Supplementary Document for STAGFUZZER: Pattern-Aware Phase Alternation for Fuzzing Smart Contracts

1 Motivating Example

Figure 1 shows a contract `QuizContract` with 3 functions (`set`, `play`, and `withdraw`, denoted by α, β, and γ, respectively) and 6 state variables (`owner`, `solved`, `quiz`, `answer`, `ownerBalance`, and `userBalance`, denoted by `o`, `s`, `qu`, `a`, `oB`, and `uB`, respectively). Suppose that an account a_1 creates `QuizContract` via `constructor(_q = "", _a = "")` with an Ether `value` of 500, which sets the state of this contract to $\{$`balance` = 500, `o` = a_1, `qu` = `_q`, `a` = `keccak256(_a)`, `s` = `false`, `oB` = 500$\}$ via lines 11–15. If another account a_2 ($\neq a_1$) invokes β (in a test case t_u) with `_a=""` and an Ether `value` of 100, the conditions in lines 28–31 are met. Function β thus stores `uB`$[a_2]$ = 200 and `s` = `true` (lines 32–33) and emits an *event* `PlayedAt` in line 35 with a_2, the `Timestamp` (through the keyword `now`) and `BlockNumber` of the *block* in which t_u is mined. If a_2 then invokes γ (in a test case t_v), the condition in line 38 is met. Function γ thus sets `amount` = 200, `uB`$[a_2]$ = 0 and `oB` = 300 via lines 39–41, and then "sends" an amount of 200 to a_2 in line 43, since the conditions in line 42 is also met. Suppose this `send` call invokes a_2's fallback function, but the fallback function fails to run to completion. Since line 43 does not check for any raised exception during the execution of the `send` call, a *Gasless Send* security vulnerability [1] will be triggered, denoted by ♠, where no funds are transferred out and no exception is thrown.

A fuzzer should exhibit certain desirable behaviors to trigger ♠: (*i*) discover a new code region, (*ii*) identify a test case t_b connecting to a preceding test case t_a, (*iii*) execute t_b "soon after" executing t_a, and (*iv*) minimize wastages on test cases irrelevant to the connection.

Fuzzing starts from seed test cases. We suppose S=$\{t_\alpha$, t_β, $t_\gamma\}$ is a given set of seed test cases to execute lines 19–20, 28–31 and 34–35, and 38–42, in α, β, and γ, respectively. We further suppose that all the test cases *always* satisfy the `require` conditions in lines 19, 28, and 38. Moreover, we denote **lines 32–33** by ♣.

Figure 2 depicts the overall workflows of the following four fuzzers on how they test `QuizContract`.

AFL [5] mutates individual seed test cases. It starts from constructing a seed queue, e.g., $S' = \langle t_\beta, t_\gamma, t_\alpha\rangle$ from S. It mutates t_β and t_β's descendants to create several mutants. If such a mutant reaches ♣, the contract state is updated to `s` = *true* and `uB`$[a_2] > 0$ for an account a_2 specified in that mutant. Next, if a mutant originating from t_γ reaches line 43, it triggers ♠. But, if no t_β's or its descendant's mutant reaches ♣ yet, all such mutants are wasted. In general, AFL picks a test case, say t'_γ, for test case generation because of its coverage history. But, the picking is blind to how t'_γ relates to a specific historic test case (a mutant of t_β that reaches ♣) and unaware of the moment of historic test case execution.

The next two fuzzers mutate both seeds and test suites.

