Supplementary Material for the ESEC/FSE 2023 Paper: "SmartFix: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models"

Sunbeom So[*]
Korea University
Republic of Korea
sunbeom_so@korea.ac.kr

Hakjoo Oh[†]
Korea University
Republic of Korea
hakjoo_oh@korea.ac.kr

1 PATCH COMPONENT EXTRACTION RULES (SECTION 3.1.1)

Given an original contract s_0 to repair and its vulnerability report VR_0, $\textsc{Extract}(s_0, VR_0)$ outputs $C = C_{\mathsf{IO}} \cup C_{\mathsf{RE}} \cup C_{\mathsf{EL}} \cup C_{\mathsf{SU}} \cup C_{\mathsf{TX}}$, where each C_v is a set of patch components for fixing bugs whose type is v (Section 3.1.1). For simplicity, we focus on explaining representative rules for each bug type.

C_{IO} is obtained by adding a corresponding guard statement right before each reported line (e.g., $\mathsf{Insert}(l, \texttt{require}(\mathsf{a} + \mathsf{b} \geq \mathsf{a})))$ or by changing comparison operators within functions that contain reported lines using the Rep template. While reasons for IO bugs would be mostly due to missing runtime checks, we also support changes of binary relations, because IO bugs may happen due to flawed guards as well (e.g., CVE-2018-11411, Figure 2).

C_{SU} is obtained as follows for rectifying faulty access controls: adding authority guards to functions that contain lines reported to be vulnerable to SU, adding authority guards to functions that contain assignments to address-typed l-values, negating conditions on address-typed expressions in modifiers whose address-typed expressions are used, and changing functions that may be mistakenly named to constructors using the ToCnstr template.

C_{EL} is obtained in a similar way to C_{SU} as root causes of EL and SU are often similar, but we consider one more repair template to handle cases unrelated to flawed access controls.[1]

C_{TX} is obtained by replacing `tx.origin` by `msg.sender` in functions (or modifiers) that contain lines reported to be vulnerable; it is recommended to use `msg.sender` instead of `tx.origin` for authorizing accounts [3].

C_{RE} is obtained based on two well-known schemes [1, 2] that aim to prevent RE bugs. The first scheme is to make all functions comply with so-called checks-effects-interactions pattern [1, 2]. When generating patch components based on this scheme, we use Move template in order to move assignments, which are behind `call` statements, in front of `call` statements. The second scheme is the mutex-based scheme [2], where we apply `nonReentrant` modifiers to functions to prevent reentrancy itself. When generating patch components based on the second scheme, we consider multiple groups of functions to apply `nonReentrant` modifiers. This is because relying on only one group (i.e., a single strategy) is more likely to generate incorrect patches. For example, if we simply add `nonReentrant` modifiers to all functions for fixing RE bugs, a patched contract may contain deadcode (e.g., the contract in [2], which contains `withdraw` and `getFirstWithdrawalBonus` functions). Examples of function groups include: public/external functions that contain external calls or may manipulate state variables, and public/external functions that do not contain internal function calls among the functions in the previous group.

2 FILTERING BENIGN ALARMS (SECTION 4)

Filtering Benign IO Alarms. We ignore benign IO alarms via pattern matching. For example, we consider the addition at line 7′ in Figure 2 is benign; even if the overflow occurs, it does not affect changes of global states and the execution is properly handled (i.e., terminated by `return false`).

Filtering Benign RE Alarms. We designed a procedure for postprocessing RE-related alarms raised by VeriSmart [4, 5]. The latest VeriSmart [4] detects two types of RE bugs: RE_{eth} (reports alarms if ethers can be stolen due to reentrancy), and RE_{state} (reports alarms if states can be modified by untrusted users in reentering executions). However, we found that a contract that is safe w.r.t. either of the two safety conditions is often safe from vulnerabilities due to reentrancy. For example, the second code snippet in [2], which is safe w.r.t. RE_{eth} but unsafe w.r.t. RE_{state} (i.e., the reentrancy itself is possible), does not have reentrancy vulnerabilities.

