The Journal of Systems and Software 216 (2024) 112107

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

ReenRepair: Automatic and semantic equivalent repair of reentrancy in smart contracts☆

Ruiyao Huang [a,b], Qingni Shen [a,b,*], Yuchen Wang [a,b], Yiqi Wu [a,b], Zhonghai Wu [a,b], Xiapu Luo [c], Anbang Ruan [d,b]

[a] School of Software and Microelectronics, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, 100871, China
[b] PKU-OCTA Laboratory for Blockchain and Privacy Computing, Beijing, China
[c] The Hong Kong Polytechnic University, Hong Kong, China
[d] Octa Innovations Ltd., Beijing, China

ARTICLE INFO

Keywords:
Smart contracts
Reentrancy
Program repair
Template-based repair
Semantics

ABSTRACT

Reentrancy, the most notorious vulnerability in smart contracts, has attracted extensive attention. To eliminate reentrancy before deploying contracts, there is a need to locate and repair the contracts. However, existing tools suffer from false positive localization, original semantics destruction, and high gas overhead. In this work, we propose a template-based gas-optimized reentrancy repair method with semantic maintenance. We avoid false positive locations from verifying the attack's effectiveness, using connectivity and read–write dependencies. We design the semantic equivalence check algorithm based on the def-use chain. We optimize the lock and reordering templates for reentrancy repair and add semantic maintenance operations. We implement our tool, ReenRepair, and compare it with two state-of-the-art detection tools and two repair tools. The results show that ReenRepair yields good location precision, the highest repair rate, and the lowest gas overhead. All semantic changes caused by lock and 89.66% of semantic changes caused by reordering are successfully maintained.

1. Introduction

Ethereum, a blockchain platform that supports smart contracts, has undertaken many cryptocurrency trading businesses. However, due to the immutability of blockchain, if smart contracts have security vulnerabilities, huge losses will be irreparable. Reentrancy is one of the most notorious vulnerabilities, derived from the "TheDAO" attack. "TheDAO" contract was exploited in June 2016 resulting in a direct loss of US$160 million in Ether at the time (Zhou et al., 2020). The vulnerable function updated the storage variable after the message call. The attacker exploited this and devised the attacker's contract to re-enter the vulnerable function multiple times to withdraw 'money' through the message call. Similar attacks did not stop: Uniswap and Lendf.me were hacked in April 2020, attacker stole $25 million worth of cryptocurrency (Paganini, 2020). According to statistics in Zhou et al. (2020), as shown in Fig. 1, reentrancy causes the highest direct financial loss.

The reentrancy has prompted research on enhancing the security of smart contracts, based on Automatic Program Repair (APR), because the reentrancy vulnerability in the smart contract is similar to the concurrent bug in the program. Generally, APR consists of three steps: fault localization, patch generation and patch verification (Gazzola et al., 2019). The localization step identifies the locations where the bug exists. The patch step generates fixes and modifies the program where the location step provides. The verification step checks if the synthesized fix repaired the software. In the fault location phase, APR tools integrate the existing detection tools or design their custom location algorithms. Elysium (Ferreira Torres et al., 2022) integrates Oyente (Luu et al., 2016) as the detection tool, which detects reentrancy through the feasibility of the call loop only. sGuard (Nguyen et al., 2021) and Smartshield (Zhang et al., 2020b) use their custom location algorithms which are similar to Securify (Tsankov et al., 2018), detecting reentrancy through the pattern "state change after message call" only. Such conservative patterns of reentrancy lead to false positives (FPs) in location up to 605 FPs in 608 detection results (Xue et al., 2020). In the patch generation phase, APR tools are divided into search-based and template-based. The search-based approach generates mutations of the bug code to search which can repair successfully. SCRepair (Yu et al., 2020) uses random mutations to generate candidate

☆ Editor: Justyna Petke.

* Corresponding author.
E-mail addresses: hryao@pku.edu.cn (R. Huang), qingnishen@ss.pku.edu.cn (Q. Shen), wyc247517@pku.edu.cn (Y. Wang), 2101210430@stu.pku.edu.cn (Y. Wu), wuzh@pku.edu.cn (Z. Wu), daniel.xiapu.luo@polyu.edu.hk (X. Luo), ar@8lab.cn (A. Ruan).

https://doi.org/10.1016/j.jss.2024.112107
Received 28 November 2023; Received in revised form 12 April 2024; Accepted 20 May 2024
Available online 22 May 2024

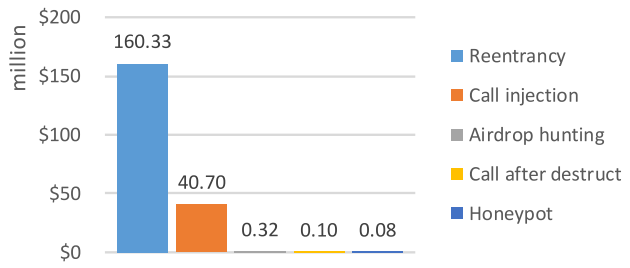


Fig. 1. Losses of smart contract vulnerabilities (Zhou et al., 2020).

patches and selects the patches that can pass all test cases as correct patches. However, the repair success rate for reentrancy is only 75%. The repair rate is low because there is no target vulnerability during mutation. The template-based approach defines repair templates for specific vulnerabilities. sGuard (Nguyen et al., 2021) and Elysium (Ferreira Torres et al., 2022) design templates for adding locks at the source code and bytecode level, respectively. However, the lock template used in sGuard modifies lock storage variables for every reentrancy function, without any optimization leading to high gas costs. Smartshield (Zhang et al., 2020b) designs a template for adjusting instructions order at the bytecode level, which mitigates gas problems. In the patch verification phase, APR tools use test cases (history transactions in blockchain) or static analysis to verify the contract behavior. However, the existing methods lack the maintenance of semantic equivalence, resulting in problems such as deadlock after repair. Smartshield checks semantic conflicts in the patch verification step, but the semantic maintenance is left to the developers.

To eradicate the reentrancy, the contract repair should be completed before the contract is deployed on the chain, which presents three challenges. Firstly, false positives (FPs) in location. FPs widely exist in the detection tools, resulting in introducing unnecessary patches that add overhead and new risks. Secondly, high gas overhead in patch generation. The repair process inevitably introduces additional instructions. The indiscriminate use of patches causes excessive gas overhead. Thirdly, lack of semantic equivalence maintenance after patch verification, resulting in breaking the original function of the contract after repair. Our work hopes to introduce semantic equivalence maintenance to correct semantic changes in patch generation.

In this paper, we seek to address the above challenges by proposing a framework for repairing reentrancy in smart contracts, named ReenRepair, which is able to repair reentrancy and keep semantic equivalent with low gas overhead. The main contributions of the paper are summarized as follows.

- We propose a novel smart contract repair framework ReenRepair, which targets to locate reentrancy and repair reentrancy semantically equivalent.
- We model the reentrancy more fine-grained to gain a high-precision location. First, we summarize two false positive scenarios that do not have reentrancy attack effects, then model the reentrancy with read–write dependency and path connectivity to reduce false positives in reentrancy localization (Section 4.2).
- We present the semantic equivalence measurement approach. We give the semantic equivalence conditions based on the execution process of the smart contract, then devise an algorithm to check the semantic equivalence between the original contract and its patched version based on the CFG by using the def-use chain to track the storage variables used by state update instructions (Section 4.3).
- We present two gas-optimized repair templates for reentrancy: the bit-lock template and the reordering template. To reduce the gas overhead, we optimize the SSTORE instruction in the templates by changing usage scenarios and reducing the frequency (Section 4.4). The templates are equipped with semantic maintenance operations which correct the semantics by restoring the original def-use relationship when the semantics change after being patched.
- We evaluate ReenRepair based on real-world contracts and synthetic contracts (Section 5). Compared to the state-of-the-art, our approach achieves the highest repair rate 81% and lower gas overhead by reducing at least 30% for one patch.

2. Background

In this section, we provide background on smart contracts, Ethereum virtual machine (Section 2.1), the gas mechanism in Ethereum (Section 2.2) and the reentrancy in smart contracts (Section 2.3).

2.1. Smart contract and Ethereum virtual machine

The smart contract is an immutable piece of code that runs on the blockchain. In Ethereum, the smart contract is widely used in financial issues, recording and trading digital assets. The digital asset data is usually stored permanently on the blockchain. Persistent data is referred to as storage, represented by state variables. The transaction of digital assets involves the interaction of multiple account addresses, including the contract address. Message calls are a way to allow communication between contracts, implementing invokes of other contract functions or transfers ethers. However, there are security risks when interacting with contracts at unknown addresses because of unknown behaviors, such as reentering the caller contract again or running out of gas.

The execution environment for smart contracts is called Ethereum Virtual Machine (EVM) which runs the bytecodes compiled from smart contract source code. The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack items) is 256-bit. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model. Unlike stack and memory, which are volatile, storage is non volatile and is maintained as part of the system state (Wood, 2023). The instruction, which can also be called OPCODE, responsible for the storage variable is SLOAD and SSTORE, and the instruction responsible for message call is CALL.

2.2. Gas mechanism

In order to avoid issues of network abuse, all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of gas. For smart contracts, different types of instructions have different gas costs: Generally, the simple arithmetic instructions use low gas, such as ADD, NOT, AND (3 gas), and more complicated arithmetic instructions, such as SHA3 cost a little more (30 gas) (Wood, 2023). The storage-related instructions, SLOAD and SSTORE, cost very much. The gas varies with the blockchain version and instruction usage scenario, as shown in Table 1. The consumption of gas is determined by whether the storage has been accessed or written in the same transaction. In the Istanbul version, 800 gas per word for SLOAD, 20 000 gas per word for SSTORE when the storage value is set to non-zero from zero and 5000 gas per word for SSTORE when the storage value's zeroness remains unchanged or is set to zero. In the Berlin version, SLOAD may cost 2100 or 100 gas in different situations, SSTORE may cost 22 100 gas at most, 2900 at least (Vitalik Buterin, 2020). No matter which version, SLOAD is always cheaper than SSTORE, and changing non-zero storage variables is always cheaper than changing storage variables from zero to non-zero.

