ContractGuard© White Paper
GuardStrike Inc.

Section 1. Introduction

Nowadays, smart contracts [9], [18] are implemented as Turing-complete programs that execute on the infrastructure of the blockchain [19]. It provides a framework which potentially allows any program (a.k.a. contract) to be executed in an autonomous, distributed, and trusted way. Smart contracts thus have the potential to revolutionize many industries. Popular applications of smart contracts include crowd fund raising, online gambling and so on. Ethereum [4], [5] is the first to introduce the functionality of smart contracts. Based on the Ethereum platform, Solidity is the most popular programming language for smart contracts [7].

Like traditional C/Java programs, smart contracts may contain vulnerabilities. Unlike traditional programs, smart contracts cannot be modified easily once they are deployed on the blockchain [15]. As a result, a vulnerability renders the smart contract forever vulnerable, which significantly magnifies the problem. In recent years, there have been an increasing number of news reports on attacks targeting smart contracts, which exploits security vulnerabilities in Ethereum smart contracts. One particularly noticeable example is the DAO attack [2], i.e., an attacker stole more than 3.5 million Ether (which is equivalent to about $45 million USD at the time) exploiting a vulnerability in the DAO contract. To fix the vulnerability, a hard fork is launched which is not only expensive but also caused much controversy [2].

It is thus desirable to develop tools for validating smart contracts to identify vulnerabilities, ideally before they are deployed. Among the range of complementary techniques for validating smart contracts, `ContractGuard` focuses on automatic testing of smart contracts as testing is often the least expensive. To automatically test smart contracts (or any program), we must solve the following three problems:

1. the test automation problem (i.e., how to automatically run the test cases),
2. the test generation problem (i.e., what test cases do we generate, among all the possible ones), and
3. the oracle problem (i.e., what are regarded as vulnerabilities).

To address the aforementioned problems GuardStrike has developed `ContractGuard` that mainly exploits symbolic execution and fuzzing techniques integrated with static and dynamic analysis that have been proven effective for software testing. With optimizations particularly designed for smart contacts, `ContractGuard` is highly efficient and accurate. It has been systematically applied to a set of more than 50 thousand smart contracts. `ContractGuard` is on average more than two orders of magnitudes faster than similar state-of-the-art tools. It also covers more branches and reveals many more vulnerabilities.

Section 2. Fuzzing

ContractGuard adopts an adaptive fuzzing strategy that is designed to maximize code coverage through multi-objective optimization. Given a smart contract, ContractGuard automatically configures a blockchain network, deploys the smart contract, and generates multiple transactions each of which calls a function in the contract. The transactions are then executed with an EVM enriched with a set of oracles for identifying vulnerabilities.

ContractGuard focuses on code coverage, in particular, branch coverage. A branch in *S* is covered by a test suite if and only if there is a test case *t* in the suite that visits the edge at least once. The branch coverage of a test suite is calculated as the percentage of the covered branches over the total number of branches. Note that identifying the total number of (feasible) branches statically in a smart contract is often infeasible for two reasons. First, some branches might be infeasible (i.e., there does not exist any test case that visits the branch) and knowing whether a branch is feasible or not is a hard problem. Second, EVM has a stack-based implementation which makes identifying all potentially feasible branches hard. ContractGuard thus aims to generate a test suite that maximizes the branch coverage.

ContractGuard monitors the execution of the transactions to collect certain feedbacks, e.g., whether a certain branch has been covered and how far the branch is far being covered. Whenever a vulnerability is revealed, the transactions and the network configuration (i.e., a test case) are saved for reporting to the user later on. Otherwise, some of test cases are selected as seeds based on feedback collected during the transaction execution according to certain seed selection criteria. Afterwards, the seeds are mutated to generate the next-generation of test cases. This process repeats until a time out occurs.

Section 2.1 Mathematical Models

A **smart contract** *S* typically has a number of instance variables, a constructor and multiple functions, some of which are public. It can be equivalently viewed in the form of a control flow graph (CFG) $S = (N, i, E)$, where N is a finite set of control locations in the program; $i \in N$ is the initial control location, i.e., the start of the contract; and $E \subseteq N \times C \times N$ is a set of labeled edges, each of which is of the form (n, c, n') where c is either a condition (for conditional branches like if-then-else or while-loops) or a command (i.e., an assignment). A node in the graph is branching if and only if it has multiple child nodes and its outgoing edges are labeled with conditions.