CONTRAMASTER [4] shuffles S to create several test

```
1 contract QuizContract{
2  address public owner;                    //denoted: o
3  bool public solved;                      //denoted: s
4  string public quiz;                      //denoted: qu
5  bytes32 private answer;                  //denoted: a
6  uint256 private ownerBalance;            //denoted: oB
7  mapping(address=>uint256) userBalance;   //denoted: uB
8  event PlayedAt(address player, uint _t, uint _b)
9
10 constructor(string _q,string _a) public payable{
11   owner = msg.sender;
12   quiz = _q;
13   answer = keccak256(_a);
14   solved = false;
15   ownerBalance = msg.value;
16 }
17 function set(string _q,string _a) public payable{
18   // denoted as α
19   require(msg.sender==owner);
20   require(solved==true);
21   quiz = _q;
22   answer = keccak256(_a);
23   solved = false;
24   ownerBalance += msg.value;
25 }
26 function play(string _r) external payable {
27   // denoted as β
28   require(msg.sender!=owner);
29   require(solved==false);
30   require(msg.value > 0 && msg.value <= 100);
31   if(answer==keccak256(_r)) {
32     ♣ userBalance[msg.sender] += 2*msg.value;
33     ♣ solved = true;
34   } else { ownerBalance += msg.value; }
35   emit PlayedAt(msg.sender, now, block.number)
36 }
37 function withdraw() external { // denoted as γ
38   require(msg.sender!=owner);
39   uint256 amount = userBalance[msg.sender];
40   userBalance[msg.sender] = 0;
41   ownerBalance = ownerBalance - amount;
42   if(amount>0) {
43     ♠ msg.sender.send(amount); } // Gasless Send
44 }
45 }
```

Fig. 1. A simplified Solidity smart contract.

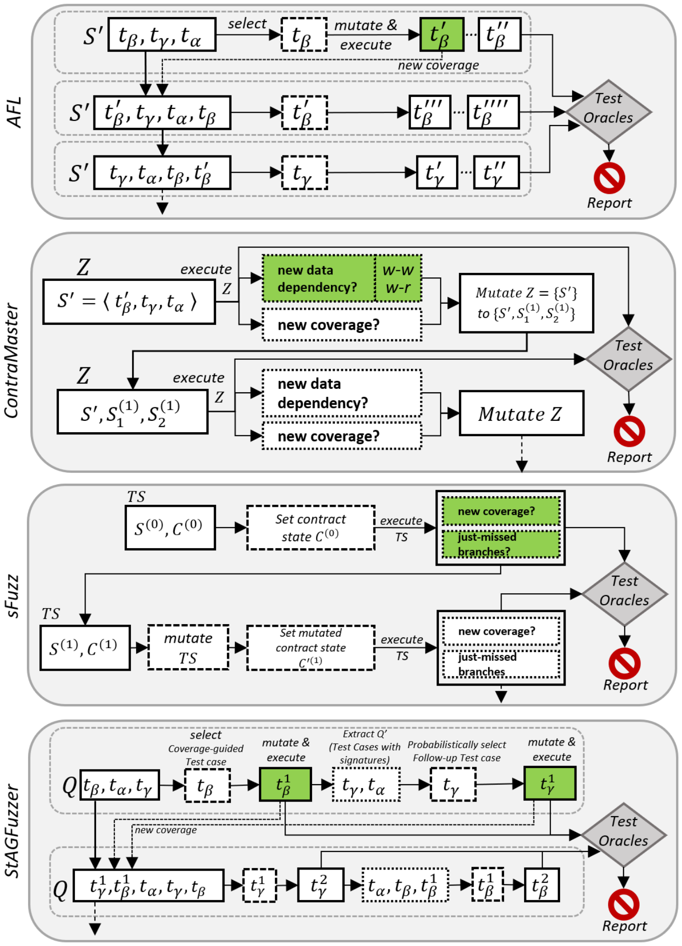

Fig. 2. Workflow of AFL, ContraMaster, sFuzz, and StAGFuzzer.