Table 1
Gas cost for SLOAD and SSTORE. Accessed: The storage slot has already been accessed. SSTORE: From zero to a non-zero value. SSTORE*: From a non-zero to a different non-zero value. SSTORE#: The storage slot has already been written.

OPCODE	Istanbul	Berlin	
		Not accessed	Accessed
SLOAD	800	2100	100
SSTORE	20 000	22 100	20 000
SSTORE*	5000	5000	2900
SLOAD+SSTORE*	5800	5000	3000
SSTORE*+SLOAD	5800	5100	3000
SSTORE#	800	100	100

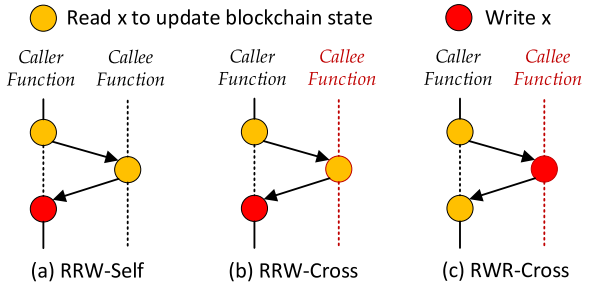


Fig. 2. Three types of reentrancy pattern.

```
mapping(address⇒uint) etherBalance;
function withdrawEther() public { //caller and callee
  if (etherBalance[msg.sender] > 0) { //read
    msg.sender.call.value(etherBalance[msg.sender])(""); //read
    etherBalance[msg.sender] = 0; //write
  }
}
function transferEther(address to, uint amount) public { //callee
  if(etherBalance[msg.sender] ⩾ amount) { //read
    etherBalance[to] += amount; //update other storages
    etherBalance[msg.sender] -= amount ;
  }
}
```

Fig. 3. RRW-Self and RRW-Cross contract snippet. Exploiting withdrawEther(), attackers can re-enter withdrawEther() through RRW-Self reentrancy and re-enter transferEther() through RRW-Cross reentrancy.

2.3. Reentrancy

Reentrancy attacks first appeared in "The DAO" contract in 2016. In existing reentrancy cases, we find that one reentrancy contains at least two executions of functions, and these functions' call relationships are nested. For the convenience of description, we take two function calls as an example. We denote the outer function by *caller function* and the inner function by *callee function*. The cause of reentrancy is similar to atomicity-violation problems in concurrency bugs. In smart contracts, the message call lets the callee function execute before the caller function fully executes. Thus, the storage variables reading in the caller or callee functions faced dirty reading problems. Based on the read–write sequence, there are three types of reentrancy shown in Fig. 2: RRW-Self, RRW-Cross, and RWR-Cross.

RRW-Self has been mentioned in plenty of papers (Tsankov et al., 2018; Rodler et al., 2019; De Moura and Bjørner, 2008; Luu et al., 2016; Zhang et al., 2020a; Yu et al., 2020). It is a reentrancy type attacked by re-entering the same function. In the caller function, the storage variable is read (R) and written (W) before and after the message call. In the callee function, the same storage variable is read (R) and utilized to transfer. As shown in Fig. 3, withdrawEther() has reentrancy and can be re-entered because the etherBalance[msg.sender] is updated after the message call.

RRW-Cross caught the attention of researchers (Rodler et al., 2019; Xue et al., 2020) and the community (Diligence, 2020) two years later

```
mapping(uint ⇒ uint) splits;
mapping(uint ⇒ uint) deposits;
function splitFunds (uint id) public { //caller
  address payable alice = ...; //ignore the details
  address payable bob = ...; //ignore the details
  uint depo = deposits[id];
  deposits[id] = 0;
  alice.call.value((depo*splits[id])/100)(""); //read
  bob.call.value((depo*(100 - splits[id]))/100)(""); //read
}
function updateSplit(uint id, uint split) public { //callee
  require(split ⩽ 100);
  splits[id] = split; //write
}
```

Fig. 4. RWR-Cross code snippet. Exploiting splitFunds(), attackers can re-enter updateSplit() through RWR-Cross reentrancy.

after the first discovery of RRW-Self. It is a type of reentrancy attacked by re-entering a different function. The reading and writing pattern of RRW-Cross is the same as RRW-Self. As shown in Fig. 3, withdrawEther() has reentrancy, and attacker can re-enter transferEther() to modified etherBalance[to].

RWR-Cross was first discovered in January 2019 (ChainSecurity, 2019). It is also a type of re-entering different functions. The storage variable is read (R) twice before and after the message call in the caller function, and the same storage variable is written (W) in the callee function. As shown in Fig. 4, splitFunds() has reentrancy, and the attacker can re-enter into updateSplit() to change the splits[id], leading to increase/decrease the value of bob receiving.

The repair strategy for reentrancy, such as the lock strategy, has no difference in patching the three types above, which will introduce gas redundancy and even deadlock. We will discuss these in Section 3.

3. Motivation

3.1. Reentrancy repair problems

In this section, we present three problems of reentrancy repair in existing works.

Reentrancy localization. Localization is the first step in the repair. Accurate localization with a detailed description of the location is a prerequisite for fixing. False positive localization introduces unnecessary patches resulting in increasing extra gas overhead and the risk of semantic changes. Detailed localization information includes not only the function and statements but also for the description of the attack process, such as the attack execution sequences. This allows different strategies to generate the appropriate patches.

Semantic equivalence. Semantics describe formally what a program means and are presented in program behavior. In the blockchain, the behavior is reflected in terms of success or failure of transaction execution, modification of state variables, transfer amounts, generated message calls, etc. The patch introduced for repair has the potential to change the semantics. Existing tools lack detecting the semantic equivalence after patching, and correcting the semantically changed parts if necessary. In the case of locking, for example, the semantic change is reflected in the deadlock generated after locking, which causes the transaction execution to fail. In the case of modifying the statements, the new order of the statement may lead to the final state of the blockchain change, such as different storage values, and different transfer amounts.

Take the lock strategy as an example. As shown in Fig. 5, buyTokens() and isGoalReached() are both detected to have reentrancy, then sGuard add nonReentrant modifier for both functions, without semantically equivalence check. However, buyTokens() invokes isGoalReached(); thus after adding nonReentrant modifiers, the execution of buyTokens() will be reverted because of deadlock, and buyTokens() will

```
1  modifier nonReentrant_() {
2      require(!locked_);
3      locked_ = true;
4      _;
5      locked_ = false;
6  }
7  function buyTokens() nonReentrant_
8      payable public {
9      require(!isFinished);
10     require(isContractActive());
11     require(!isGoalReached()); //DEAD LOCK
12     ...
13 }
14 function isGoalReached() nonReentrant_
15     public returns (bool) {
16         return issuedTokens ⩾ _goal;
17 }
```

Fig. 5. Deadlocks in sGuard Repair.

```
1  function withdrawToken() public {
2    if (etherBalance[msg.sender] > 0) {
3      etherBalance[msg.sender] = 0;
4      msg.sender.call.value(etherBalance[msg.sender])("");
5      etherBalance[msg.sender] = 0;
6    }
7  }
8
9  function Collect(uint _am) public {
10     if (balance[msg.sender] ⩾_am) {
11       balance[msg.sender] -=_am;
12       if(msg.sender.call.value(_am)("")) {
13           balance[msg.sender] -=_am;
14           LogFile.AddMessage(msg.sender,_am,"Collect");
15       }
16     }
17 }
```

Fig. 6. Semantic changes caused by reordering. The red statements represent the reordered statements. The yellow statements represent the statements whose semantics are affected by the reordered statements. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

never succeed. To keep semantic equivalence, deadlock is one of the problems to avoid. In this case, only the lock for isGoalReached() is necessary. Because the reentrancy in buyTokens() results from invoking isGoalReached(), the lock for buyTokens() should be removed.

In reorder strategy, when the write statement is moved, the storage variable value is changed in advance, leading to the value after the moved statement changes. In Fig. 6 withdrawToken() function, the etherBalance[msg.sender] at line 4 reads 0 after reordering, causing msg.sender to fail to receive the transfer with the correct value. In Fig. 6 Collect() function, the balance[msg.sender] balance is also wrongly deducted when the call transfer fails in line 12 after reordering. In order to maintain semantic equivalence, it is necessary to correct the affected variable value by storing the storage variable value into a local variable before the moved write statement and retrieving for the affected variable.

Gas Overhead. The added patch introduces additional gas overhead, increasing transaction execution costs. Besides the unnecessary patches introduced due to false positives, the necessary patches still cost a lot of gas in the current implementation. In the case of locking, nonReentrant modifier, implemented as shown in Fig. 5, is added to all functions with reentrancy, no matter whether the caller function or the callee function. Both locks seem necessary, but some operations in the nonReentrant modifier are unnecessary. In the callee function, the assignment to the lock is unnecessary because the lock has been set to true in the caller. Additional assignments will introduce more SSTORE instructions, one of the expensive instructions in EVM, resulting in more gas costs.

An intuitive idea to optimize gas usage is to use fewer SSTORE instructions, like the reorder strategy. The reorder strategy naturally consumes less gas because patches are usually only move statements, without introducing new SSTORE instructions. Another idea is to find an alternative to SSTORE. However, the memory set instruction, MSTORE, is not adapted to fixing the reentrancy scenario because the memory will be reset, and the lock value stored in memory will be refreshed when invoking a function through the message call. Thus, we have to use SSTORE, but only when it is necessary.

3.2. Reentrancy repair goals

Based on the three problems in reentrancy repair mentioned above, combined with the effectiveness of the patch, indeed avoid reentrancy occurring, three indicators are summarized to measure the repair result:

Precision. To guarantee the precision of reentrancy location, the reentrancy attack effects should be paid more attention to avoid false positives: whether the storage modification or transfer can be executed as the attacker expects; whether such effects only occur in the reentrancy do not occur in the normal sequential execution scenario.

Effectiveness and semantic equivalence. The effectiveness of the reentrancy patch means that the smart contract with the patch can fix the reentrancy. At the same time, the original semantics of the smart contract should not be modified during repair, which also means the repair process is semantically equivalent.