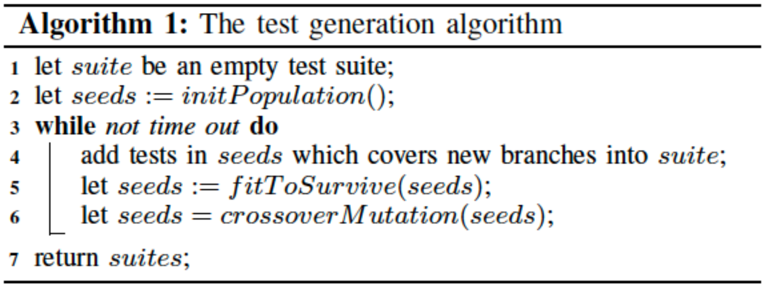
A **test case** for *S* is a pair $t=(\sigma_i, \Sigma)$ where σ is a configuration of the blockchain network and Σ is a sequence of transactions (i.e., function calls). The configuration σ_i contains all information on the setup of the network which is relevant to the execution of the smart contract. Formally, σ_i is a tuple (*b*, *ts*, *sa*, *sb*, *v*) where *b* is the current block number, *ts* is the current block timestamp, *sa* is the address of the smart contract, *sb* is its balance and *v* is the initial valuation of the persistent state. $\Sigma = \langle m_0(\overrightarrow{p_0}), m_1(\overrightarrow{p_1}), \cdots \rangle$ is a sequence of public function calls of the smart contract, each of which has an optional sequence of concrete input parameters $\overrightarrow{p_i}$. Note that m_0 must be a call of the constructor. The task of fuzzing a smart contract is thus to generate a set of test cases according to certain criteria. The execution of a test case *t* traverses through a path in the CFG *S*, which visits a set of nodes and edges. A trace generated by *t* is a sequence of pairs of the form $\langle (\sigma_0, n_0), (\sigma_1, n_1), \cdots \rangle$ where $(n_0, n_1, \cdots)$ are the sequence of nodes visited by *t* and σ_i is the configuration at the time of visiting node n_i for all *i*. A test suite is a set of test cases.

Section 2.2 Feedback-Guided Fuzzing

The general idea of feedback-guided fuzzing is to transformation the test generation problem into an optimization problem and use some form of ‘feedback’ as an ‘objective function’ in solving the optimization problem. At the top level, `ContractGuard` employs a genetic algorithm to evolve the test suite in order to iteratively improve its branch coverage. The overall algorithm is shown in Algorithm 1. Variable *suite* is the test suite to be generated when the algorithm terminates. It is initially empty. A generated test case is added into *suite* if it covers a new branch. Variable *seeds* is a set of seed test cases, based on which new test cases are generated. First, we generate an initial test suite using function *initPopulation*(). The loop from line 3 to 6 then iteratively evolves the test suite. In particular, `ContractGuard` adds those test cases in seeds which cover new branches (i.e., any branch that has not been covered by test cases in *suite*) into *suite* at line 4. At line 5, `ContractGuard` filters the test cases in *seeds* through function *fitToSurvive*() so as to focus on those seeds which are more likely to lead to test cases covering new branches later. At line 6, function *crossoverMuatation*() generates more test cases based on the test cases in seeds. The loop continues until a pre-set time out is triggered.

Function *initPopulation*() generates an initial population containing multiple test cases. As mentioned above, to generate a test case, `ContractGuard` needs to generate an initial configuration σ_i as well as a sequence of function calls with concrete parameters. The initial configuration by default is as follows: *b = 0, t = 0, sa = 0xf0, sb = 0xff0000000000000000000000* and *v* is set using the initial value for each variable representing the persistent state. Since the initial test suite may have an impact of the fuzzing process [12], `ContractGuard` allows a user to customize the initial configuration by providing an initial set of test cases.