suites and adds them to a set Z. By executing each test suite, such as $S' = \langle t_\beta, t_\gamma, t_\alpha \rangle$ in Z, it observes S' to induce one write-read and one write-write data dependencies[1] on oB from t_β to t_γ. For each such dependency k (for $k = 1 \ldots 2$) from a test case $S'[i]$ (= t_β) to a test case $S'[j]$ (= t_γ) in S', it clones S' as a new test suite $S_k^{(1)}$, swaps the positions of t_β and t_γ in $S_k^{(1)}$, then mutates $S_k^{(1)}$. Next, it adds $S_k^{(1)}$ to Z. As such, Z contains 3 test suites: $\langle S', S_1^{(1)}, S_2^{(1)} \rangle$, where $S_1^{(1)} = \langle t^{(1)}_{1\gamma}, t^{(1)}_{1\beta}, t^{(1)}_{1\alpha} \rangle$ and $S_2^{(1)} = \langle t^{(1)}_{2\gamma}, t^{(1)}_{2\beta}, t^{(1)}_{2\alpha} \rangle$. Observe that Z contains 4 test case pairs that each calls β before γ: $\langle t_\beta, t_\gamma \rangle$, $\langle t_\beta, t^{(1)}_{1\gamma} \rangle$, $\langle t_\beta, t^{(1)}_{2\gamma} \rangle$, and $\langle t^{(1)}_{1\beta}, t^{(1)}_{2\gamma} \rangle$. Any mutant of t_γ will trigger ♠ if it happens after any mutant of t_β that has reached ♣. Then, the above construction and execution of Z repeats. Compared to AFL, ContraMaster improves the discovery of data dependency reversal and tests the same reversed dependency of k several times. But, a test suite, such as $S_k^{(1)} \in Z$, may contain some test cases (e.g., $t^{(1)}_{k\,\alpha}$) irrelevant to the intended test case pair in $S_k^{(1)}$, delaying the full execution of the pair. (ContraMaster has other features like adding mutants into Z by mutating the gas prices of test cases and resetting the contract states.)

1. In a sequence of execution traces, two events (p, q) form a write-read or write-write dependency on a state variable x if there is no other write on x in between p and q.

sFuzz [2] first generates a seed test suite $S^{(0)}=\langle t_\beta, t_\gamma, t_\alpha \rangle$ from S. It executes $S^{(0)}$, in which t_β, t_γ, and t_α discover new branches covering lines 28–31 and 34–35, 38–42, and 19–20, respectively. So, it generates the mutants t^1_β, t^1_γ, and t^1_α from t_β, t_γ, and t_α, respectively, and adds them to a new seed test suite $S^{(1)}$. It further identifies the set of *just-missed* branches[2] [2]: b_1 (lines 20–21), b_2 (lines 31–32), and b_3 (lines 42–43). It picks and mutates one test case from $S^{(0)}$ nearest to each of b_1, b_2, and b_3 (i.e., mutate t_α to t^2_α, t_β to t^2_β, and t_γ to t^2_γ), and adds the three mutants to $S^{(1)}$. So, $S^{(1)}$ becomes $\langle t^1_\beta$, t^1_γ, t^1_α, t^2_α, t^2_β, $t^2_\gamma \rangle$. sFuzz further shuffles and prunes $S^{(1)}$ and adds more test cases (taken and mutated from $S^{(0)}$) followed by executing the resultant test suite. It repeats the above process after assigning $S^{(0)} = S^{(1)}$.

To keep the example neat, we skip the shuffling, pruning, and addition process on $S^{(1)}$. Suppose the whole test session *TS* is $\langle S^{(0)}, S^{(1)} \rangle$. So, *TS* contains six test case pairs each calling β followed by γ: $\langle t_\beta, t_\gamma \rangle$, $\langle t_\beta, t^1_\gamma \rangle$, $\langle t_\beta, t^2_\gamma \rangle$, $\langle t^1_\beta, t^1_\gamma \rangle$, $\langle t^1_\beta, t^2_\gamma \rangle$ and $\langle t^2_\beta, t^2_\gamma \rangle$. Any mutant of t_γ will trigger ♠ if it happens after any of t^1_β or t^2_β that reaches ♣. sFuzz improves the chance of reaching just-missed branches by having higher concentrations of test cases nearest to them (e.g., both t^1_β and t^2_β in $S^{(1)}$ are near b_2). In the test suite $S^{(1)}$, test cases are independent of one another, and unrelated test cases may separate the two test cases in such a pair (e.g., t^2_α between t^2_β and t^2_γ). The pruning of $S^{(1)}$ may accidentally remove such pairs, and the test case addition to $S^{(1)}$ dilutes the concentration of test cases of these pairs in $S^{(1)}$.