Low Gas overhead. Gas determines the success of function execution and the consumption of ethers. The gas overhead of the patch should be as low as possible to satisfy the two goals above.

4. Design of ReenRepair

4.1. Overview

In this section, we introduce our ReenRepair framework, which locates and repairs reentrancy automatically. ReenRepair consists of two components: reentrancy localization and reentrancy repair. Fig. 7 depicts the overall architecture of ReenRepair.

Reentrancy localization. To locate reentrancy, we use the read–write pattern extracted from real cases in Section 3 and the connectivity of the reentrancy path. All execution paths in a contract are depicted in CFG, constructed from the EVM bytecode using symbolic execution, by which we could extract the conditional expression for later analyzing the connectivity of the path. During the process of CFG construction, armed with taint tracking technology, we can track the storage variables to extract the read–write dependency in a single path. To solve the cross type of reentrancy, we construct the attack path based on the read–write dependency and regroup the conditional expression in the caller function and the callee function to check the connectivity of the attack path.

Reentrancy repair. Based on the lock and reordering ideas, we designed two gas-optimized repair templates by optimizing the SSTORE usage: the bit-lock template and the reordering template. In the template, we reduce the usage of SSTORE and create pre-conditions for SSTORE to run with low gas overhead. To ensure the semantics of the contract are unchanged after repair, we design an algorithm to check semantic equivalence by the def-use chain. When semantics are changed after repair, we use semantic equivalence maintenance operation to correct the patched contract. The bit-lock templates will remove the lock pairs in the original function invocation chain to avoid deadlock; the reordering template will add local variables to keep the def-use relationship unchanged in the affected area.

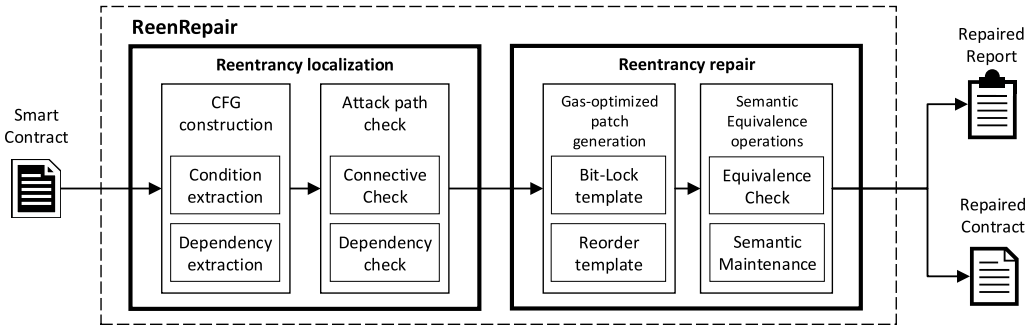


Fig. 7. Overview of our framework ReenRepair.

Table 2
Reentrancy location pattern. R: read; W: write; C: message call; U: update; Dependency: the dependency in the Caller and Callee; Attack Path: The path that must be connective in reentrancy; In the same line, R/W reads or writes the same storage variable.

Type	Caller	Callee	Dependency	Attack path
RRW Self	R-C-W	R-C-W	C DepOn R	R-C-R-C-W
RRW Cross	R-C-W	R-U	C DepOn R U DepOn R	R-C-R-U-W
RWR Cross	R-C-R-U	W	C DepOn R U DepOn R	R-C-W-R-U

4.2. Reentrancy localization

Main problems in reentrancy location are false positives. The reason is that the located reentrancy will not cause an actual attack effect. The first false positive scenario is that the expected storage modification or transfer cannot be executed, such as a function with locks but is still detected as containing reentrancy; the second false positive scenario is that the reentrancy execution result is no different from the normal sequential execution of the functions.

To ensure that the reentrancy located constitutes an attack effect, first, we summarize the instructions related to the execution effect. The updating on the blockchain state contains three categories: modifying storage variables (SSTORE), writing logs (LOG1, etc.) and changing account balances (CALL, etc.). Second, we use *connectivity* and *read–write dependency* to prevent the false positives mentioned above. Connectivity is used to describe whether the execution path from the beginning to the effect instructions in reentrancy is connective. To make the path connective, the branch conditions before effect instructions must be all satisfied. Dependency includes data dependency and control dependency. Data dependency means that the effect instructions use the read value as an argument, such as the call instruction depends on splits[id] in Line 8 in Fig. 4. Control dependency means that the read value is used in a condition on the way to effect instructions, such as the call instruction depends on etherBalance[msg.sender] in Line 3 in Fig. 3.

For the three reentrancy types in Section 2.1, the reentrancy location patterns are shown in Table 2. Take the RRW-Cross line as an example. There exists a subsequence shaped like ‘read, message call, write’ (R-C-W) in the caller function and a subsequence shaped like ‘read-update’ (R-U) in the callee function, where the instruction of message call and update both depend on the same read value. In Fig. 3, the caller contains R in Line 3, C in Line 4, and W in Line 5. The callee contains R in Line 9 and U in Line 10. The attack path is composed of the caller and the callee, and the callee is inserted after the caller’s message call.

RRW reentrancy happens when dirty reading occurs in the callee function. The storage variable has not been written in caller but read in callee, and the storage value is used to update the blockchain state. In Self reentrancy, the caller and the callee are the same. Thus, the updation is specialized to message call. RWR reentrancy happens when dirty reading occurs in the caller function. The second read result of the storage variable in the caller is tampered with by the callee.

To check the reentrancy pattern in the contract, there are two main steps: (1) CFG construction to get caller and callee paths and the conditions and dependency in them; (2) Attack path check to combine the caller and callee paths and check whether the reentrancy path is connective and satisfies the pattern. In the implementation, we use the symbolic execution method to build CFG for contracts and to get the branch conditions based on the symbolic execution engine proposed by Luu et al. (2016) using Z3 SMT solver (De Moura and Bjørner, 2008). We add our own taint tracking rules into the CFG construction process to get the read–write dependency on the path through the instructions such as SLOAD and SSTORE, LOG, CALL.

Taint Engine. Taint tracking is a technique for monitoring the flow of information through the program. Any program value whose computation depends on data derived from a taint source is considered tainted. Based on such properties, we can track data dependency and control dependency in our approach. A taint policy determines exactly what sorts of operations introduce new taint, how taint flows as a program executes, and what checks are performed on tainted values (Schwartz et al., 2010).

Based on the taint policy, our taint engine includes three parts: taint creation, propagation and sink. The taint creation and propagation occur when the instructions are symbolically executing. The taint creation occurs when SLOAD is executing, and the taint ID is based on the execution pc at that time. The taint propagation occurs between stack, memory and storage depending on the instructions. The propagation rules are detailed in Table A.8 in Appendix. The taint sink will be checked when SSTORE, CALL and other state updation instructions execute. Fig. 8 demonstrates an example of taint creation and propagation. Before executing SLOAD, the top elements in the stack without any taint and the second element with one taint marked as ‘0×...’. After executing SLOAD, the new taint is introduced and named ‘0 × 145’. After executing ADD, the top 2 elements in the stack are added and their taints are merged and the result is pushed into the stack. SWAP and POP just change the position of existing taints. After executing DUP2, the second element and its taint are duplicated. After executing SSTORE, the taints propagate to storage. During SSTORE execution, the taint sink will check the dependency of SSTORE, in this case, SSTORE depends on SLOAD whose pc is ‘0 × 145’ and ‘0×...’.

CFG construction. The CFG is built based on the EVM bytecodes (instructions) compiled from the source code. The detail steps in CFG construction are shown in Algorithm 1. The instruction sequence will be segmented by jump instructions to form CFG blocks (Line 2). CFG blocks are firstly connected by adjacent pc values, forming static edges (Line 3). During the symbolic execution, the CFG blocks will be connected through the jump destination, forming dynamic edges (Line 11). A CFG example is shown in Fig. 9. When constructing the CFG based on DFS, the instructions on each path in the CFG are executed

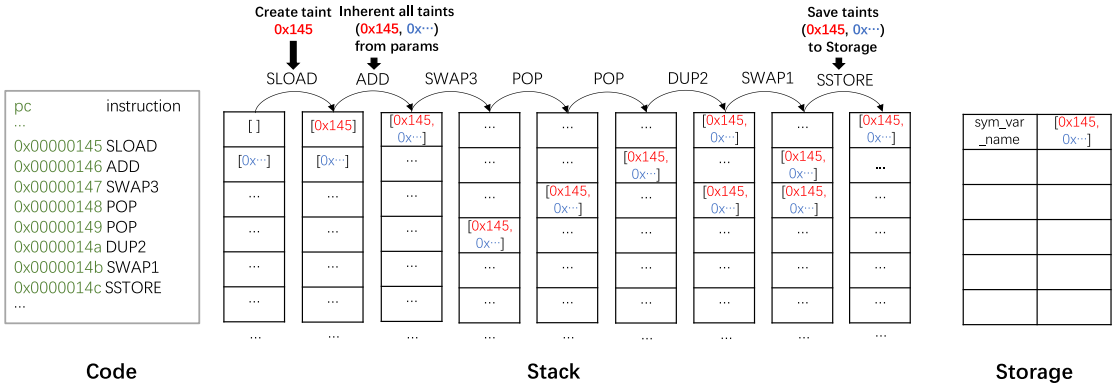


Fig. 8. An example of taint creation, propagation during the symbolic execution.