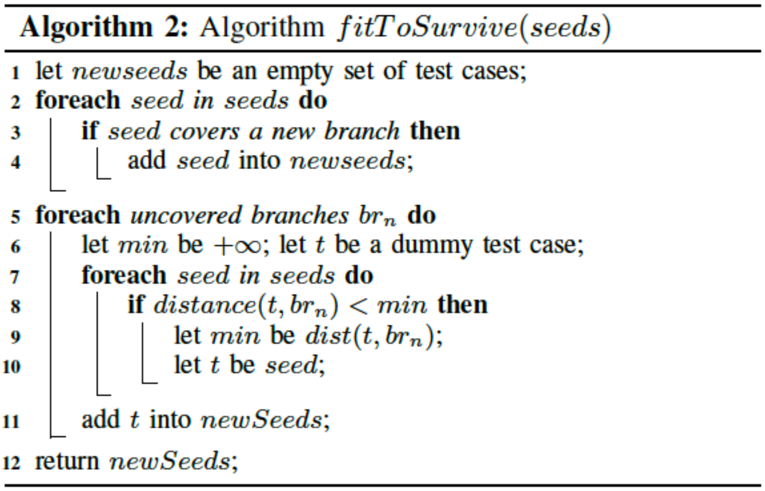
Next, `ContractGuard` generates multiple sequences of transactions, each of which is a function call with concrete parameters. For a contract with *n* functions, `ContractGuard` first generates *n* sequences. In each sequence, a different function is called once after the constructor is called. This makes sure that each function is tested at least once to achieve 100% function coverage. In addition, `ContractGuard` generates a random sequence of *r* function calls by randomly selecting one function from the set of public functions in the contract at a time until *r* function calls are generated. For each function call, `ContractGuard` generates a randomly value for each parameter based on its type. Each test case is encoded in form of a bit vector. The size of the bit vector equals to the number of bits for encoding the configuration plus the number of bits encoding the function calls. Note that to encode a function call, we first use $\lceil \log n \rceil$ bits, where *n* is the total number of public functions excluding the constructor, to encode the called function and then encode each parameter value. If the parameter value is of variable length, `ContractGuard` uses $\lceil \log bound \rceil$ (where *bound* is a bound on the maximum length with a default value of 256) to encode the length of the parameter value. Before executing the test case, the bit vector is decoded to a test case according to our internally defined protocol on how to encode/decode the bit vectors.



Algorithm 1: The test generation algorithm

```
1  let $suite$ be an empty test suite;
2  let $seeds := initPopulation()$;
3  while not time out do
4      add tests in $seeds$ which covers new branches into $suite$;
5      let $seeds := fitToSurvive(seeds)$;
6      let $seeds = crossoverMutation(seeds)$;
7  return $suites$;
```

After executing the seeds at line 4 in Algorithm 1, function *fitToSurvive*() is called to evaluate the fitness of the seeds according to certain fitness function. Note that the fitness function plays an extremely important role in genetic algorithms. `ContractGuard` combines two complementary strategies. Firstly, `ContractGuard` monitors the execution and records the branches that each seed covers. A test case is deemed 'fit to survive' if it covers a new branch in the contract. This strategy has been shown to be effectiveness in many settings [1]. However, such approach often makes very slow progress in covering the remaining branches after the initial branches are quickly covered. `ContractGuard` thus adopts a second adaptive strategy that selects seeds based on a quantitative measure on how far a test case is from satisfying a branching condition of an uncovered branch.



Algorithm 2: Algorithm $fitToSurvive(seeds)$

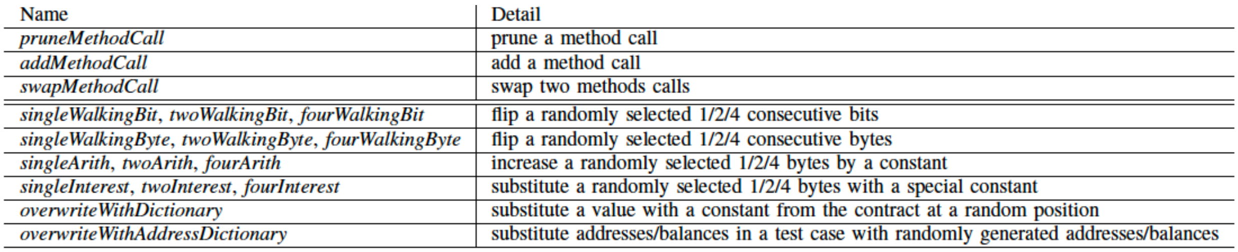
```
1  let $newseeds$ be an empty set of test cases;
2  foreach seed in seeds do
3      if seed covers a new branch then
4          add $seed$ into $newseeds$;
5  foreach uncovered branches $br_n$ do
6      let $min$ be $+\infty$; let $t$ be a dummy test case;
7      foreach seed in seeds do
8          if $distance(t, br_n) < min$ then
9              let $min$ be $dist(t, br_n)$;
10             let $t$ be $seed$;
11     add $t$ into $newSeeds$;
12 return $newSeeds$;
```

Function *fitToSurvive*(*seeds*) selects seeds as shown in Algorithm 2. The loop from line 2 to 4 goes through every test case to select those that cover a new branch. Afterwards, for each uncovered branch in the smart contract, the loop from line 5 to line 11 selects a test case from seeds which is the closest to cover the branch according to the distance between *t* and br_n. Note that one seed is selected for each uncovered branch, which makes this algorithm more like multi-objective optimization. All selected seeds are then used for crossover and mutation to generate more test cases in the next step.

