower as compared to total number of functions which indicate that most functions in these contracts affect the state of the blockchain. Each contract on average makes about 4 calls to other contracts. Median, lower and upper quartiles are 2, 0 and 6 respectively. Mean number of *payable* [4] functions per contracts is 1.57 with median being 1. Lower and upper quartiles are 0 and 2 respectively.

TABLE 2. Descriptive statistics of our dataset.

Feature	Mean	Median	Quartiles	
			Lower	Upper
Lines of code	157.5	134	53	196
No. of Functions	23.1	21	8	41
Pure Functions	3.75	4	1	7
Message Calls	3.89	2	0	6
Payable Functions	1.57	1	0	2

B. SETUP OF GASFUZZER

We ran our experiment on a desktop computer that was equipped with 64GB of RAM, an 8-core Intel Xeon 2.2 GHz processor with Ubuntu 18.04 running. Our experimental setup was inspired by ContractFuzzer so we also used the same instrumented GETH client (version 1.7.0) to interact with a private Ethereum Blockchain. We deployed all the smart contracts in our dataset to this private chain (i.e. our TestNet). The mining difficulty is set very low so that our transactions could be mined easily and without any unnecessary delay. We carried out our experiment with only one miner since none of the smart contract security vulnerabilities that we were considering should have any impact due to different numbers of miners.

C. EXPERIMENTAL PROCEDURE

1) PRE-PROCESSING

GasFuzzer started with a phase of light-weight static analysis of each smart contract in the dataset. This helped in performing realistic message calls upon real contracts from the Ethereum Main-Net. In contrast to ContractFuzzer, which provided random inputs from a pre-written file for address type inputs, our implementation of GasFuzzer provided address inputs of actual smart contracts deployed on the TestNet and these contracts were downloaded from the actual Ethereum Main-Net in an automated manner. To accomplish this task, a light-weight static analysis is performed upon the binary files of smart contracts in the dataset. During this process GasFuzzer looked for function signatures that appeared right after an external message call to other contracts. Once a set of these function signatures was obtained, GasFuzzer searched on the Main-Net for any contracts containing functions that matched with the function signatures in these message calls. Fortunately, a public dataset for Ethereum Blockchain was available on Google BigQuery [13] for GasFuzzer to perform this task automatically. For each function signature identified, GasFuzzer downloaded the bytecodes of 20 latest contracts containing matching functions and deployed them on the TestNet. These smart contracts can be referred to as the dependency contracts. The addresses on which these

[1] https://github.com/TechBeatle/EthereumSmartContractsDataset

[2] https://etherscan.io/

[3] A pure function is a function in a smart contract that does not alter the state of any storage variable.

[4] A payable function is a function in a smart contract that allows it to receive funds in ether.

downloaded contracts got deployed were then provided as options of address type inputs to be chosen for fuzzing inputs wherever an address type of value was required to produce a valid message call. Following the deployment of all the smart contracts in our data set and their dependency contracts, the fuzz testing process was ready to be started.

2) EXPERIMENT DETAILS

We obtained ContractFuzzer from [18] and built on top of it to implement GasFuzzer. Like ContractFuzzer, GasFuzzer also calls each smart contract with the following three types of accounts: One was the account that was used to deploy these contracts, hence making it the owner account. The second one was an Externally Owned Account (EOA) which has never interacted with the contract before. The last one was an Agent account which was an attacking contract to look for any Re-entrancy bugs. The same Attack Agent contract, employed by ContractFuzzer, is used by GasFuzzer as well. However, different from ContractFuzzer, GasFuzzer tested each contract in our dataset on an individual basis. After deploying all the contracts at once, a separate fuzzing process was started for each contract generating separate logs for each smart contract and hence each smart contract's analysis can easily be done on an individual basis.

For each contract in our experiment regarding the gas-greedy strategy, each function was called about 30 times on average with a variety of inputs and a gas allowance of 80,000. As all the smart contracts in our dataset are deployed at the beginning of fuzzing process, the state of each smart contract is set to default and no prior transactions exist for that particular smart contract. Transactions are sent to the blockchain in batches after generation and on average a batch consists of about 30 transactions. These transactions are picked up and mined. We would like to point out here that some transactions never get picked up for mining and in our opinion this problem is related to the Geth version being used. Overall, this number was very small and had no significant effect on the experiments. It is necessary to highlight at this point that experiments conducted for gas-leveling strategy are not yet fully automated. For gas-leveling strategy, a value of 5 is chosen for γ, k and m, while ε is set to 70% (from Algorithm 3).