Algorithm 1 CFG construction algorithm

Input: Contract instructions *inss*
Output: Reentrancy pattern *reen_pattern*

```
1: // build CFG blocks and static edges
2: cfg_blocks = buildBasicBlocks(inss)
3: cfg = constructStaticEdges(cfg_blocks)
4: stack, mem, storage = init()
5: for path in cfg do
6:    RW, RC, RL, RCR, RCW = {}
7:    for ins in path do
8:       params, taints = stack.pops(ins)
9:       // build CFG dynamic edges
10:      if ins == JUMPI or ins == JUMP then
11:         constructDynamicEdges(ins, cfg, cfg_blocks)
12:         if ins == JUMPI then
13:            appendConditions(params, path.conditions)
14:         end if
15:      end if
16:      // collect dependency and check reentrancy pattern
17:      new_taints = createAndPropagateTaint(ins, taints)
18:      result = calculate(params)
19:      checkCalleePattern(ins, path, RW, RC, RL)
20:      checkCallerPattern(ins, path, RCR, RCW)
21:      update(stack, mem, storage, result, taints)
22:   end for
23:   caller_pattern = [RCW, RCR]
24:   callee_pattern = [RC, RW, RL]
25:   reen_pattern[path.id] = [caller_pattern, callee_pattern]
26: end for
27: return reen_pattern
```

Algorithm 2 Attack path check algorithm

Input: Reentrancy pattern *reen_pattern*
Output: Reentrancy appearance *has_reen*, reentrancy functions *reen_funcs*, reentrancy instructions *reen_inss*

```
1: // Filter duplicate paths
2: caller_paths, callee_paths = filterPaths(reen_pattern)
3: has_reen = false; reen_func = ''
4: for path in caller_paths do
5:    reen_list = null
6:    func = getFunctionSignature(path)
7:    // Skip caller paths in the same caller function
8:    if func == reen_func then
9:       continue
10:   end if
11:   for caller_pattern in path do
12:      // Concatenate caller and callee paths
13:      if caller_pattern is "RCW" then
14:         reen_list = connectPathRCW(caller_pattern, callee_paths)
15:      else
16:         reen_list = connectPathRCR(caller_pattern, callee_paths)
17:      end if
18:      // Skip caller patterns in the same caller path
19:      if reen_list is not null then
20:         has_reen = true; reen_func = func
21:         recordReenLocations(reen_list, reen_funcs, reen_inss)
22:         break
23:      end if
24:   end for
25: end for
26: return has_reen, reen_funcs, reen_inss
```

symbolically. For each instruction, firstly, the parameters are popped from the stack (Line 8); secondly, the taints of parameters are created and propagated (Line 17); thirdly, calculations are made based on parameters and instructions (Line 18); fourthly, if the instruction is related to reentrancy, the reentrancy pattern will be checked and record (Line 19, 20); finally, the computed results and taints are updated in stack, memory and storage (Line 21).

For instructions such as SSTORE/CALL/LOG that will affect the state of the blockchain, these instructions' dependency must be checked for reentrancy. Specifically, if the taints are not empty, the callee pattern RW/RC/RL will be recorded. For caller pattern check, we describe RCW pattern as example: When SSTORE is executed, every CALL before the SSTORE instruction on the current path is checked, and when there is a call-dependent SLOAD, whose address is the same as the address of SSTORE, we think that this SSTORE will form an RCW pattern with the SLOAD and CALL before. The key information of the pattern will also be saved, such as pc of instructions, address of the storage and path conditions of CALL, etc.

Attack path check. According to the reentrancy attack path pattern, we reorganize and concatenate the path to form an 'attack' path and use the constraint solver to solve whether the condition constraints of the 'attack' path are satisfied and whether the path is connective. If it is connected, there is reentrancy, and related functions and instructions are recorded, which are utilized to locate the source code through Abstract Syntax Tree (AST).

Attack path check is divided into two steps. Firstly, reen_pattern is filtered by only keeping one path for different paths with the same precondition before CALL and reentrancy pattern. Secondly, the caller and callee paths are concatenated to check connectivity and dependency consistency. The splicing algorithm is as shown in Algorithm 2, we traverse the filtered caller paths, and combine each caller pattern with the callee pattern on each callee path in turn. In the traversing process, we set two skip conditions to reduce unnecessary loops: Whenever there is a caller pattern on the caller path to form a connective reentrancy path, (1) stop continuing checking for other caller patterns on the current caller path, shown in Line 18, and (2) stop continuing checking for other caller paths in the same caller function, shown in Line 7. Goes

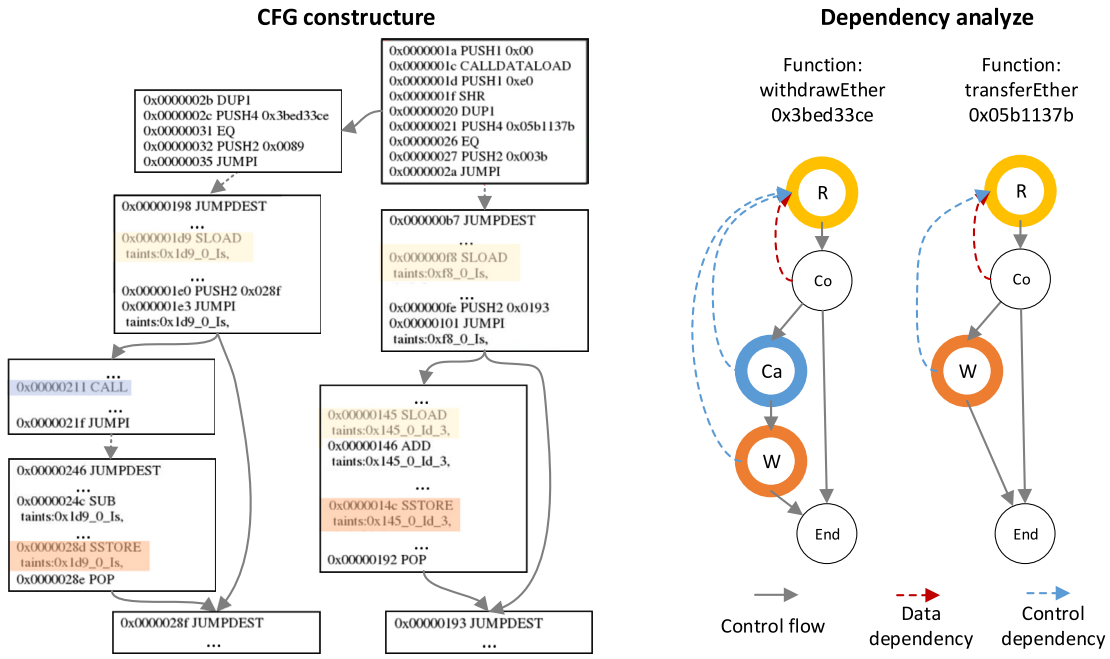


Fig. 9. CFG constructure and dependency analysis. The CFG based on code in Fig. 3 is shown on the left, where part of the basic block is omitted from the dotted line. Instructions such as SLOAD, SSTORE are annotated with associated taints, which are named with the instruction address and the symbolic variable name. The dependency graph abstracted from CFG is shown on the right, where R represents read, W represents write, Co represents conditional branch, Ca represents call.

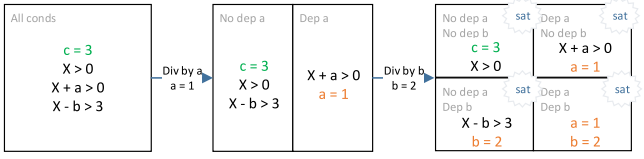


Fig. 10. Constraints division. The precondition is green (c = 3), path constraints are black, write operation sequence is orange (a = 1, b = 2). Here, the whole problem is divided into 4 sub-problems, based on whether the constraints depend on a and b. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

directly to the caller path and caller pattern checks of the next function. These rules also apply to callee patterns.

When solving connectivity, the path explosion and complicated condition constraints are two difficult points. To mitigate the path explosion, we filter the duplicated paths with the same condition constraints and eliminate unnecessary loops . To solve complicated constraints, instead of solving all constraint conditions in one problem, we separate the conditions into three sub-paths to solving: (1) from the beginning to the external call in the caller path, (2) the callee path, (3) after the external call in the caller path. The solution of each sub-path is regarded as the precondition for the solution of the next sub-path, and the reentrancy can only be achieved when all three sub-paths have solutions. To solve the sub-path solution, whose input includes preconditions, path constraints and write operation sequence, the path constraints and preconditions are divided into sub-problems for Z3, according to the write operation sequence. When each sub-problem has a solution, the solution for this segment of the path is considered successful. The division process is shown in Fig. 10.

4.3. Semantic equivalent patch

To check whether the behavior of a patched contract is consistent with the original one, we construct the semantics of the contract based on the contract execution process, which concretizes into CFG built in Section 4.2. We check the semantic equivalence based on three conditions: Input unchanged, Connectivity unchanged, and no new valid path.

The contract semantics are constituted by all possible execution processes. Each execution process of a smart contract is based on a context, executing a sequence of instructions, generating blockchain state updating, interacting with other contracts, ending with successful execution behaviors, or ending with rolling back transactions with a failed execution behavior. The same contract generates different behaviors based on different contexts, and the set of behaviors caused by all possible execution processes constitutes the contract's semantics. Concretely, We mapped execution processes in the contract to the paths in the CFG we construct in Section 4.2. The blockchain state updating behavior maps to the behavior of state update instructions on the path. The interactions map to the behavior of call-related instructions. The success/roll back behavior map to the behavior of end instructions on the path.

To check semantic equivalence, we compare the semantic difference between the paths before and after repair. After repair, there are two kinds of paths: modified paths P_{mod} and new paths P_{new}. The modified path P_{mod} has a corresponding path P_{org} in the original contract, which means most of the instructions in P_{org} are still in P_{mod}. The difference between P_{mod} and P_{org} is the order of the instructions and some additional instructions. The new paths P_{new} do not have corresponding paths in the original contract. P_{new} is the path introduced by the new conditional branch. Fig. 11(a) and (c) demonstrate P_{mod} in yellow solid line and P_{new} in green solid line. To check semantics for P_{mod}, the goal is to check whether the state updation is consistently, which means the input of the state update instruction is unchanged and the connectivity of the path leading to the state update instruction is unchanged. For P_{new}, no state update should be valid, which also means the path ends with a revert instruction.

