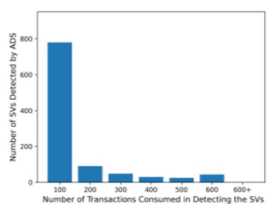
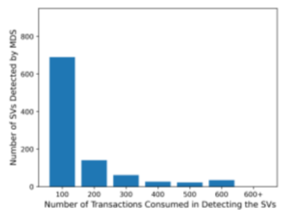
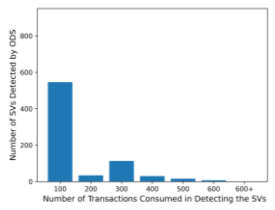
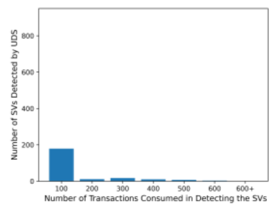
Supplementary Document of

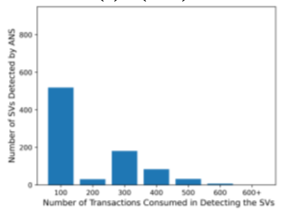
The Effects of Blockchain Environment on Security Vulnerability Detection in Fuzzing Ethereum Smart Contracts: An Empirical Study

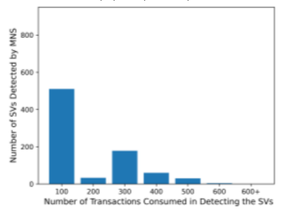

(a) ℂ(ADS)

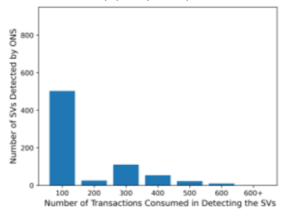

(b) ℂ(MDS)

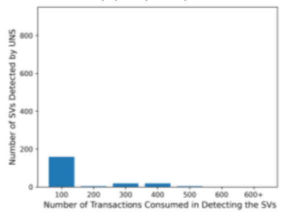

(c) ℂ(ODS)

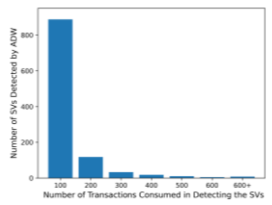

(d) ℂ(UDS)

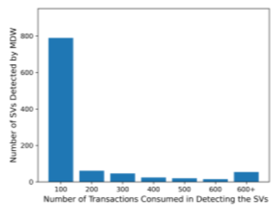

(e) ℂ(ANS)

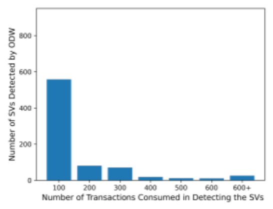

(f) ℂ(MNS)

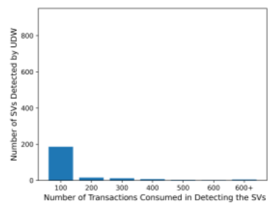

(g) ℂ(ONS)

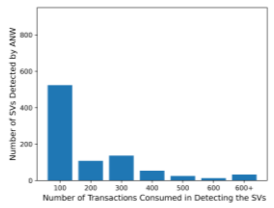

(h) ℂ(UNS)

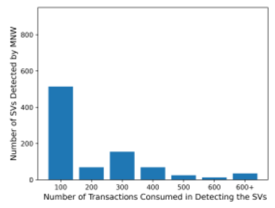

(i) ℂ(ADW)

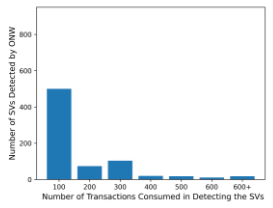

(j) ℂ(MDW)

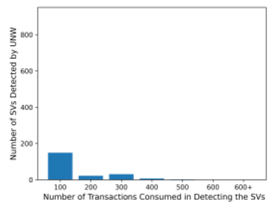

(k) ℂ(ODW)

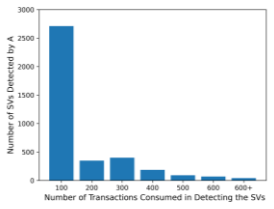

(l) ℂ(UDW)

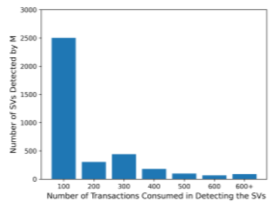

(m) ℂ(ANW)