The test oracles in the ContractFuzzer tool [18] were used to detect security vulnerabilities. In particular, the oracle for Exceptions Disorder checks a call chain for any cases where the root call throws no exception while one of the nested message calls throws one and it was never handled properly.

3) MUTATION STRATEGY

In the gas-greedy strategy, GasFuzzer should mutate transactions. Two operators were implemented for this purpose in the experiment to demonstrate the feasibility of GasFuzzer. The first mutation operator is randomly flipping the bits of an input parameter and the second one is the addition of random bits to the input parameter. Furthermore, these mutations were applied selectively on certain input types. For an address type input no mutation operation was applied and the parameter was provided as it was before. For bools, uints, ints and other fixed length input parameters, only the first mutation operator was applied and for other input parameters such as bytes and strings any one or both of the mutation operators was applied. Even though two basic mutation operators were used in the experiment, the experimental results have already shown that GasFuzzer can be effective.

4) SEED PRIORITIZATION

In the gas-greedy strategy, GasFuzzer needs seed prioritization. It started with assigning a uniform priority to each seed input and every time a seed input was used to generate a new gas expensive transaction, the priority of the original seed was reduced with $\theta = 0.5$ in Algorithm 2. In addition, GasFuzzer kept introducing new randomly generated transactions into the process. Whenever a transaction was to be picked for fuzzing, GasFuzzer chose between extracting a transaction from the seed input queue or generating one from scratch. In the experiment, GasFuzzer was configured to maintain a balance between these two options by giving a 70% probability ($\rho = 0.7$ in Algorithm 2) to generate a new random transaction and the remaining probability (i.e. 30%) for reusing an existing transaction from the seed queue.

D. DATA ANALYSIS

In this section, we report our findings and answer the RQs. We use GF and CF to signify the GasFuzzer and ContractFuzzer implementations, respectively.

1) ANSWERING RQ1

To answer RQ1, we compare the number of Exceptions Disorder security vulnerabilities detected by CF and GF. Table 3 summarizes the results. From the table, GasFuzzer was more effective than ContractFuzzer in the detection of Exceptions Disorder by 28%.

TABLE 3. Summary of exceptions disorder vulnerabilities detected.

Vulnerability Type	No. of Contracts with Vulnerability		Increase
	CF	GF	
Exceptions Disorder	25	**32**	28.0 %

The Exceptions Disorder security vulnerability surfaced when two or more contracts interacted with each other via lower-level message calls, e.g., address.call(). Suppose an EOA initiates a transaction for contract C_1 followed by C_1 interacting with C_2 via a later message call. If an exception occurs at any step in the call chain, then that exception should be passed back to its caller function. However, if a necessary reversion in those contracts does not take place, unexpected states may be resulted.

GF produced transactions that tended to be more gas-expensive than CF. In case a transaction produced an Out-of-Gas Exception, which normally it would not, and that exception had not been handled properly by the involved smart contracts, unexpected states may be achieved as a result. From the results reported in Table 3, GF was shown to be more effective than CF in finding such unhandled cases.

Answer to RQ1: From the experimental results, GasFuzzer is likely more effective in finding Exceptions Disorder security vulnerabilities than ContractFuzzer.

2) ANSWERING RQ2

We compare the effectiveness of GF with CF to see whether there is any significant decline in the effectiveness of GF on other types of security vulnerabilities that CF can detect.

The results in Table 4 show that there are no significant performance degradations of GF in finding other security vulnerabilities detectable by CF. Both GF and CF do not find any cases of Dangerous DelegateCalls or Freezing Ether on our dataset. As for Gasless Send, Re-entrancy and Block Number Dependency, both the tools found the same number of security vulnerabilities. As for the Timestamp dependency, which was very similar to Block number dependency, we observed a slight drop in the number of cases identified by GF. Upon closer inspection of these cases, it was found that the use of block timestamp was done under certain assertion conditions. For example, in a contract named lockEtherPay, hardcoded timestamps were being used to control when Ether transfer can take place. By the time the reported experiments were conducted using GasFuzzer, the end time to allow Ether transfer had passed. Since no Ether transfer could take place, GasFuzzer did not identify this smart contract as vulnerable to Timestamp Dependency.

TABLE 4. Effectiveness comparison of GF and CF on other five types of security vulnerabilities.