Function *crossoverMutation*() is designed to generate new test cases based on those in seeds through crossover and mutation. `ContractGuard` adopts the crossover strategy similar to AFL. For instance, `ContractGuard` (1) randomly chooses two test cases from seeds; (2) breaks the two test cases into two pieces at a selected position; and (3) swaps the second pieces to form two new test cases. Note that due to correlations between the bits representing a test case, there is no guarantee that the resultant test cases are valid and thus `ContractGuard` always checks for validity and discard those invalid ones. Mutation is another way of generating new test cases based on the seeds. Given a test case encoded in the form of a bit vector, `ContractGuard` supports a rich set of mutation operators to generate new test cases. All mutation operators are shown in Table I. To ensure that the mutated test cases are valid, the mutations in different categories are implemented differently.

Recall that a test case is in the form of an initial configuration and a sequence of function calls with concrete parameters. The first three mutation operators aim to alter the sequence of function calls, by pruning a function call, adding a function call or swapping two function calls. For those values in a test case other than those representing the function names, `ContractGuard` categorizes them into two groups. The first group contains those values which have fixed-length. `ContractGuard` systematically applies the remaining mutation operators shown in Table I. The second group contains those values which have variable-length. Note that for such values, their lengths are encoded as part of the test case as well. For such cases, `ContractGuard` first mutates the value representing the length in such a way that the result is a random value between 0 and 255 where 255 is an upper bound. If the new length is less than the current one, the corresponding value is shortened accordingly by pruning the additional bits. If the length is more than the current one, random type-compatible values are padded accordingly.

Table 1: Mutations for fix-length values



Name	Detail
pruneMethodCall	prune a method call
addMethodCall	add a method call
swapMethodCall	swap two methods calls
singleWalkingBit, *twoWalkingBit*, *fourWalkingBit*	flip a randomly selected 1/2/4 consecutive bits
singleWalkingByte, *twoWalkingByte*, *fourWalkingByte*	flip a randomly selected 1/2/4 consecutive bytes
singleArith, *twoArith*, *fourArith*	increase a randomly selected 1/2/4 bytes by a constant
singleInterest, *twoInterest*, *fourInterest*	substitute a randomly selected 1/2/4 bytes with a special constant
overwriteWithDictionary	substitute a value with a constant from the contract at a random position
overwriteWithAddressDictionary	substitute addresses/balances in a test case with randomly generated addresses/balances

Section 2.3 Fuzzing Engine Components

The fuzzing engine It has 3 main components: **runner**, **libfuzzer** and **liboracles** .

Component **runner** solves the test automaton problem, i.e., how to run test cases. `ContractGuard` takes as input the bytecode of a smart contract along with either the application binary interface (ABI) or the source code of the contract. The runner then generates a bash script file which contains a list of commands to analyze the ABI or compile the source code, and set options for the other two components. The runner sets up a private test network based on which smart contracts are deployed and transactions are executed. To generate test cases for functions with address-type parameters, `ContractGuard` deploys a pool of externally owned accounts in the private test network with random balance for each account. The pool size is less than or equal to the number of addresstype parameters. The values for address-type parameters are then chosen randomly from this pool. In addition, `ContractGuard` deploys two special smart contracts as attackers, which are a *normal attacker* and a *reentrancy attacker*. Each attacker is set as the owner of contract under test in turn. The *normal attacker* throws an exception whenever other contracts call its payable fallback function. The *reentrancy attacker* calls back the function which makes a call to its payable fallback function. If the attacker fails to call back, it acts as a *normal attacker*. Note that the *reentrancy attacker* is only loaded to additionally detect *Reentrancy* vulnerability. Otherwise, the normal attacker is loaded to avoid call loops of *Reentrancy Attacker* which significantly reduces the speed of `ContractGuard`.