To check whether the input of state-update instructions and connectivity of the path is unchanged in P_{mod}, we only need to check whether the storage variables value, which are used by state-update instructions and conditional instruction, is unchanged because the context and starting state are the same. However, it is difficult to acquire concrete values of storage variables under all possible concrete contexts and states. Thus, we use the def-use relationship of storage variables in the

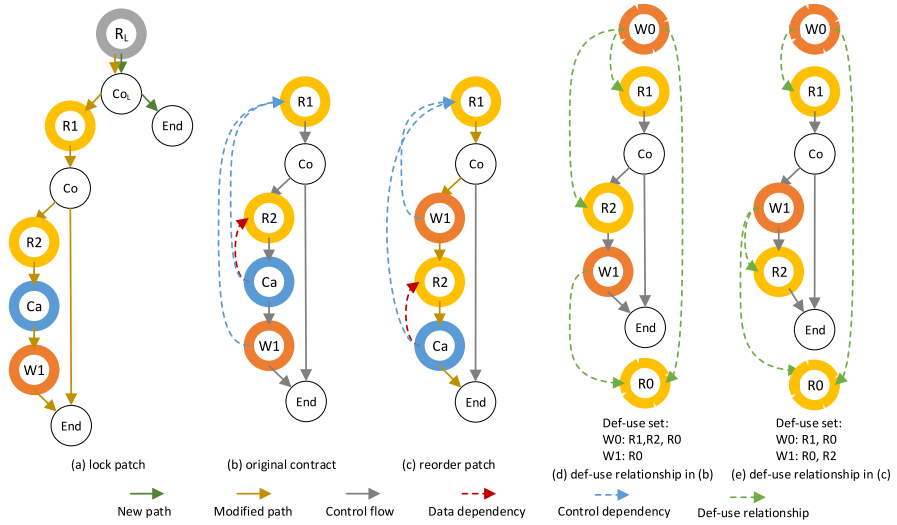


Fig. 11. The dependency graph abstracted from CFG for semantic analysis, where R represents read node in yellow (original existed) and grey (new added), W represents write node in orange, Co represents conditional branch, Ca represents call node in blue. New paths, modified paths, and control flows are represented by green solid lines, yellow solid lines, and grey solid lines, respectively. Data dependency, Control dependency, and def-use relationship are represented by red dashed lines, blue dashed lines, and green dashed lines, respectively. (b) shows the full version dependency in Fig. 9. (a) shows the lock patch version of (b) and the gray R node represents the reading lock storage. (c) shows the reorder patch version of (b). Compared to (b), W node is moved before R node in (c). (d) shows the def-use relationship of storage in (b). (e) shows the def-use relationship of storage in (c). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

CFG to simulate possible situations. If the def-use relationship of the storage is unchanged, then the value of storage will also be unchanged. Fig. 11(d) and (e) demonstrate the def-use relationship in the green dotted line.

As summarized, we say that a contract and its patched version are semantic equivalent, when based on the same starting state and same context parameters, the following **three conditions** are satisfied:

1. **Input unchanged.** The value of input for state-update instruction for P_{mod} and P_{org} is the same, which is checked by whether the def-use relationship of the storage dependent by the state-update instructions is unchanged.

2. **Connectivity unchanged.** The connectivity of the path leading to state-update instruction for P_{mod} and P_{org} is the same, which is checked by whether the def-use relationship of the storage dependent by the conditional instructions is unchanged.

3. **No new valid path.** The new path P_{new} does not introduce effects and the path ends with revert, which is checked by whether the end instruction is REVERT.

Algorithm 3 Semantic equivalence check algorithm

Input: Original contract *org*, Patched operations *patch*
Output: Check result *eq*, Unequal reason *err*

```
1: // Check Conditions 1: Input unchanged
2: update_ins = findUpdateIns(org)
3: storage1 = extractDependStorage(update_ins)
4: if !isSameDefUse(storage1, org, patch) then
5:     return FALSE, 'Input'
6: end if
7: // Check Conditions 2: Connectivity unchanged
8: cond_ins = findConditionIns(org)
9: storage2 = extractDependStorage(cond_ins)
10: if !isSameDefUse(storage2, org, patch) then
11:     return FALSE, 'Connectivity'
12: end if
13: // Check Conditions 3: No new valid path
14: new_branch = findNewBranch(org, patch)
15: if !isRevert(new_branch) then
16:     return FALSE, 'Not Revert'
17: end if
18: return TRUE
```

The semantic equivalence check algorithm is shown in Algorithm 3. Firstly, we check **Input unchanged** in Line 2–6. We find the state-update instructions and extract the storage they depend on based on the taints constructed during CFG construction, as shown in Fig. 11(b) and (c). Then we check whether the def-use relationship of the storage is unchanged. Secondly, we check **Connectivity unchanged** in Line 8–12. we find the conditional branch instructions and the storage they depend on. Then we check whether the def-use relationship of the storage is unchanged. Thirdly, we check **No new valid path** in Line 14–17. we find the new branches introduced by the patch and check whether the paths, where the new branch is located, end with a REVERT instruction.

The details of the def-use relationship check are as follows: For the storage variables both in the original and patched contract, we first collect the def-use set of the original contract by finding the path starting from one write node and ending with a read node, without other write nodes on this path. As Fig. 11(d) shows, the def set includes $W0$ and $W1$, and the use set for $W0$ includes $R1$, $R2$ and $R0$. Notice that we add a write node $W0$ before the graph and a read node $R0$ after the graph to track the state update after the execution process finishes. After the patch, we check whether the existing def-use relationship is still satisfied and whether the new def-use relationship is built. For the storage only appearing in the patched contract, because of no original def-use relationship to compare, we check its effects as follows: When it does not appear as an input for state-update instructions, we consider its def-use relationship is unchanged. When it appears in the conditional expression, for the existing path branch, the value of the new storage value can make the connectivity not change, which is checked by the constraint solver, we consider its def-use relationship to be unchanged.

The approximation in the algorithm. We build the semantics check algorithm based on the CFG constructed by the symbolic execution. The semantics of the contract are approximate to the state-update behaviors in all the paths in the CFG. To check semantic equivalence under a repair scenario, we check the state-update behaviors by analyzing the def-use relationship of the storage dependent by the state-update instruction.

The precision/guarantees of the algorithm will be affected by 2 factors as follows:

1. The coverage of CFG for the contract. The coverage of CFG determines the completeness of contract semantics. The CFG generated by the symbolic execution sometimes may face the path explosion problem. If the number of iterations exceeds the bound, the subgraph of the node will not recurse further for performance reasons, which may miss some semantics.

2. The approximation of the state change with the def-use relationship of storage change. The premise for considering def-use relationship of storage as a measure is that the dependency between the storage variable and the state-update instruction is unchanged. Based on the patch operations used in this paper, we do not change the expression with storage variables or just replace the storage with the same value local variables in the existing statements. For more complex repair operations, such as changing the expression of the condition, it is necessary to add the judgment of expression equivalence before the def-use relationship check.

4.4. Gas-optimized templates

Existing reentrancy patch templates in the source code level have a large gas overhead. Therefore, we design the reordering template and optimize the lock template to reduce the gas overhead. Both templates aim to restore or protect the atomicity of functions. We optimize gas usage based on the SSTORE instruction from three perspectives: first, do not introduce additional SSTORE instructions; second, reduce the usage of SSTORE instruction; third, reduce the consumption of gas when the SSTORE instruction has to be used.

Lock solution. The lock solution is an effective way to defend against reentrancy, but it still faces two challenges in automated repair scenarios: high gas overhead and deadlock, leading to semantic inequivalence.

To reduce the usage of SSTORE instruction. We break the original lock into the $check$ part and the $acquire$ part. The $check$ part does not contain SSTORE, which is added to the callee function to avoid re-entering. The $acquire$ has SSTORE to avoid the reentrancy starting from the caller. Compared to the old implementation, this optimization works well with Cross-type reentrancy because it uses less gas when invoking callee functions. As shown in Fig. 12(a), $check\_lock$ and $acquire\_release\_lock$ implement $check$ and $acquire$ part in our template. To reduce gas consumption, when the SSTORE instruction has to be used, SSTORE needs to modify a value that is not 0. As shown in Fig. 12(a), $lock$ is set non-zero in the first line and will be initialized in constructor, increasing the deployed gas cost. However, the deployment is a one-time execution, and function invocations will execute multiple times, so the overall gas is optimized as the number of function invocations grows.

We assign a lock ID for each caller to track the relationship between the caller function and the callee function. Each caller function has a unique ID. All callee functions that can be reentered by this caller function use the same ID as the caller. All the locks' statuses are mapped in $lock$ variable in a bit. Because the $lock$'s type is uint256, and at least the highest bit must be 1, the max number of callers is 255, which is far enough to prevent reentrancy.

The lock template is generated as a pre-defined template, including lock variable definition, acquire_release_lock modifier definition, check_lock modifier definition and modifiers application statements. When applying the lock template, the definition will be added at the top of the contract. Based on the $reen\_funcs$ recorded by Algorithm 2, modifiers application statements with corresponding lock ID will be added to the reentrancy function.

After adding locks, new storage variable $locks$ and a new condition for checking locks are introduced. To ensure semantic equivalence, there must be an assignment for locks to make the original behavior unchanged, avoiding deadlock. In the implementation, after adding candidate locks, we use Algorithm 3 to find semantic change caused by 'Connectivity'. Then remove redundant locks in the nested-function scenario.

Reordering solution The reordering solution fixes reentrancy by reordering atomicity-violation operations, such as moving writing statements or reading statements before the message call: in RRW type reentrancy, reordering the writing statement; in RWR type reentrancy, reordering the reading statement. In the reordering solution, generally, we do not introduce new SSTORE instructions so that the gas overhead is nearly equal to the original.

In the RRW repair template, we move the writing statement related to reentrancy just before the message call statement. Then we check if such movements will change the def-use relationship in the original function through Algorithm 3. If it contains semantic changes, we need to determine whether they can be corrected by the maintenance operation defined in our template. We denote the area between the destination and the source of the movement $affectedArea$, as shown in Fig. 6 yellow part. Because the template is known to move a writing statement, the semantic change means that the def-use relationship for the moved storage variable in $affectArea$ is changed. It also indicates that the $affectedArea$ contains read or write operations on the same storage variable. We maintain semantics in two scenarios:

- **Read Only.** There exist only the reading statements in $affectArea$. we use a local variable to store the storage variable value before calling statement to retrieve the original def use relationship. When the read and write statements are at the same AST level, the read statement in $affectedArea$ will be replaced by the local variable, as shown in Fig. 12(b) withdrawToken function. When the statements are not at the same level, there might be if-branches in the same function or the statements are not in the same function. If they are in the same function, the affected storage in $affectArea$ will be assigned by the local variable which keeps the original state, as shown in Fig. 12(b) Collect() function. If they are not, it is hard to fix by move statement, we will leave it for the future.
- **Write exists.** There exist the writing statements in $affectArea$. It is difficult to maintain through simple statements because of the more complex def-use relationships involved, so it is not dealt with in this paper.