Security Vulnerability Type	No. of Vulnerabilities Detected	
	ContractFuzzer	GasFuzzer
Gasless Send	129	129
Re-entrancy	64	64
Timestamp Dependency	252	249
Block No. Dependency	168	168
Dangerous DelegateCall	0	0
Freezing Ether	0	0

Answer to RQ2: We find no significant performance degradations in the ability of GasFuzzer to detect other security vulnerabilities in comparison with ContractFuzzer.

3) ANSWERING RQ3

The Exceptions Disorders vulnerability has been discussed in RQ1. Results from the RQ1 gave us an insight that a fuzzer could further improve its detection ability on this type of security vulnerability by not only manipulation of the input value provided in a transaction but allocating a varying gas allowance may also help affect the security vulnerability detection rate. Our experiments show that manipulating gas allowances for expensive transactions has a positive effect on security vulnerability detection.

TABLE 5. Detected smart contracts with security vulnerabilities in RQ3.

Contract Name	Security Vulnerability
AirDrop	Transfer of assets is not verified in a call to external contract.
fatherContract	Return value of message call to target contract is not verified to be successful.
POGame	Return value of message call to target contract is not verified to be successful.
CoinContract	Return Value of message call to external contract is received but transaction is not reverted if it fails.
GameCoin	Return Value of message call to custom fallback function in an external contract is not verified.
X2ETH	Ether transfer via low-level call is not verified to be successful.

As a result of this experiment, GasFuzzer found 6 additional Exceptions Disorder vulnerabilities and the vulnerable contracts are listed in the Table 5. Most of the smart contracts listed here were unable to ensure whether a message call to an external smart contract had been executed successfully. CoinContract is the exception among these which received the returned value from a call but never verified if the value is false to consequently revert the parent calls as well. Among these contracts, only X2ETH has been self-destructed while others are still live on the Ethereum network which is why the addresses of these smart contracts are not being disclosed in this paper. These results show that certain security vulnerabilities can only be detected under certain pre-conditions which did not necessarily have to be triggered at specific states of the blockchain. It was also found that even initiating transactions with a wrong gas allowance can lead to unexpected results. These security vulnerabilities are very hard to be identified by current fuzzing tools since a high gas allowance is usually reserved in a testing environment.

We also analyzed the execution of the transactions by both GF and CF to look for cases where Out-of-Gas exceptions were thrown due to increased gas requirements. For RQ3, 0.27 million transaction sequences were executed and almost all of the transactions with revised gas allowances threw exceptions except for the cases we reported or maximum gas allowance was provided.

Answer to RQ3: The results show that six new Exceptions Disorder security vulnerability can be identified by GasFuzzer, and all of them are previously unknown real bugs.

E. FURTHER DISCUSSION

From the data analysis reported in the previous sub-section, we believe that gas-related security vulnerabilities were mostly hiding in plain sight, but they can only be exercised under special conditions. The gas allowance assigned to a transaction was critical to detect those security vulnerabilities

```
contract AirDrop {
...

function transfer(address c, address to, uint vs)
public
validAddress(c)
{
  //permanent storage changes

  bytes4 id = bytes4
    (keccak256("transferFrom(address,address,uint)");
  c.call(id, msg.sender, to, vs);
}

...
}
```

FIGURE 5. A simplified version of AirDrop smart contract.

as revealed through answering RQ3 (gas-leveling strategy). A simplified version of the AirDrop smart contract has been provided in Fig. 5 where the function transfer expects two addresses and an unsigned integer. After performing some internal state changes, this function makes an external call to a smart contract deployed at the first input address with a function that matches the signature calculated in id. As explained earlier in Section VI that GasFuzzer initially performs a light-weight static analysis looking for such calls to download and deploy matching realistic smart contracts, a realistic message call can be made. The gas-leveling strategy makes sure that such calls are made with varied gas-allowances so that security vulnerabilities can be identified even if the underlying logic in the target smart contract is error free. In this case, AirDrop smart contract updates its internal storage for some transfer and then an external call is made to transfer some assets to an address. If due to insufficient gas allowance, the message call throws an error-out-of-gas exception which is not checked (such as in this case), an inconsistency will arise.

Without these pre-conditions achieved, it will be quite difficult to hunt out these gas-related security vulnerabilities. Moreover, from our experiments, we observed that both manipulating transactions with higher gas consumption or with a wrongful gas allowance can lead smart contracts to run into dangerous state transitions.