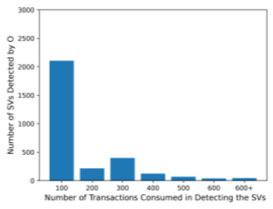

(n) ℂ(MNW)

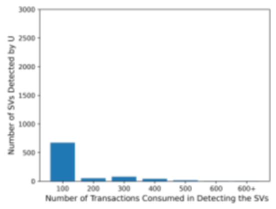

(o) ℂ(ONW)

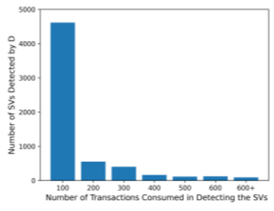

(p) ℂ(UNW)

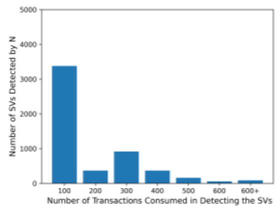

(q) F1: A

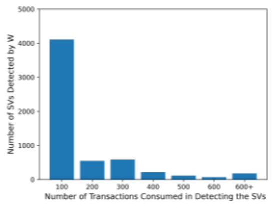

(r) F1: M


(s) F1: O


(t) F1: U


(u) F2: D


(v) F2: N


(w) F3: W

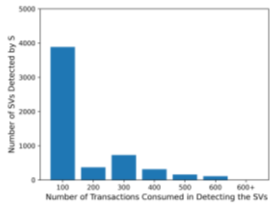

(x) F3: S

Figure 7: Number of SVs detected by Each Configuration or Level within Different Buckets of Test Budget Applied

Appendix I

Figure 7 presents the number of SVs detected by each configuration (in subfigures (a) to (p)) and each level of each design factor (in subfigures (q) to (x)). For ease of presentation, the detected SVs are placed into incremental buckets of 100 transactions, representing the number of transactions needed to detect these SVs. In the figure, higher bars for a configuration or level represent higher effectiveness in terms of the number of SVs detected, while higher bars in the initial buckets represent higher cost-effectiveness in terms of the number of transactions consumed to detect the SVs.

Appendix II - Applicability to Greybox Fuzzing

In this section, we compare the performance of a well-known coverage-guided greybox fuzzer AFL [44] adopted for smart contracts, referred to as AFL$_{\text{Blockchain}}$. We first configure AFL$_{\text{Blockchain}}$ with $\mathbb{C}$(ANS) as we infer it to be its default configuration used in AFL [44]. We infer Level A for F1 because AFL randomly mutates any part of the seed input (including the sender's address). Level N is inferred for F2 because AFL does not control any execution environment, and Level S is inferred for F3 because AFL allows an equal distribution of test resources among the test subjects. We refer to this configuration of AFL as AFL$_{\text{ANS}}$. For comparison, we configure AFL$_{\text{Blockchain}}$ with $\mathbb{C}$(ADW), the most cost-effective configuration obtained from answering RQ4. We refer to it as AFL$_{\text{ADW}}$.

In this case study, our main goal is to show the applicability of the most cost-effective configuration to greybox fuzzing. We believe, based on our understanding of the current test framework (adopted from ContractFuzzer [26]), the three factors are well-suited for the application of AFL (a vanilla greybox fuzzer that uses code coverage to guide the testing process).

Other smart contract fuzzers use different techniques and heuristics to guide the fuzzing process, and understanding their implementations would be error-prone and challenging to us, raising a threat of committing implementation mistakes. In existing works [23][26][32][43], their experiments used third-party tools without putting them into the same test framework to facilitate stricter comparisons. Since we aim at comparisons on the same test framework, therefore, we choose not to implement other fuzzers in the current case study.

A. *AFL for Ethereum Smart Contracts*

To show the applicability of the studied design factors to greybox fuzzing, Algorithm 2 presents AFL$_{\text{Blockchain}}$ for greybox fuzzing of Ethereum smart contracts. The algorithm generates and executes the seed transactions in lines 6–18, while the main fuzzing process is performed in lines 19–38.

Like Algorithm 1, this algorithm accepts a set of smart contracts $\mathbb{II}$ along with two accounts, a_1 (*owner*) and a_2 (*unknown*), and a test budget B.