Based on the observation, given two types of RE-related alarms (RE_{eth}, RE_{state}) raised by VeriSmart, we postprocess them to produce a final vulnerability report (VR) as follows.

- If there were non-zero queries (verification targets) for RE_{state} and all of their safety is proven, we do not include RE alarms in VR (i.e., we consider that the analyzed contract is safe from RE).
- If there were non-zero queries for RE_{eth} and all of their safety is proven, we do not include RE alarms in VR.
- If there were non-zero queries for RE_{eth} and some of their safety is not proven, we report RE_{eth} alarms as RE alarms; in this case, we exclude RE_{state} alarms, in order to avoid to consider fixing the other types of bugs as a minor impact when computing safety scores (Section 3.2).
- Otherwise, we report RE_{state} alarms, if any, as RE alarms.

Note that, following the above procedure, SmartFix guarantees patch safety for RE, in that the safety of output patches is formally verified for at least one kind of safety conditions for RE (the first two cases).

[*]Current affiliation: Gwangju Institute of Science and Technology (GIST)
[†]Corresponding author
[1]E.g., https://github.com/smartbugs/smartbugs-curated/blob/main/dataset/access_control/wallet_02_refund_nosub.sol

3 BUG-INJECTION PATTERNS FOR RE AND TX BENCHMARKS (SECTION 5.1)

We describe bug-injection patterns for constructing RE Bench and TX Bench (Section 5.1). We crafted our benchmarks by applying these patterns to deployed smart contracts. When necessary, we applied multiple patterns to a contract.

3.1 RE Bench

Pattern 1. Changing Order. We inject RE vulnerabilities by moving statements, which may alter global states, behind external calls. This pattern is for breaking checks-effects-interactions pattern [1, 2]. An example is the following, where the statement at line 3 is moved to line 4′.

```
1   function withdraw(uint _amount) {
2     require(balances[msg.sender] >= _amount);
3   (-) balances[msg.sender] -= _amount;
4     msg.sender.call.value(_amount)();
4'  (+) balances[msg.sender] -= _amount;
5   }
```

Pattern 2. Removing Mutex-Related Conditionals. We remove mutex-related modifiers for preventing RE. An example is the following.

```
1   (-) function withdraw(uint _amount) nonReentrant {
1'  (+) function withdraw(uint _amount) {
2     require(balances[msg.sender] >= _amount);
3     msg.sender.call.value(_amount)();
4     balances[msg.sender] -= _amount;
5   }
```

Pattern 3. Replacing Statements for Sending Ethers. Given a `transfer` or a send statement for transferring ethers, we replace each of them with a `call` statement that is another one for sending ethers, because reentrancy attacks by `transfer` and send are currently hardly possible due to the gas limit for them. A mutation example is the following.

```
1   function withdraw(uint _amount) {
2     require(balances[msg.sender] >= _amount);
3   (-) msg.sender.transfer(_amount);
3'  (+) msg.sender.call.value(_amount)();
4     balances[msg.sender] -= _amount;
5   }
```

3.2 TX Bench

Since TX happens when `tx.origin` (the initial transaction sender in a call-chain) instead of `msg.sender` (the immediate transaction sender) is used to recognize certified users, we inject TX vulnerabilities by replacing `msg.sender` by `tx.origin` in statements for identifying privileged users. An example is the following.

```
1   modifier onlyOwner {
2   (-) require (msg.sender == owner);
2'  (+) require (tx.origin == owner);
3     _;
4   }
```

REFERENCES

[1] [n. d.]. An overview on reentrancy bugs from SWC Registry. https://swcregistry.io/docs/SWC-107. Accessed: August 2023.

[2] [n. d.]. Ethereum Smart Contract Best Practices. https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/known_attacks/. Accessed: August 2023.

[3] [n. d.]. Solidity Documentation. https://docs.soliditylang.org. Accessed: August 2023.

[4] [n. d.]. VeriSmart: a formal verification tool for Solidity smart contracts. https://github.com/kupl/VeriSmart-public. Accessed: August 2023.

[5] S. So, M. Lee, J. Park, H. Lee, and H. Oh. 2020. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 718–734. https://doi.org/10.1109/SP.2020.00032