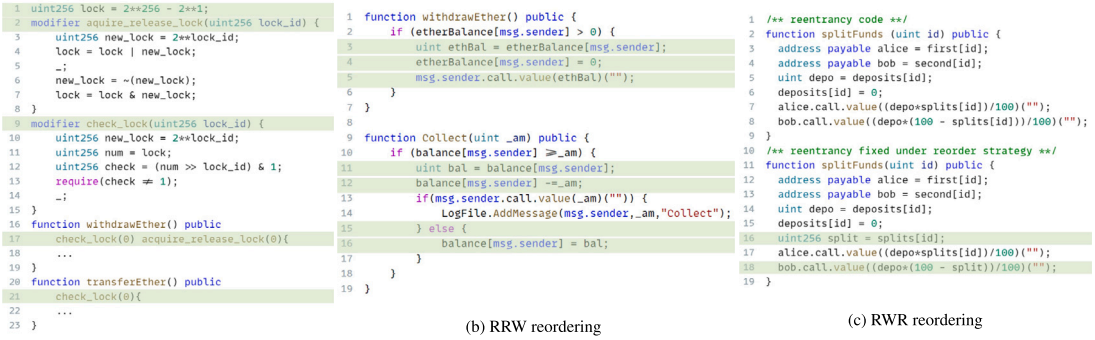
In the RWR repair template, we replace the storage variable in the second read statement with a local variable, whose initialization is the same value as the replaced storage variable before the call statement, as shown in Fig. 12(c). The semantic equivalence checking and maintenance is similar to RRW.

The reorder template defines the move operation for the writing statement (start position, destination position), insertion operation for semantic maintenance (new local variable definition), and replace operation for storage variable replacement (replaced storage variable, new local variable), the reorder template application is based on the statements corresponding to *reen_inss* record by Algorithm 2. The location of statements is achieved by mapping from instructions recorded by Algorithm 2 to the AST of source code. The mapping can be obtained during compilation.

5. Evaluation

Along with the motivation and the main stated objective mentioned in Section 3.2 we evaluate ReenRepair based on both real-world contracts and typical cases. Our main research questions are listed as follows:

- RQ1: Is ReenRepair precise in locating reentrancy in smart contracts?
- RQ2: Is ReenRepair effective in repairing reentrancy in smart contracts without changing the original semantics?



```
uint256 lock = 2**256 - 2**1;
modifier aquire_release_lock(uint256 lock_id) {
    uint256 new_lock = 2**lock_id;
    lock = lock | new_lock;
    _;
    new_lock = ~(new_lock);
    lock = lock & new_lock;
}
modifier check_lock(uint256 lock_id) {
    uint256 new_lock = 2**lock_id;
    uint256 num = lock;
    uint256 check = (num >> lock_id) & 1;
    require(check == 1);
    _;
}
function withdrawEther() public
    check_lock(0) acquire_release_lock(0){
    ...
}
function transferEther() public
    check_lock(0){
    ...
}
```

(a) RRW lock

```
function withdrawEther() public {
    if (etherBalance[msg.sender] > 0) {
        uint ethBal = etherBalance[msg.sender];
        etherBalance[msg.sender] = 0;
        msg.sender.call.value(ethBal)("");
    }
}

function Collect(uint _am) public {
    if (balance[msg.sender] >=_am) {
        uint bal = balance[msg.sender];
        balance[msg.sender] -=_am;
        if(msg.sender.call.value(_am)("")) {
            LogFile.AddMessage(msg.sender,_am,"Collect");
        } else {
            balance[msg.sender] = bal;
        }
    }
}
```

(b) RRW reordering

```
/** reentrancy code **/
function splitFunds (uint id) public {
  address payable alice = first[id];
  address payable bob = second[id];
  uint depo = deposits[id];
  deposits[id] = 0;
  alice.call.value((depo*splits[id])/100)("");
  bob.call.value((depo*(100 - splits[id]))/100)("");
}
/** reentrancy fixed under reorder strategy **/
function splitFunds(uint id) public {
  address payable alice = first[id];
  address payable bob = second[id];
  uint depo = deposits[id];
  deposits[id] = 0;
  uint256 split = splits[id];
  alice.call.value((depo*splits[id])/100)("");
  bob.call.value((depo*(100 - split))/100)("");
}
```

(c) RWR reordering

Fig. 12. Repair templates.

Table 3
Results of reentrancy location. TP: Number of true positive functions; FP: Number of false positive functions. Precision = TP/(TP + FP); The up arrow ↑ means that higher is better for this metric, while the down arrow ↓ does the opposite; Abort: Timeout or compiler version incompatible; Fail rate: Number of Abort/Number of contracts in dataset 1 and 2; P/F = Precision/Fail rate.

Tool	Real world (dataset 2)			Synthetic (dataset 1)			Total					
	Function-level			Function-level			Contract-level		Function-level			P/F↑
	TP	FP	Precision↑	TP	FP	Precision↑	Abort	Fail rate↓	TP	FP	Precision↑	
Securify	34	29	53.97%	3	0	100.00%	6	13.04%	37	29	56.06%	4.30
Clairvoyance	24	1	96.00%	2	0	100.00%	18	39.13%	26	1	96.30%	2.46
sGuard	7	16	30.43%	3	0	100.00%	6	13.04%	10	16	38.46%	2.95
ReenRepair	45	5	90.00%	11	0	100.00%	9	19.57%	56	5	91.80%	**4.69**

- RQ3: Is ReenRepair effective in optimizing gas overhead compared to other tools?

We also compare ReenRepair to Securify2.0,[1] Clairvoyance,[2] SCRepair and sGuard,[3] two reentrancy detection tools and two reentrancy repair tools. All experiments were performed atop a machine having 2 Intel(R) Core(TM) i9-10920X CPUs at 3.50 GHz with 12 cores each and 2 threads/core, and 128 GB of RAM, running 64-bit Ubuntu v18.04. We ran our test cases on Ganache (v6.12.2) under the Brownie(v1.19.3)[4] framework and used Z3 v4.8.9 as our constraint solver.

5.1. Datasets

We collect data based on two criteria: (1) Covers the three types of reentrancy mentioned in this article (RRW-Self, RRW-Cross and RWR-cross); (2) Use real-world cases whenever possible. We used three datasets to evaluate ReenRepair. For the first criteria, the first dataset consists of four artificially constructed reentrancy contracts, including all three types of reentrancy, to prove the effectiveness of our method. We use synthetic data because some type of reentrancy is rare and lacks real-world data. For the RWR-Cross reentrancy, we construct two contracts, one including control dependency, and the other including data dependency. For the second criteria, the second and third datasets consist of the contracts from Ethereum mainnet, to prove the scalability of our method in repairing real-world smart contracts. The second dataset consists of 42 contracts collected from the manual analysis dataset of existing works (Yu et al., 2020; Chen et al., 2020; Wu et al., 2020), whose address or transaction address had been mentioned in paper explicitly, with solidity source code available on etherscan.io.[5] The third dataset includes the top 50 activated ERC20 contracts on the Ethereum main net. More specifically, our datasets contain 64 reentrancy functions in the first and second datasets, including both the caller and callee functions.

[1] https://github.com/eth-sri/securify2.
[2] http://47.100.164.141:8080/code.
[3] https://github.com/duytai/sGuard.
[4] https://github.com/eth-brownie/brownie.
[5] https://etherscan.io/.

5.2. The precision of location (RQ1)

In this task, we evaluated ReenRepair's ability to locate reentrancy based on all three datasets. We also compared our work with state-of-art detection tools, including Securify2.0, Clairvoyance and sGuard. Although the source code of SCRepair is publicly available, we did not manage to compile it.

ReenRepair has high location precision, as Table 3 demonstrates location results based on the first and the second datasets. Here we do not include the third dataset localization results in the overall statistical range, because all four tools have more than 25% abort rate for the top 50 tokens, and Clairvoyance even has 82% of examples abort.

We evaluate the precision of the location from the function level, and the results are presented in Table 3 according to Total (real-world and synthetic), Real-world-only and Synthetic-only. For the synthetic data, all tools' precision reach 100%, however, ReenRepair obtains the highest TP due to its capability of detecting cross-type reentrancy. For the total and real-world-only, compared to existing tools, ReenRepair's precision reaches (91.80% and 90.00%), which is far better than Securify (56.06% and 53.97%) and sGuard (38.46% and 30.43%) and is second only to Clairvoyance. However, Clairvoyance has almost twice the failure rate of ReenRepair, which affects the precision of Clairvoyance. The absolute number of Clairvoyance's FP is less, and for the 5 FPs produced by ReenRepair, Clairvoyance's location results are aborted. Besides, Clairvoyance missed 33 reentrancy because of abort, while ReenRepair missed only 7. To comprehensively consider the tool's usability and precision, this paper calculates the ratio of precision and fail rate, denoted as "P/F", and ReenRepair has a score of 4.69, indicating a good balance of usability and precision. In ReenRepair, there are still 5 bug-free functions located with reentrancy, because of errors in critical storage variable location. Noticed that the reason for not choosing the contract level is that some tools, such as Securify and sGuard, include both functions with real reentrancy and false positives in the locality results of the same contract.

The results of the reentrancy type are counted in units of caller-callee pairs and are demonstrated in Table 4. RRW-Self reentrancy is the most common type of reentrancy, which accounts for 68.18% (30 cases) in the real-world dataset. RRW-Cross and RWR-Cross account for

Table 4
Results of reentrancy types located by ReenRepair. N/A: Not applicable. The tool is not capable for this type of reentrancy. RRW-Self: storage variable follows read–read–write pattern. The caller and callee function is same. RRW-Cross: storage variable follows read–read–write pattern. The caller and callee function is different. RWR-Cross: storage variable follows read–write–read pattern. The caller and callee function is different.