Algorithm 1: AFL$_{\text{Blockchain}}$ Algorithm

```
Input:   𝕀𝕀 : a set of smart contracts to test
         a_1 : owner account
         a_2 : unknown account
         B : test budget
Output:  V : a list of triples
1   V = ⟨⟩
2   foreach sc_i in 𝕀𝕀
3       Deploy(sc_i) on the test framework with a_1
4       budget = ResourceAllocationStrategy (sc_i, 𝕀𝕀, B)            //Factor F3
5       count = 0                                                  // Transaction counter
6       Q = ⟨⟩                                                     // Seed Queue
7       foreach fn in sc_i.functions:                              // Seed generation and execution
8           t_s = generate a random seed test case for fn
9           SA = AddressSelectionStrategy(sc_i, a_1, a_2)          //Factor F1
10          foreach account a in SA
11              Q.append (t_s.senderAddress = a)
12      end for
13      cov(Q) = ExecutionStrategy(sc_i, Q)                        //Factor F2
14      foreach transaction tx in Q:
15          foreach security vulnerability v_j found in tx
16              V = V ∪ {⟨sc_i, v_j, tx⟩}
17      end for
18      count += length(Q)
19      do
20          L = ⟨⟩
21          foreach transaction t in Q
22              t_mut = t.mutate(t.input())
23              SA = AddressSelectionStrategy(sc_i, a_1, a_2)      //Factor F1
24              foreach account a in SA
25                  L.append (t_mut.senderAddress = a)
26          end for
27          if (count + length(L)) > budget
28              L = L[0 : (budget–(count+length(L)))]
29          cov(L) = ExecutionStrategy(sc_i, L)                    //Factor F2
30          foreach transaction tx in L:
31              if cov(tx) contains an element not in cov(∀tx_i ∈ Q)
32                  Q = ⟨tx,Q⟩
33              foreach security vulnerability v_j found in tx
34                  V = V ∪ {⟨sc_i, v_j, tx⟩}
35          end for
36          count += length(L)
37      while count < budget
38  end for
```

An empty list V is initialized at line 1. The algorithm then iterates over all the smart contracts in $\mathbb{II}$.

In the iteration for a smart contract sc$_i$, the smart contract is first deployed on the test framework using a_1 as the owner account at line 3. The algorithm then determines a *budget*, which is the number of transactions for fuzzing sc$_i$ (line 4) via *ResourceAllocationStrategy*(). Then, a transaction counter, *count*, is initialized at line 5, and an empty list of seed transactions Q is initialized at line 6. From lines 7–12, the algorithm iteratively generates seed transactions and appends them to Q. Each function *fn* of sc$_i$ is iteratively selected (line 7), and a seed test case t_s is randomly generated for *fn* (line 8). At line 9, the algorithm gets a list of sender accounts *SA* via *AddressSelectionStrategy*(). At line 10–11, for each account *a* in *SA*, the test case t_s is converted into a transaction with its sender address set to *a* and appended to Q. The transactions in Q are executed via the *ExectuionStrategy*() at line 13, and the code coverage (*cov*) is recorded. Each security vulnerability v_j of sc_i detected by transaction *tx* in Q is added to V as a triple $\langle \mathsf{sc}_i, v_j, t_k \rangle$ (lines 14–17). The algorithm increases *count* by the length of Q at line 18.

Inside the loop at lines 19–38, an empty list of transactions L is initialized at line 20. The algorithm initiates a loop for the total number of transactions in Q at lines 21–26. It then iteratively generates a mutant test case t_{mut} at line 22 for each transaction in the Q. At line 23, the algorithm gets a list of sender accounts *SA* via *AddressSelectionStrategy*(). At line 24–25, for each account a in *SA*, the test case t_{mut} is converted into a transaction with its sender address set to a and appended to L. The algorithm at lines 27–28 checks whether executing the full L exceeds the *budget* and truncates L accordingly if this is the case. The transactions in L are executed via the *ExectuionStrategy*() at line 29, and the code coverage (*cov*) is recorded. The algorithm at lines 30–35 iterates over each transaction *tx* in L. *tx* is added to the head position of the Q at lines 31–32 if a previously uncovered coverage item is discovered. Each security vulnerability v_j of sc_i detected by transaction *tx* is added to V as a triple $\langle \mathsf{sc}_i, v_j, t_k \rangle$ (lines 33–34). The algorithm increases *count* by the length of L at line 36. The algorithm checks at line 37 whether the *budget* has been exhausted.

The algorithm terminates at line 38.

B. Experimental Procedure

The experiment to compare the cost-effectiveness of AFL$_{\mathsf{ANS}}$ and AFL$_{\mathsf{ADW}}$ is conducted on the same experimental setup presented in Section V on dataset Ω. The procedure of the two experiments (referred to as E17 and E18) are summarized as follows.