StAGFuzzer iteratively creates pairs of test cases. After executing S, the three test cases t_α, t_β, and t_γ are found to access the following state variables: R_α={o, s}, R_β={o, s, a, oB}, and R_γ={o, uB, oB} for read accesses and $W_\alpha = \emptyset$, W_β={oB}, and W_γ={uB, oB} for write accesses, respectively.

Suppose S is shuffled into the seed queue $Q=\langle t_\beta, t_\alpha, t_\gamma \rangle$. Figure 3 summarizes StAGFuzzer's first six rounds on test case pair generation.

In the first round, StAGFuzzer mutates t_β into t^1_β (and moves t_β to Q's tail). Suppose t^1_β covers ♣ (lines 32–33), resulting in R^1_β = {o, s, a uB} and W^1_β = {s, uB}. As t^1_β finds a new basic block (♣), it is added to Q as the first element. (We note that the coverage-guided phase is a typical process in CGF fuzzers [5].) StAGFuzzer then picks one test case among the test cases in Q that do not call the entry function $tx(t^1_\beta)$ (i.e., t_γ and t_α) to produce a follow-up test case. For the pair $\langle t^1_\beta, t_\gamma \rangle$, uB $\in W^1_\beta$ with each access in {uB $\in R_\gamma$, uB $\in W_\gamma$} marks a *ctsv* signature occurrence on uB, which further marks an *itcv* signature occurrence on oB $\in W_\gamma$. Moreover, uB $\in R^1_\beta$ with the access in uB $\in W_\gamma$ marks a *ctsv* signature occurrence on uB, which further marks an *itcv* signature occurrence on oB $\in W_\gamma$. Similarly, for the pair $\langle t^1_\beta, t_\alpha \rangle$, s $\in R_\alpha$ carries a *ctsv* signature occurrence. As the pair $\langle t^1_\beta, t_\gamma \rangle$ contains more signature occurrences than $\langle t^1_\beta, t_\alpha \rangle$, t_γ is more likely to be picked for a mutation to produce t^1_γ. Suppose t^1_γ thus produced meets the condition in line 42 and triggers ♠. As t^1_γ covers line 43, StAGFuzzer adds it to Q as the first element.

2. A branch node has two outgoing branches. If only one outgoing branch is covered, the remaining one is called a just-missed branch.