	Real world (dataset 2)			Synthetic (dataset 1)			Total		
	RRW-Self	RRW-Cross	RWR-Cross	RRW-Self	RRW-Cross	RWR-Cross	RRW-Self	RRW-Cross	RWR-Cross
Securify	34	N/A	N/A	3	N/A	N/A	37	N/A	N/A
Clairvoyance	24	0	N/A	2	0	N/A	26	0	N/A
sGuard	7	N/A	N/A	3	N/A	N/A	10	N/A	N/A
ReenRepair	**30**	**13**	**1**	**1**	**4**	**3**	**31**	**17**	**4**

29.55% (13 cases) and 2.27% (1 case), respectively. Apparently, cross-type reentrancy is rare in real-world smart contracts, which is harder for attackers to exploit. Although Clairvoyance claims to have the ability to detect cross-function reentrancy, the result of Clairvoyance contains only the type of RRW-Self, and the number of RRW-Cross is zero. Securify and sGuard are not applicable for RRW-Cross and RWR-Cross. Securify detects more RRW-Self than ReenRepair is due to less abort cases, but Securify has far more false positive than ReenRpair.

5.3. The effectiveness and semantical equivalence of the patch (RQ2)

In this task, based on the first and second datasets, we evaluate two repair templates used in ReenRepair. We also compared our work with the sGuard and SCRepair. Due to compilation constraints issues with SCRepair, we only compared the 4 contracts mentioned in their paper (Yu et al., 2020).

To assess the semantical equivalence of the patch, we constructed test cases manually, because some contracts have less than 10 historical transactions, making it challenging to cover semantic checks and the fuzzing tool has the limitation of narrow-check. We wrote test cases to test the functions affected by the repair, including the modified function and the function calling the modified function. The test cases check whether the behaviors of these functions are consistent before and after being repaired. We evaluate the self-built test cases by source code coverage with the help of Brownie, a Python-based development and testing framework for smart contracts. The average coverage achieved 95%. To assess the effectiveness of the patch, which means to check whether the reentrancy is eliminated, we use ReenRepair to detect the repaired contract again. For the contracts that are still reported to have reentrancy, we conduct a manual review to remove false positives. The detailed manual check process is shown as follows. We invite two volunteers who have 4 years of Solidity development experience to verify whether every reported reentrancy is false positive. The verification refers to the reentrancy pattern defined in Table 2. If both volunteers agree that the reentrancy case reported by the detection tool is correct (or incorrect), the case is considered true positive (or false positive). If the two volunteers disagree, the two parties finalize the result through discussion.

Both of the ReenRepair templates have a better repair rate than the state-of-the-art tools. Table 5 summarizes the results. The bit-lock template has the highest repair rate 80.95%, followed by reorder template at 76.19%, outperforming compared to SCRepair (75.00%) and sGuard (16.67%). The results illustrate our repair method based on the template is more targeted to reentrancy than the random generation strategies used in SCRepair and free from semantic change problems encountered in sGuard, whose example showed in Fig. 5

Both repair templates lead to some semantic changes after repair and most of the changes are maintained successfully. Fig. 13 shows the semantic check and maintenance result. There are 2 semantic changes in the bit-lock scenario out of a total of 62 modifications and 29 semantic changes in reorder scenario out of 35 modifications, accounting for 83%. The results illustrate that the statement editions have a high probability of affecting the semantics and the importance of semantic checking. After applying maintenance operation, most of the semantic changes were corrected. All semantic changes caused by

Table 5
Repair result. Scs: Number of contracts repaired successfully; Fail: Number of contracts failed repaired; F-loc: Failed by location error; F-sem: Failed by semantic not equivalence.

Tools	Scs	Fail	F-loc	F-sem	Scs rate
SCRepair	3	1	–	–	75.00%
sGuard	7	35	33	2	16.67%
Reen:bit-lock	34	8	8	0	80.95%
Reen:reorder	32	10	8	2	76.19%

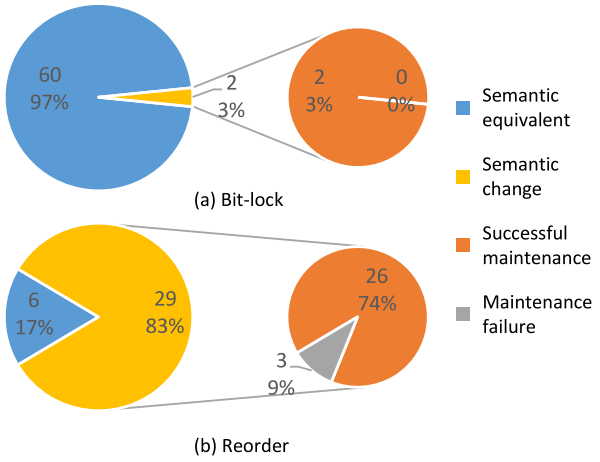


Fig. 13. Semantic check and maintenance result.

bit-lock and 89.66% of semantic changes caused by reordering were successfully maintained, indicating the effectiveness of ReenRepair in repairing reentrancy and keeping semantic equivalence at the same time.

The reason for failure in reorder template is the error storage variables' locations leading to moving wrong statements. Because of the mechanism, multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible, in Solidity. It is challenging to tell the difference between the variables in such a compacted slot. As shown in Fig. 14. The write statement changes *_user.r_block*, a member variable of struct in Line 14. The type of *_user.r_block* is uint32, which will be compacted in one slot with *_user.r_id* (uint32), *_user.r_number* (uint8) and *_user.r_payout* (uint32), because of the mechanism that multiple, contiguous items that need less than 32 bytes are packed into a single storage slot if possible in Solidity.

5.4. The cost of the patch (RQ3)

Gas overhead is inevitable. In this task, we evaluate the gas usage of the patch from two perspectives: the total gas overhead of contracts and each patch's gas overhead. We calculate the gas consumption by running the test cases we used in Section 5.3.

Total Gas consumption. The reorder template introduces the least gas overhead in both deployment and execution phrases, as shown in Table 6 and Fig. 15. During the contract deployment phase, the ReenRepair reorder template introduces the least gas overhead 1069, which

```
// reorder fail: write statement locations
//InstaDice_0xfe1b613f17f984e27239b0b2dccfb1778888dfae
function payoutPreviousRoll()
    public lock_check(0) lock_aquire_release(0)
    returns (bool _success)
{
    User storage _user = users[msg.sender];
    if (_user.r_block == uint32(block.number)) {...}
    if (_user.r_block == 0) {...}
    Stats storage _stats = stats;
    // _finalizePreviousRoll() contains call.value
    _finalizePreviousRoll(_user, _stats);
    _user.r_id = 0;
    _user.r_block = 0;
    _user.r_number = 0;
    _user.r_payout = 0;
    stats.totalWon = _stats.totalWon;
}
```

Fig. 14. Reordering Strategy failed by locating write statements.

Table 6
Results of gas overhead.

Template	Deployment	Execution
sGuard	113 695	6278.208
Reen:bit-lock	45 343.57	2693.967
Reen:bit-lock:check	–	898
Reen:bit-lock:acquire	–	3511.5
Reen:bit-lock:check+acquire	–	4183.25
Reen:reorder	1069	5.78

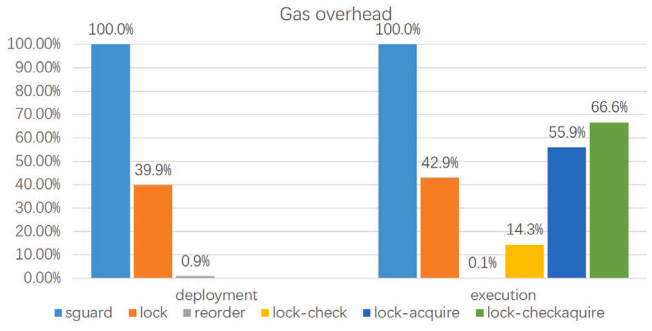


Fig. 15. Results of gas overhead.

is only 0.9% of sGuard. ReenRepair bit-lock and sGuard template introduce more gas overhead because of the lock storage variable definition. However, the ReenRepair template uses only 39.9% gas overhead of sGuard. During the contract execution phase, the ReenRepair reorder template still has the least gas overhead 5.78, which is 0.1% of sGuard. The ReenRepair bit-lock template uses 42.9% gas overhead of sGuard, benefiting from the optimization of storage variables.

One patch Gas consumption. During the experiment we found that gas overhead fluctuates from contract to contract because of the difference in the number and the part of the patch used. Thus, to reflect the effectiveness of gas optimization in this paper, we also compared one-patch-cost between different templates.

As Fig. 15 shows, to repair one reentrancy, a reordering patch consumes only 0.1% gas of the nonReentrant patch used by sGuard. To repair one self-type reentrancy, ReenRepair uses one check and one acquire part together in the bit-lock template, and the gas overhead is 66.6% of sGuard. To repair one cross-type reentrancy, ReenRepair use a check and an acquire part separately, but sGuard need two nonReentrants which leads to higher cost. In summary, compared with sGuard, the additional gas overhead introduced by ReenRepair is smaller, and the gas optimization strategy is effective.

6. Related work

APR for concurrent bugs. Considering the similarity of reentrancy and concurrent faults, we discuss whether the existing APR approaches can be applied to rectify reentrancy contracts. Based on the patch generation, approaches for concurrent faults are classified into two types: search-based and template-based.

A representative search-based work is ARC (Kelk et al., 2013). ARC uses a genetic algorithm without crossover to mutate an incorrect program, searching for a variant of the original program that fixes the deadlocks and data races. In detail, ARC generates subsequent generations by applying mutation operators on the original buggy program randomly, such as adding synchronized blocks, expanding synchronized regions, removing synchronization and shrinking synchronization blocks. The mutated individual is maintained only if the fitness function is improved.