Component **libfuzzer** solves the test generation problem, i.e., how to selectly generate test cases, by implementing the fuzzing strategy presented in previous sections. It is responsible for multiple tasks. First, it constructs the CFG of the given smart contract on-the-fly. Second, component libfuzzer implements the fuzzing algorithm discussed earlier. One optimization is that we identify view functions (i.e., those which do not change any variables) and exclude them from test cases. The justification is that these view functions do not change the states and having them does not additional expose those vulnerabilities `ContractGuard` targets at. Since, view functions are marked by view, pure or constant keywords, `ContractGuard` reads ABI file to recognize them. Note that the architecture of libfuzzer makes it easy to integrate new mutation operators or optimization algorithms.

Component **liboracles** solves the oracle problem, i.e., it monitors the execution of a test case and checks whether there is a vulnerability according to an extensible library of oracles used in `ContractGuard`. `ContractGuard` monitors the execution of test cases through the hooker mechanism. Whenever EVM executes an opcode, it creates an event containing read-only execution information, such as the values of the stack, memory, program counter, and the current executed opcode. `ContractGuard` monitors these events for constructing the CFG and computing distance between a testcase and branches, as well as logs the events for vulnerability detection. To reduce the execution overhead, vulnerability detection is conducted offline in batches. This design allows `ContractGuard` to easily support different versions of Solidity. `ContractGuard` has an extensible architecture that allows it to easily support different oracles as well. These oracles are checked one by one based on the log of a test case. The test cases that expose vulnerabilities in the contract are kept in a separate test suite and reported to the user together with the vulnerabilities that they expose. Note that by design, the fuzzing engine always reports true positives according to our definition of vulnerability except in the case of Freezing Ether. However, in practice, a reported vulnerability might be a false positive still as it may be what the user intended (i.e., our definition of vulnerability is too strict). In the case of Freezing Ether, the identified ‘warning’ might be a false positive if there exists some test case which calls send() or transfer() but such test cases are never generated. Technically, the problem of checking whether there is Freezing Ether vulnerability can only be solved if we cover all feasible opcode (which is often infeasible).

Section 3. Symbolic Execution

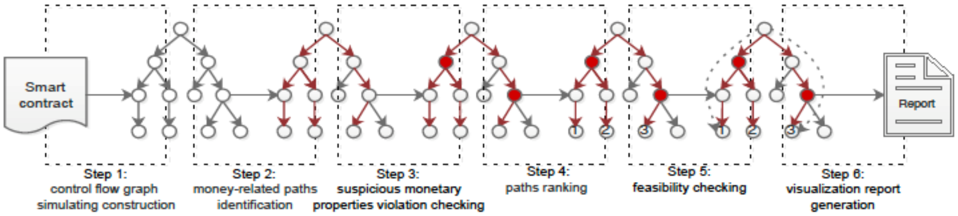

Smart contract
Step 1: control flow graph simulating construction
Step 2: money-related paths identification
Step 3: suspicious monetary properties violation checking
Step 4: paths ranking
1
2
Step 5: feasibility checking
3
1
2
3
Step 6: visualization report generation
Report

Figure 1: overall workflow

Overall, symbolic execution engine works as follows, as illustrated in Figure 1.

1. `ContractGuard` firstly constructs a control flow graph (CFG) which captures all possible control flow including those due to the inter-contract function calls. It then systematically generates paths (with a bounded sequence of function calls).
2. To address path explosion, `ContractGuard` then statically identifies paths which are ‘critical’. `ContractGuard` considers paths involving monetary transactions as critical paths, which is often sufficient in capturing vulnerabilities in smart contracts.
3. `ContractGuard` define a set of (configurable) money-related properties based on existing vulnerabilities and identify all paths that potentially violate our properties.
4. Considering that different properties have different criticalness and a long path may be unlikely feasible than a short one, `ContractGuard` ranks all paths by computing a criticalness score for each path based on the two factors.
5. So far the paths that exhibit vulnerable behavior may not be feasible. Reporting these paths to users lead to substantial false positive. `ContractGuard` exploits symbolic execution to remove infeasible paths.
6. Finally, for top ranked paths, `ContractGuard` automatically checks whether it is feasible using symbolic execution techniques. And, the feasible paths are presented to the user for inspection.

Section 3.1 CFG Analysis

`ContractGuard` constructs a CFG for a smart contract (the compiled EVM opcode with a single entrance for whole and for each function) to capture all possible paths. Formally, a CFG in the symbolic execution engine is a tuple (N, $root$, E) such that

- N is a set of nodes, where each node is a basic block of opcodes.
- $root \in N$ is the first basic block of opcodes.
- $E \subseteq N \times N$ is a set of edges, where each edge (n,n') corresponds to exactly a control directly from flow n to n'.