In future, we will further generalize GasFuzzer for it to be able to perform gas-aware fuzzing better by manipulating gas allowance for gas-expensive transactions. We will work on making this process more generic and automated to be able to perform it in an effective manner.

F. THREATS TO VALIDITY

In this section, we present the threats to validity of the experiment.

The evaluation is based on a set of smart contracts deployed in the public blockchain within a particular period. Using other periods will produce a different dataset, and we will see a different number of vulnerabilities detected in the corresponding experiment. However, since we did not know which contracts containing the detected vulnerability, the experiment is still fair in comparing CF and GF. We tend to believe that GF will still detect more Exceptions Disorder cases than CF on other datasets due to its intrinsic ability to distinguish transactions with insufficient gas and/or drive the mutations toward the high end of the gas consumption spectrum.

Algorithms 2 and 3 require some configuration parameters to be initialized. We only evaluated CF and GF on one set of parameters, which already took weeks to complete due to the large amount of transactions produced, and at par with the scale of the experiment reported in the original paper of ContractFuzzer. Having said that, further generalization is necessary.

The experiment used the test oracle of CF. There may be bugs in the implementation and the set of test oracles of CF was limited. The use of other test oracles and their implementations may produce different results.

There may be implementation errors in GF. To alleviate this issue, we have tested GF on a small dataset of self-crafted smart contracts.

The experiment only implemented two mutation operators for GF. We tend to believe that the use of more mutation operations will produce the diversity of mutated transactions. In the literature on traditional mutation testing, the general trend is that a more diverse test suite tends to detect more failures. Apparently, the effectiveness of GF could be further improved if more mutation operators can be used, which requires further experimentation to confirm.

We measured the effectiveness of GF and CF by the number of instances detected by each kind of test oracle. The use of other criterion may produce different results. Moreover, due to the need to mutate transaction and maintain the data structure, GF is less efficient than CF in generating transactions for fuzz testing. One may consider that by allowing the same amount of time budget for either tool to test the same smart contracts, CF will generate more transactions than GF. In the current experiments, we set the same timeout limit to run both tools.

VII. RELATED WORK

Oyente was one of the first tools aimed at smart contract verification proposed by Luu *et al.* in [21]. Oyente is a symbolic execution based smart contract verification tool that uses control flow graphs (CFGs) of smart contracts under test to perform symbolic execution on them. Oyente looks for any vulnerable patterns that may lead to the discovery of some types of security vulnerabilities that include transaction order dependency, re-entrancy and timestamp dependence. Using similar techniques based on Symbolic Execution, another tool called MAIAN was introduced in [23]. MAIAN looks to find out whether a contract can be classified as greedy (contracts that can be made to transfer ether to an address it never transacted with), prodigal (contracts that tend to lock Ether under certain state conditions), suicidal (contract whose code can be removed from the blockchain by address that do not own it) or a combination of these. A major drawback of both these approaches is a high number of false positives. In [16], Jiang *et al.* present Artemis, which is an improved smart contract security vulnerability detection

tool that is based on Oyente. Artemis can effectively find out four types of security vulnerabilities such as Freezing Ether, Block number dependency, Expensive Fallback and Dangerous Delegatecall.

Securify [26] and Vandal [2] are both static analysis based smart contract verification tools. Securify establishes security patterns in a domain-specific language that are then verified for accordance or defiance. For contract analysis, a stackless static-single assignment form of the bytecode is used to deduce predefined semantic facts (data-flow/control-flow dependencies). Vandal, on the other hand, decompiles the bytecode and collects features of the smart contract under test as a datalog. These tools perform well on re-entrancy vulnerability.

ZUES [19] and SmartCheck [25] also try to verify smart contracts for security vulnerabilities but require the provision of source code. ZEUS uses a Solidity based Abstract Syntax Tree (AST) to gather policies which can be edited by smart contract developers. Different from ZEUS, SmartCheck converts Solidity code into Intermediate Representation (IR) which is XML based. This IR is checked for patterns, violation of which leads to detection of security vulnerabilities. Both these approaches operate on a wide range of security vulnerabilities but loose accuracy if properties are not well defined.