Round	Q	Coverage-guided phase				Follow-up phase						Remarks
		mutation	coverage		Add to Q	mutation	Signature		coverage		Add to Q	
			♣	♠			*ctsv*	*itcv*	♣	♠		
Seed execution phase on $\langle t_\beta, t_\alpha, t_\gamma \rangle$		t_β	×	×	✓	No follow-up						
		t_α	×	×	✓							
		t_γ	×	×	✓							
1	$\langle t_\beta, t_\alpha, t_\gamma \rangle$	$t_\beta \rightarrow t_\beta^1$	✓	×	✓	$t_\gamma \rightarrow t_\gamma^1$	1: $r \rightarrow w$ (uB) 2: $w \rightarrow r$ (uB) 3: $w \rightarrow w$ (uB)	oB w.r.t. 1 oB w.r.t. 2 oB w.r.t. 3	×	✓	✓	t_β^1 : s = *true*, covers ♣. t_γ^1 triggers ♠.
2	$\langle t_\gamma^1, t_\beta^1, t_\alpha, t_\gamma, t_\beta \rangle$	$t_\gamma^1 \rightarrow t_\gamma^2$	×	×	×	$t_\beta^1 \rightarrow t_\beta^2$	4: $r \rightarrow w$ (uB) 5: $w \rightarrow r$ (uB) 6: $w \rightarrow w$ (uB)	s w.r.t. 4 s w.r.t. 5 s w.r.t. 6	×	×	×	condition in line 42 not met.
3	$\langle t_\beta^1, t_\alpha, t_\gamma, t_\beta, t_\gamma^1 \rangle$	$t_\beta^1 \rightarrow t_\beta^3$	×	×	×	No follow-up						`required`(s=*false*) fails.
4	$\langle t_\alpha, t_\gamma, t_\beta, t_\gamma^1, t_\beta^1 \rangle$	$t_\alpha \rightarrow t_\alpha^1$	×	×	✓	$t_\beta^1 \rightarrow t_\beta^4$	7: $r \rightarrow w$ (s) 8: $w \rightarrow r$ (s) 9: $w \rightarrow w$ (s) 10: $w \rightarrow r$ (a)	uB w.r.t. 7 uB w.r.t. 8 uB w.r.t. 9 uB,s w.r.t. 10	✓	×	×	t_α^1 : s=*false*, modifies qu, a, oB. t_β^4 : s = *true*, covers ♣.
5	$\langle t_\alpha^1, t_\gamma, t_\beta, t_\gamma^1, t_\beta^1, t_\alpha \rangle$	$t_\alpha^1 \rightarrow t_\alpha^2$	×	×	×	$t_\beta^1 \rightarrow t_\beta^5$	11: $r \rightarrow w$ (s) 12: $w \rightarrow r$ (s) 13: $w \rightarrow w$ (s) 14: $w \rightarrow r$ (a)	uB w.r.t. 11 uB w.r.t. 12 uB w.r.t. 13 uB,s w.r.t. 14	×	×	×	t_α^2 : s=*false*, modifies q, a, oB. t_β^5 : does not cover ♣.
6	$\langle t_\gamma, t_\beta, t_\gamma^1, t_\beta^1, t_\alpha, t_\alpha^1 \rangle$	$t_\gamma^1 \rightarrow t_\gamma^3$	×	✓	×	Similar follow-up test case selection...						t_γ^3 triggers ♠.

t	*reads/writes* on state variables					
	o	s	qu	a	oB	uB
t_β	*r*	*r*		*r*	*r,w*	
t_α	*r*	*r*				
t_γ	*r*				*r,w*	*r,w*
t_β^1	*r*	*r,w*		*r*		*r,w*
t_γ^1	*r*				*r,w*	*r,w*
t_γ^2	*r*				*r,w*	*r,w*
t_β^2	*r*	*r*				
t_β^3	*r*	*r*				
t_α^1	*r*	*r,w*	*w*	*w*	*w*	
t_β^4	*r*	*r,w*		*r*		*r,w*
t_α^2	*r*	*r,w*	*w*	*w*	*w*	
t_β^5	*r*	*r*		*r*	*r,w*	
t_γ^3	*r*				*r,w*	*r,w*

Fig. 3. STAGFUZZER test case generation.

In the second round, the first element Q_0 in Q is t_γ^1. As such, STAGFUZZER creates and executes a mutant t_γ^2 from t_γ^1, which fails to meet the condition in line 42 as `uB` keeps no account with a positive balance yet, and produces R_γ^2 = R_γ and W_γ^2 = W_γ. STAGFUZZER then chooses one among t_α, t_β, and t_β^1 to generate a follow-up test case. t_β^1 creates more signature occurrences with t_γ^2 and is more likely to be chosen to produce the follow-up test case (t_β^2 for t_γ^2). t_β^2 is then executed.