`ContractGuard` also considers inter-contract functions calls, where there is a *CALL* to a foreign function that is assumed to call the current function including third-part contract. Given a bound b on the number of function calls, we can systematically unfold the CFG so as to obtain all paths during which only b or fewer functions are called.

Given a bound b on the number of call depth and a bound on the loop iterations, there are still many paths in the CFG to be analyzed. `ContractGuard` focuses on money-related paths to avoid path explosion as almost all vulnerabilities [8] are ‘money’-related. A node is money-related if and only if its BBL contains any of following opcode instructions: *CALL*, *CREATE*, *DELEGATECALL* or *SELFDESTRUCT* . In general, one of these instructions must be used when Ether is transferred from one account to another. A path which traverses through a money-related node is considered money-related.

Next, `ContractGuard` prioritizes paths that violate critical properties. The objective is to prioritize those paths which may trigger violation of critical properties for user inspection. The properties, several of which are listed below, are designed based on previously known vulnerabilities and they can be configured and extended.

- Respect the Limit. `ContractGuard` allows users to set a limit on the amount of Ether transferred out of the contract's wallet. For each path, `ContractGuard` statically checks whether Ether is transferred out of the wallet and whether the transferred amount is potentially beyond the limit. To do so, for each path, `ContractGuard` uses a symbolic variable to simulate the remaining limit. Each time an amount is transferred out, `ContractGuard` decreases the variable accordingly and check whether the remaining limit is less than zero. If so, the path potentially violates the property. Note that if it is impossible to determine the exact amount to be transferred, `ContractGuard` conservatively assumes the limit may be broken.
- Avoid Non-Existing Addresses. Any hexadecimal string of length no greater than 40 is considered a valid (well-formed) address in Ethereum. If a non-existing address is used as the receiver of a transfer, the Solidity compiler does not generate any warning and the contract can be deployed on Ethereum successfully. If a transfer to a non-existing address is executed, Ethereum automatically registers a new address (after padding 0s in front of the address so that its length becomes 160bits). Because this address is owned by nobody, no one can withdraw the Ether in it since no one has the private key. For every path which contains instruction *CALL* or *SELFDESTRUCT* , `ContractGuard` checks whether the address in the instruction exists or not. This is done with the help of EtherScan. A path which sends Ether to a non-existing address is considered to be violating the property. Currently, to minimize the number of requests to EtherScan, `ContractGuard` only queries external transactions, thus may lead to false positives when the address has only internal transactions. Of course, users can configure `ContractGuard` to also check internal transactions.
- Guard Suicide. `ContractGuard` checks whether a path would result in destructing the contract without constraints on the date or block number, or the contract ownership. A contract may be designed to "suicide" (with the opcode *SELFDESTRUCT*) after certain date or reaching certain number of blocks, and often by transferring the Ether in the contract wallet to the owner. A famous example is Parity Wallet which resulted in an estimated loss of tokens worthy of $155 million [3]. `ContractGuard` thus checks whether there exists a path which executes *SELFDESTRUCT* and whether its path condition is constituted with constraints on date or block number and contract owner address. While checking the former is straightforward, checking the latter is achieved by checking whether the path contains constraints on instruction *TIMESTAMP* or *BLOCK*, and checking whether the path condition compares the variables representing the contract owner address with other addresses. A path which calls *SELFDESTRUCT* without such constraints is considered a violation of the property.

- Be No Black Hole In a few cases, `ContractGuard` analyzes paths which do not contain *CALL*, *CREATE*, *DELEGATECALL* or *SELFDESTRUCT*. For instance, if a contract has no money-related paths (i.e., never sends any Ether out), `ContractGuard` then checks whether there exists a path which allows the contract to receive Ether. The idea is to check whether the contract acts like a black hole for Ether. If it does, it is considered a vulnerability.

Section 3.2 Ranking Program Paths

To allow user to focus on the most critical paths as well as to save analyses efforts, `ContractGuard` prioritizes the paths according to the likelihood they reveal critical vulnerability. For each path, we calculate a criticalness score and rank the paths according to the scores. The criticalness score is calculated as follows: let pa be a path and V be the set of properties which pa violates.

$$criticalness(pa) = \Sigma_{pr \in V} \alpha_{pr} / \varepsilon \times bound(pa)$$

where α_{pr} is a constant that denotes the criticalness of violating property pr, $bound(pa)$ is the depth bound of path pa and ε is a positive constant. Intuitively, the criticalness is designed such that the more critical a property the path violates, the larger the score is; and the more properties it violates, the larger the score is. Furthermore, it penalizes long paths so that short paths are presented first for user inspection.