Echidna [7] and Harvey [27] use Fuzzing to concretely verify smart contracts. Echidna needs oracles to be written by developers inside unit tests which again puts the strain on contract developers. Harvey on the other hand uses classic Greybox Fuzzing techniques like AFL [28] and AFL-Fast [1] to generate inputs but with a modification of input prediction. ContractFuzzer [17] is a black-box fuzzer that generates random transactions to find security vulnerabilities in Ethereum smart contracts. It uses the Application Binary Interfaces (ABIs) of the smart contracts to generate transactions without any feedback from the execution and employs an instrumented EVM to execute these transactions. Execution logs are then analyzed for security vulnerability detection. In [15], He *et al.* propose a fuzz testing tool which tries to behave like a symbolic execution engine for smart contracts. Imitation learning is used to train a fuzzer on large number of inputs generated through a symbolic execution engine proposed in [24]. The fuzzer is basically a set of neural networks which had been trained on the dataset of generated transactions. A recent work by Nguyen *et al.* [22] presents an adaptive fuzzer which applies a light-weight multi-objective strategy to target difficult to reach branches in Ethereum smart contracts. EvmFuzzer [12] is also a tool that uses fuzz testing, not to verify smart contracts for security vulnerabilities, but to identify discrepancies among various implementations of EVM in different programming languages.

The only previous works that discuss gas consumption as an important entity are Gasper [3] and GasReducer [4] but these tools only aim at finding out patterns in smart contracts that lead to gas wastage. The focus here is not to consider gas consumption for security verification but to reduce the amount of gas consumed to make the process more cost-efficient. These techniques construct CFGs form bytecode to perform symbolic execution employing an SMT solver to discover possible execution paths.

VIII. CONCLUSION

In this paper, we have presented a novel technique GasFuzzer. It consists of two strategies. The gas-greedy strategy has been formulated based on the insight that gas consumption of executed transactions provides lightweight information about the executed program code to deal with blockchain states of the involved smart contracts. GasFuzzer used this aspect of information to iron out transactions subject to further generation of mutated transactions. The experiment has shown that GasFuzzer can detect more Exceptions Disorder security vulnerabilities than the previous black-box state-of-the-art technique ContractFuzzer by 28% while it does not compromise the ability to detect other kinds of security vulnerabilities. The gas-leveling strategy is novel in that it formulates a novel test data adequacy criterion and uses it to guide the generation of mutated transactions with lower gas allowances. The experiment has shown that this strategy is effective in detecting Exceptions Disorder security vulnerabilities that have been missed to expose in the experiment above. Through this work, we believe that by focusing on gas-expensive transactions and manipulation of gas allowance, one can significantly improve the fuzz testing process for some of the most serious security vulnerabilities that can be induced in smart contracts. We plan to further explore along this research direction in the future. A version of GasFuzzer has been deployed in FUSE, an online fuzz testing service for Ethereum smart contracts under the HKSAR ITF (project no. ITS/378/18).

REFERENCES

[1] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Vienna, Austria, Oct. 2016, pp. 1032–1043.

[2] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018, *arXiv:1809.03981*. [Online]. Available: http://arxiv.org/abs/1809.03981

[3] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, Klagenfurt, Austria, Feb. 2017, pp. 442–446.

[4] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *Proc. 40th Int. Conf. Softw. Eng. New Ideas Emerg. Results (ICSE-NIER)*, vol. 3, Gothenburg, Sweden, May/Jun. 2018, pp. 81–84.

[5] CoinMarketCap. *Total Market Cap of Bitcoin*. Accessed: Jul. 2019. [Online]. Available: https://coinmarketcap.com/currencies/bitcoin

[6] Consensys. *Mythril Classic: Ethereum Smart Contract Verification Tool*. Accessed: Jul. 2019. [Online]. Available: https://github.com/ConsenSys/mythril-classic

[7] *Echidna: A Fuzz Tester for Ethereum Smart Contracts*. Accessed: Jul. 2019. [Online]. Available: https://github.com/trailofbits/echidna

[8] Ethereum. *Ethereum Yellow Paper*. Accessed: Jul. 2019. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[9] Etherscan. *Ethereum (ETH) Blockchain Explorer*. Accessed: Jul. 2019. [Online]. Available: https://etherscan.io

[10] Etherscan. *Smart Contracts With Verified Source Code on Ethereum*. Accessed: Jul. 2019. [Online]. Available: https://etherscan.io/contractsVerified

[11] Etherscan. *Total Number of Transactions on the Ethereum Blockchain*. Accessed: Jul. 2019. [Online]. Available: https://etherscan.io/txs

[12] Y. Fu, M. Ren, F. Ma, H. Shi, X. Yang, Y. Jiang, H. Li, and X. Shi, "EVMFuzzer: Detect EVM vulnerabilities via fuzz testing," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. - ESEC/FSE*, Tallinn, Estonia, Aug. 2019, pp. 1110–1114.