In the third round, STAGFUZZER generates and executes a mutant t_β^3 from t_β^1 (and moves t_β^1 to Q's tail). Since `s` = *true*, the condition in line 29 fails, resulting in R_β^3 = {`o`, `s`} and W_β^3 = $\emptyset$. STAGFUZZER does not produce any follow-up test case because all test cases in Q not calling β cannot mark any signature occurrences on any state variables with t_β^3.

In the fourth round, STAGFUZZER mutates t_α to generate t_α^1 and executes t_α^1. As t_β^1 has reached ♣ (and sets `s` = *true*), t_α^1 discovers lines 21–24. STAGFUZZER adds t_α^1 to Q as the first element. Its follow-up test case will more likely be a mutant of t_β^1 because of having more *ctsv* and *itcv* signature occurrences in $\langle t_\alpha^1, t_\beta^1 \rangle$ than other pairs.

Figure 3 also shows the fifth round (which performs like the fourth round) and the sixth round (in which t_α^1 has a higher chance of generating the follow-up test case).

STAGFUZZER is thus more likely to generate mutants originating from t_γ after observing mutants originating from t_β. We also see that mutants originating from t_α (in the 4th and 5th rounds) lead to the generation of mutants originating from t_β to cover ♣, which enables the next mutant originating from t_γ to trigger ♠. All these factors increase the probability of triggering ♠.

Furthermore, STAGFUZZER treats each parameter value (except for addresses) as a bit/ASCII string and randomly alters, adds, or removes some bits/ASCII characters, followed by a probability of resetting the whole bit/ASCII string to zero or a null string. Suppose, in the fourth round of the above example, both t_α^1 and t_β^4 use ”” (null string) as the parameter values for `_a` and `_r`. In this case, t_β^4 passes the condition in line 31, reaching ♣.

2 FURTHER DISCUSSION ON THE RESULTS

An *itcv* signature relates a cross-transactional data flow (*ctsv* signature) on one state variable to another state variable. The success of *itcv* signatures indicates that future data flow techniques for fuzzing smart contracts may explore cross-variable definitions and same-variable definition-and-use in generating, selecting, prioritizing, or evolving test cases and test suites.

A fuzzer can boost its effectiveness by using strategies with clear phase alternation and separations. Figure 6 and Table V, from the manuscript, show that increasing the number of test cases from 600 to 2,000 has a small effect on enlarging the advantages of branch discovery over SFUZZ. STAGFUZZER$_{2000}$ still misses reaching some branches that SFUZZ can reach and vice versa, reconfirming the just-missed branch strategy of SFUZZ to be effective. They show that during the stagnation of branch discovery, a strategy with higher throughput to discover branches from another perspective may further boost the performance of our fuzzer.

The test case generation and execution rate of STAGFUZZER is 10× lower compared to SFUZZ. (The SFUZZ tool uses batch-mode and offline trace analysis.) Our test framework mines one new block for each test case, which is also slow. Increasing the throughput while not adversely affecting the fault detection effectiveness should be a high-priority future work.

Note: This supplementary document has been placed in the GitHub repository [3] which contains the code of STAGFUZZER.

REFERENCES

[1] B. Jiang, Y. Liu, and W.K. Chan, ”CONTRACTFUZZER: Fuzzing Smart Contracts for Vulnerability Detection,” *in Proc. of ASE'18*, pp. 259–269, 2018.

[2] T. Nguyen, L. Pham, J. Sun, Y. Lin, and M. Tran, ”sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts,” in *Proc. of International Conference on Software Engineering (ICSE '20)*, pp. 778–788, 2020.

[3] STAGFUZZER. https://github.com/imrash/StAGFuzzer (accessed Feb. 2023).

[4] H. Wang, Y. Liu, Y. Li, S. Lin, C. Artho, L. Ma, and Y. Liu, ”Oracle-Supported Dynamic Exploit Generation for Smart Contracts,” *TSDC*, Vol. 19, No. 3, pp. 1795–1809, 2022.

[5] M. Zalewski, American Fuzzy Lop http://lcamtuf.coredump.cx/afl (accessed Feb. 2023).