To support greybox fuzzing, the test framework used in the above controlled experiment is extended with mutation operators. Algorithm 2 is also implemented. Two types of mutation operators are implemented for E17 and E18 below. The first mutation operator is to randomly flip the bits of an input parameter and the second one is an addition of random bits to the input parameter. For an address type input parameter, no mutations are applied. For fixed-length input parameters like *bools*, *uints*, and *ints*, only the first mutation operator is applied, and for other input parameters such as bytes and strings, any one or both mutation operators are applied. The *value* (amount of ether) sent with the test case is chosen between zero and non-zero values for test cases that call a payable function externally. For non-payable functions, the value parameter is always kept at zero, as providing a non-zero value to a non-payable function could revert the transaction.

For code coverage profiling, the test framework implements the basic block coverage.

The test framework generates one random test case for each SAF of each smart contract in Ω via line 8 in Algorithm 2. It is also extended so that in each repeated trial, it executes E17 and E18.

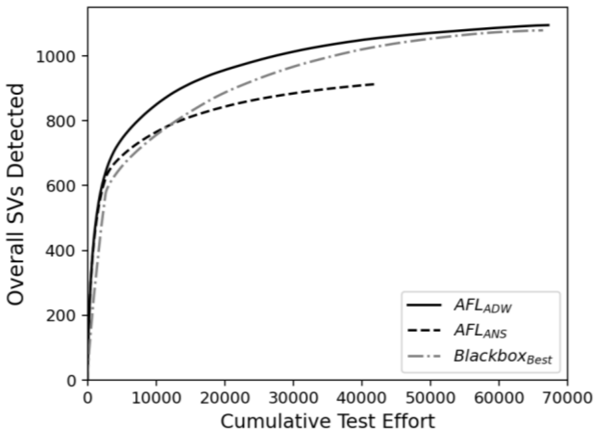


Figure 8: Comparison of AFL, AFL$_{\text{Blockchain}}$, and Blackbox$_{\text{Best}}$ on Cumulative Test Effort in SecurityVulnerability Detection

E17: We conduct this experiment by initializing the test framework with an empty testnet. It then applies AFL$_{\mathsf{ANS}}$ with $\langle \Omega, a_1, a_2, B \rangle$ as the input. All other settings are the same as E2.

E18: We conduct this experiment by repeating E17 with AFL$_{\mathsf{ADW}}$ instead of AFL$_{\mathsf{ANS}}$.

C. Answering RQ5

We denote the technique in the most cost-effective configuration for blackbox fuzzing, which is $\mathbb{C}(\mathsf{ADW})$ in E14, by Blackbox$_{\text{Best}}$.

The results for E17 and E18 are presented in Figure 8, which can be interpreted similarly to Figure 5. The figure presents the cumulative test effort to detect SVs by AFL$_{\mathsf{ANS}}$, AFL$_{\mathsf{ADW}}$, and Blackbox$_{\text{Best}}$. The total numbers of SVs detected by AFL$_{\mathsf{ANS}}$ and AFL$_{\mathsf{ADW}}$ are 913 and 1096, respectively, in five repeated trials, resulting in a 19.7% improvement.

The solid black line representing AFL$_{\mathsf{ADW}}$ is observed to be located consistently higher than both the dot-dashed-grey line representing Blackbox$_{\text{Best}}$ and the dashed-black line representing AFL$_{\mathsf{ANS}}$. The graph shows that compared to AFL$_{\mathsf{ANS}}$, AFL$_{\mathsf{ADW}}$ can consistently detect more SVs for the same amount of total test budget. It can further be observed from the figure that AFL$_{\mathsf{ADW}}$ detected 1050 SVs with the same cumulative test effort that was consumed by AFL$_{\mathsf{ANS}}$ in detecting the 913 SVs it detected, which is 14.9% more.

We also conduct *Tukey's HSD* test for the pair (AFL$_{\mathsf{ADW}}$, AFL$_{\mathsf{ANS}}$) to obtain a p-value of less than 0.001. Furthermore, *Pearson's r* value for the pair is larger than 0.50, showing a *large* effect size.

These results further strengthen the results obtained in answering RQ1–RQ4 that the three design factors identified in this work play a pivotal role in fuzz testing of smart contracts. The results also demonstrate that AFL_{ADW} can significantly improve the performance of a smart contract fuzzer (represented by AFL) to detect security vulnerabilities.

Summary of Findings in the Case Study: In comparison to AFL_{ANS}, an improvement of 19.7% was shown by AFL_{ADW} in the detection of SVs, and AFL_{ADW} can spend fewer transactions to detect the same number of SVs detected by AFL_{ANS} in the case study.