[13] Google BigQuery. (2019). *Google Big Query Ethereum Dataset*. Accessed: Jul. 2019. [Online]. Available: https://bigquery.cloud.google.com/dataset/bigquery-public-data:ethereum_blockchain?pli=1

[14] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in Ethereum smart contracts," in *Proc. ACM Program. Lang. (OOPSLA)*, Boston, MA, USA, Nov. 2018, pp. 1–27.

[15] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Nov. 2019, pp. 531–548.

[16] B. Jiang, A. Wang, Z. Zheng, W. K. Chan, and N. Li, "Artemis: An improved smart contract verification tool for vulnerability detection," presented at the CCF China Blockchain Conf. (CCF CBCC), 2019, pp. 1–17, Paper 87.

[17] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, Montpeller, France, Sep. 2018, pp. 259–269.

[18] B. Jiang, Y. Liu, and W. K. Chan. (2018). *ContractFuzzer*. Accessed: Jul. 2019. [Online]. Available: https://github.com/gongbell/ContractFuzzer

[19] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, USA, Feb. 2018, pp. 1–12.

[20] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. 25th ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Toronto, ON, Canada, Oct. 2018, pp. 2123–2138.

[21] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. 23rd ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Vienna, Austria, Oct.2016, pp. 254–269.

[22] T. Nguyen, L. Pham, J. Sun, Y. Lin, and M. Tran, "sFuzz: An efficient adaptive Fuzzer for solidity smart contracts," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul, South Korea, Jul. 2020.

[23] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New York, NY, USA, Dec. 2018, pp. 653–663.

[24] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Chohen, and M. Vechev, "VerX: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SSP)*, San Jose, CA, USA, May 2020, pp. 18–20.

[25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, Gothenburg, Sweden, May 2018, pp. 9–16.

[26] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. 25th ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, Toronto, ON, Canada, Oct. 2018, pp. 67–82.

[27] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," 2019, *arXiv:1905.06944*. [Online]. Available: https://arxiv.org/abs/1905.06944

[28] M. Zalewski. *American Fuzzy Lop*. Accessed: Jul. 2019. [Online]. Available: http://lcamtuf.coredump.cx/afl

IMRAN ASHRAF received the B.S. degree in computer and information sciences from the Pakistan Institute of Engineering and Applied Sciences (PIEAS) in 2011. He is currently pursuing the Ph.D. degree in computer science with the City University of Hong Kong. His current research involves program analysis and security vulnerability detection in decentralized platforms, such as Ethereum blockchains. His primary interests in security vulnerability detection techniques are fuzz testing, static analysis, and symbolic execution. He received the Gold Medal Award for his B.S. degree.

XIAOXUE MA received the B.Eng. degree (Hons.) in telecommunication engineering from the College of Physical Science and Technology, Central China Normal University (CCNU), China, in 2017. She is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong. Her current research interest includes dynamic program analysis on concurrency bug detection in multithreaded programs.

BO JIANG received the Ph.D. degree in computer science from the University of Hong Kong, in 2011. He is currently an Associate Professor with the School of Computer Science and Engineering, Beihang University, Beijing, China. In recent years, he has published more than 30 academic articles in internationally renowned journals and conferences. His main research directions are software testing, mobile security, and blockchain security technology. He has won the best papers in international conferences three times. He also serves as a Program Committee Member for international conferences, such as COMPSAC'15, ICWS'15, APSEC'2012, QSIC'2013, and as JSS, IST, TSE, ASEJ, *Science China Information Sciences*, ASE, COMPSAC, QISC, and APSEC.

W. K. CHAN is currently an Associate Professor with the City University of Hong Kong. His current main research interest is software engineering in general, and program analysis and testing for concurrent software and systems, as well as software infrastructure for AI-based systems. He is also the Special Issues Editor of *Journal of Systems and Software*, an Associate Editor of *Software Testing, Verification, and Reliability*, a Review Editor of *Array*, a Guest Editor of the IEEE TRANSACTIONS ON RELIABILITY, the Program Chair of COMPSAC 2020 and QRS 2020, and a Program Committee Member of ECSE/FSE 2020, ASE 2020, and ICSE 2021. His research results have been reported in more than 100 articles with venues, including but not limited to TOSEM, TSE, TPDS, TSC, TRel, CACM, Computer, ICSE, FSE, ISSTA, ASE, WWW, ICWS, and ICDCS.