To assess the criticalness of each property, we use the technique called failure mode and effects analysis (FMEA [17]) which is a risk management tool widely used in a variety of industries. FMEA evaluates each property with 3 factors, i.e., Likelihood, Severity and Difficulty. The criticalness α_{pr} is then set as the product of the three factors. After ranking, only paths that have a criticalness score larger than certain threshold are subject to further analysis, reducing the number of paths significantly. In order to identify the threshold for criticalness, `ContractGuard` adapts the *k*-fold cross-validation[20,13] idea in statistical area.

Section 3.3 Feasibility Checking

Not all the paths are feasible. To avoid such false alarms, `ContractGuard` filters infeasible paths through symbolic execution [11, 14]. The basic idea is to symbolically execute a given program to reduce the paths which are to be presented for users' inspection. Only if a path is found to be infeasible by symbolic execution, `ContractGuard` removes it. The symbolic execution engine in sCompile is built based on the Z3 SMT solver [16].

Section 4. Evaluation

Our test subjects include 50199 smart contracts which we collect from EtherScan [6]. Note that these include all the smart contracts available on EtherScan as of Jan, 2019. All experiment results reported below are obtained on an Ubuntu 18.04.1 LTS machine with Intel Core i7 and 16GB of memory.

Section 4.1 Efficiency

We measure the efficiency of `ContractGuard` by counting how many test cases are generated and executed per second. Naturally a test case for a more complicated contract (e.g., with many loop iterations) takes more time to execute. Thus, we show how the efficiency varies for different contracts. Figure 2 summarizes the result, where each bar represents 10% (about 3400) of the contracts and the y-axis shows the number of test cases generated and executed per second. The contracts are sorted according to how efficiency it can be tested. From the figure, we observe that the efficiency varies significantly over different contracts, i.e., `ContractGuard` generates and executes more than 5000 test cases per second on average for the top 10% of the contracts, and less than 100 test cases for the bottom 20%. On average, `ContractGuard` generates and executes more than 1000 test cases per second.

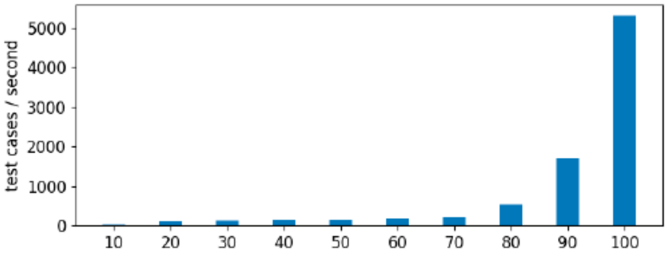
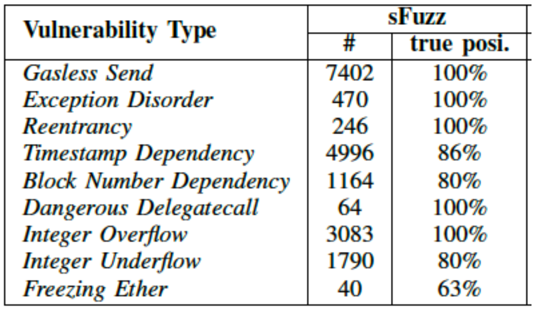


Figure 2: Efficiency Evaluation

We further conduct an experiment to measure of overhead of monitoring the execution of a test case (using the hooker mechanism) and the overall overhead of the testing process. We apply `ContractGuard` to a randomly select a set of 60 contracts and measure the time spent on executing the test cases. The results show that on average the monitoring consumes about 10% of the total execution time and the overall overhead of the testing process (including monitoring) is about 14%, which is reasonably efficient.

Section 4.1 Effectiveness

Table 2: Vulneraiblities



Vulnerability Type	sFuzz	
	#	true posi.
Gasless Send	7402	100%
Exception Disorder	470	100%
Reentrancy	246	100%
Timestamp Dependency	4996	86%
Block Number Dependency	1164	80%
Dangerous Delegatecall	64	100%
Integer Overflow	3083	100%
Integer Underflow	1790	80%
Freezing Ether	40	63%

Table 2 summarizes the number of vulnerable contracts discovered by `ContractGuard` in each category. The first column shows the type of vulnerability. The next column shows the number of vulnerable contracts found by `ContractGuard`. The sub-column # shows the number of contracts that have the vulnerability according to each vulnerability type and the second subcolumn is the percentage of true positives of the identified vulnerabilities. To evaluate the soundness of ContractGuard, we manually examine the identified vulnerable contracts to check they are true positives or not. However, we are unable to manually check all the identified vulnerability for two reasons. First, there are an overwhelming number of vulnerabilities. Second, we do not have the source code of many smart contracts and thus cannot check those contracts. Instead, we randomly sample 50 vulnerable contracts with source code in each category and manually check whether the identified vulnerability is a true positive or not. If there are fewer than 50 vulnerable contracts with source code in the category, we check all of them. For Gasless Send, Exception Disorder and Reentrancy vulnerability, all 50 sampled vulnerable contracts are true positives. For Timestamp Dependency, out of the 50 sampled vulnerable contracts, 43 of them are true positives. In the remaining 7 contracts, although block.timestamp and/or now is used in a condition, they are irrelevant to the Ether sending part (i.e., no control/data dependency). Rather their values are saved in global variables to record the creation time of specific events. ContractGuard mistakenly claims that such cases are vulnerable. For Block Number Dependency, 40 out of the 50 sampled vulnerable contracts are true positives. Similarly, the reason for the 10 false positives is the value of block.number is assigned to global variables but they are irrelevant to Ether sending process. For Dangerous Delegatecall, all 50 vulnerable contracts are indeed vulnerable. Similarly so for Integer Overflow. For Integer Underflow, 40 of the 50 identified contracts are indeed vulnerable. The reason for the 10 false positives is that because it is non-trivial to identify the correct type of a variable based on bytecode only (e.g., whether it is uint256 or uint128), ContractGuard conservatively assumes that all arithmetic operations returning a negative value may be vulnerable. Lastly, for Freezing Ether, 25 of the 40 identified contracts are true positives. The reason for the 15 false positives is that although there is a program path which allows the contract to send Ether, the program path is not covered and ContractGuard falsely assumes that there is no such program path. This percentage of such false positives is expected to be reduced if ContractGuard is applied for a longer period of time.

Section 4. Conclusion

ContractGuard exploits a symbolic execution engine and an adaptive fuzzing engine for EVM smart contracts. Experiment results show that it is significantly more reliable, faster, and more effective than existing tools that verify or test smart contracts.

REFERENCES

[1] AFL. http://lcamtuf.coredump.cx/afl/technical details.txt.

[2] Analysis of the DAO exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[3] Another parity wallet hack explained. https://medium.com/@Pr0Ger/another-parity-wallethack-explained-847ca46a2e1c. (Accessed on 06/06/2018).

[4] Ethereum white paper. https://github.com/ethereum/wiki/wiki/White-Paper.

[5] Ethereum yellow paper. https://ethereum.github.io/yellowpaper/paper.pdf.

[6] Etherscan. https://etherscan.io/.

[7] Solidity. https://solidity.readthedocs.io/.

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (sok). In International Conference on Principles of Security and Trust, pages 164–186. Springer, 2017.
[9] Christopher D Clack, Vikram A Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. arXiv preprint arXiv:1608.00771, 2016.
20. Pierre A Devijver and Josef Kittler. Pattern recognition: A statistical approach. Prentice hall, 1982.
[10] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005, pages 213–113, 2005.
[11] James C King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.
[12] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 2123–2138. ACM, 2018.
[13] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In Ijcai, volume 14, pages 1137–1145. Montreal, Canada, 1995.
[14] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. IEEE Transactions on Software Engineering, (4):266–278, 1977.
[15] Bill Marino and Ari Juels. Setting standards for altering and undoing smart contracts. In International Symposium on Rules and Rule Markup Languages for the Semantic Web, pages 151-166. Springer, 2016.
[16] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340. Springer, 2008.
[17] D. H. Stamatis. Failure Mode and Effect Analysis: FMEA from Theory to Execution. ASQ Quality Press, 2003.
[18] Nick Szabo. Formalizing and securing relationships on public networks. First Monday, 2(9), 1997.
[19] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, and Huaimin Wang. Blockchain challenges and opportunities: A survey. Work Pap.–2016,2016.