The common templates for concurrent faults are the lock template and the order template. AFix (Jin et al., 2011) is used to repair atomic problems by enclosing the critical region with mutex locks. CFix (Jin et al., 2012) evolved from AFix and can repair both mutual exclusion and ordering problems. Besides the lock and unlock operations, they also add counter and broadcast variables into the template to implement synchronization. HFix (Liu et al., 2016) introduces move operation by rearranging the placement of memory-access statements and synchronization operations, which already exist in the same thread, instead of adding new synchronization.

For concurrent faults, there are more related works based on template-based than search-based, which is largely related to the need to specify critical regions and involved variables. search-based methods need to carry out multiple mutations in order to determine the region scope and verify test cases for each mutation, which costs a lot. This disadvantage will be amplified in the blockchain scenario.

Reentrancy detection. Reentrancy detection methods classify into static detection, dynamic detection and machine learning-based detection. The one-sided description of the pattern leads to false positives. Mythril (Mueller, 2018), Securify (Tsankov et al., 2018), Manticore (Mossberg et al., 2019) and Static ECFChecker (Albert et al., 2020) use "state change after message call" only; Oyente (Luu et al., 2016) and Zeus (Kalra et al., 2018) use the feasibility of call loop only. The two above detection strategies face the first and second false positive scenarios mentioned in Section 4.2 respectively. Clairvoyance (Xue et al., 2020) filters false positives through 5 PPTs (Path Protection Technique) and improve the precision. Compared with existing static detection works, our location method takes into account two false positive scenarios, and the detection precision is basically the same as Clairvoyance and has a lower abort rate. Our location information is also more detailed than static detection tools, including the caller and the callee function relationships, and error post-position statements.

The dynamic detection object is transaction instructions, thus there is no problem of false positives because key instructions cannot be executed, and generally speaking the dynamic detection accuracy is higher than the static detection. Sereum (Rodler et al., 2019) and TXSpector (Zhang et al., 2020a) detect whether the message call depends on the storage variable which has been modified after the message call. The ECFChecker (Grossman et al., 2017) checks whether the reentrancy execution and its sequential execution results are consistent, based on the security property ECF. Although dynamic detection can detect reentrancy with higher precision, it relies on pre-transaction execution. If the repair process is based on dynamic detection, besides the contract codes, additional transaction inputs are necessary.

Machine learning-based methods treat vulnerability detection as a classification task, and train the model based on existing benchmarks or manually collected data. TMP (Zhuang et al., 2021) and Bi-GGNN (Cai et al., 2023) are both graph-neural-network-based approaches, detecting reentrancy based on the features extracted from different forms of contract graphs. PSCVFinder (Yu et al., 2023), Peculiar (Wu et al.,

Table 7
Summary of the tool of repairing reentrancy.

Tool	Patch. level	Storg. mod	Gas	Sem check	Sem Maint
SCRepair	Statement	✗	low	✗	✗
sGuard	Function	✓	high	✗	✗
SmartShield	Statement	✗	low	✓	✗
Elysium	Function	✓	high	✗	✗
Aroc	Statement	✓	high	✗	✗
SmartRep	Statement	✗	low	✗	✗
RLRep	Statement	✗	low	✗	✗
ReenRepair	Both	✓	mid	✓	✓

2021) and ReVulDL (Zhang et al., 2022) are pre-training model based approaches. PSCVFinder utilizes prompt-tuning to fill the gap between pre-training tasks and downstream vulnerability. The latter two use the graph-based pre-training model to combine code sequence and data flow information. Machine-learning-based detection needs high-quality data for training. However, the labeled dataset used in the above methods, SmartBugs (Ferreira et al., 2020) and SolidiFI Benchmark (Ghaleb and Pattabiraman, 2020), do not include cross-type reentrancy data. Most methods can detect whether the contract is vulnerable. However, the limitation of the classification task also leads to lacking the location information, such as results on locating vulnerable statements.

Reentrancy repair. Reentrancy repair methods classify into search-based, template-based methods and machine learning-based methods. The search-based approach is represented by SCRepair (Yu et al., 2020), using a genetic algorithm based on search and mutation, randomly adding, moving and replacing statements in the source code to generate candidate patches. The patch with the lowest gas cost is selected as the repair solution among all candidate patches. The generative method has great randomness and is difficult to repair complex reentrancy, such as cross-function reentrancy, leading to a low repair rate.

Existing work that uses a template-based approach includes sGuard (Nguyen et al., 2021), SmartShield (Zhang et al., 2020b), Elysium (Ferreira Torres et al., 2022) and Aroc (Jin et al., 2021). sGuard and Elysium fix the reentrancy by applying a lock strategy for the reentrancy function at the source code level and bytecode level respectively. sGuard lacks semantic invariance detection for lock addition, resulting in nested deadlocks. The process of rewriting storage variables in the non-optimized lock template also leads to a high gas overhead.

SmartShield and Aroc use reordering strategies statically and dynamically, respectively. SmartShield repair at the bytecode level, which locates the SSTORE and CALL instructions that cause reentrancy by "State changes after Call", and preserves semantic information by extracting AST. SmartShield contains the semantic equivalence check, but it cannot handle invariance, which requires manual correction. Aroc is the only method to repair reentrancy dynamically by deploying a patch contract and relating the reentrancy contract on the blockchain through the modified EVM. Aroc's template uses the idea of reordering to move state modifications before the message call in the patch contract by modifying shadow storage variables.

Machine learning-based repair methods include SmartRep (Zhou et al., 2023) and RLRep (Guo et al., 2023). Both methods repair the reentrancy by replacing "call()" with "send()", avoiding reentrancy by limiting gas consumption. SmartRep uses an encoder–decoder model to translate vulnerable code to fixed code. RLRep is a reinforcement learning-based approach that uses the three-line vulnerable source code and AST to generate repair action sequences selected from the action space. The repair actions include transfer function replacement, special statement insertion, etc. The fixed contract is evaluated by a reward function based on compilation, vulnerability detection tools, code similarity, and code entropy. Both tools can repair multiple types of vulnerability, including reentrancy. However, mainly focusing on three line bugs makes it difficult for the tool to learn multi-line patches for reentrancy. Simply replacing "call()" with "send()" also ignores the fact that some contracts require a call to trigger an external call to perform normal logic.

Existing reentrancy repairing tools are summarized in Table 7. Most tools use function patches or statement patches only. The patch used storage variable modification leading to higher gas costs. The semantic equivalence check before test cases validation only appears in SmartShield and none of the tools did the semantic maintenance.

7. Discussion

There is still room for improvement. In the location phase, although the precision of ReenRepair is over 90%, it still exists false positives. The main reason is that the identification of storage slot is not accurate enough, because in implementation, it is hard for ReenRepair to locate the innermost member variable for nested data structure with more than two levels, such as mapping, dynamic array and struct. In the semantic equivalence check, ReenRepair uses symbolic execution to simulate traversing all possible execution contexts and their execution path in functions. However, due to the limitations of symbolic execution, it is possible that the traversed context is only a subset of all possibilities, and the context traversal approach needs to be optimized in future work.

In the repair phase, for the semantic definition, we use CFG to approximate the contract behavior. The semantic might be missed when the depth of paths over the upper bound. In the future, we will check the semantics base on more formal definition of EVM bytecode (Hildenbrandt et al., 2018) or solidity code (Bartoletti et al., 2019; Jiao et al., 2020). For the semantic maintenance, the reordering strategy faces the case that the locating statements are not on the same level. In our implementation, for locating statements in the same function definition block but not in the same level of statement block (call statement in the true branch block of if statement, but write statement lies after the if statement), it can be moved without changing the semantics. However, the case where the locating statements are not in the same function definition block (e.g. the call statement exists in the function definition but the writing statement exists in the modifier definition) cannot be reordered and can only be fixed using the lock strategy.

At present, the semantic equivalence check algorithm and gas-optimized repair templates have only been applied to the reentrancy vulnerability. In the future, it may be considered to migrate the above methods to other vulnerabilities, such as vulnerabilities that may use storage variables for access control.

8. Conclusion

In this article, we have presented an approach to repair reentrancy automatically in smart contracts with low gas overhead, which keeps semantic equivalence at the same time. Our framework, ReenRepair, locates the reentrancy based on the connectivity and read–write dependency with low false positives and repairs reentrancy based on gas-optimized templates armed with the semantic maintenance operation to keep semantic equivalence. The semantic equivalence check algorithm and semantic maintenance in our method are the first work in the smart contract repair field to our knowledge. Our experiments demonstrated that our method can handle real-world contracts with relatively low gas consumption.

CRediT authorship contribution statement

Ruiyao Huang: Methodology, Software, Validation, Writing – review & editing. **Qingni Shen:** Methodology, Writing – review & editing. **Yuchen Wang:** Methodology, Writing – review & editing. **Yiqi Wu:** Validation. **Zhonghai Wu:** Funding acquisition. **Xiapu Luo:** Methodology, Writing – review & editing. **Anbang Ruan:** Writing – review & editing.

Table A.8
Taint creation and propagation rules.

Type	Propagation rules	OPCODES
Arithmetic	Stack → stack	ADD, MUL, SUB, DIV, SDIV, MOD, SMOD, ADDMOD, MULMOD, EXP, SIGNEXTEND
Comparison		LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, NOT, BYTE, SHL, SHR, SAR
Duplication		DUP1, DUP2, . . . , DUP16
Swap		SWAP1, SWAP2, . . . , DUP16
Memory store	Stack → memory	MSTORE, MSTORE8
Memory load	Memory → stack	MLOAD
Sha3	Memory → stack	SHA3
Storage load	Create new taint; stack and storage → stack	SLOAD
Storage store	Stack → storage	SSTORE
Message call	Check taints in stack and memory	CALL, CALLCODE, DELEGATECALL, STATICCALL
Log		LOG0, LOG1, LOG2, LOG3, LOG4
Create		CREATE, CREATE2
Jump	Check taints in stack	JUMPI

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Ruiyao Huang has a patent issued to Peking University, Octa Innovations Ltd. The application number is 2021105770188.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant No. 2022YFB2703301.

Appendix. Taints creation and propagation

Table A.8 demonstrates the rule of creation and propagation in ReenRepair. The creation of the taints only occurs in SLOAD, and the ID of a new taint is based on the execution pc. The taint propagation between stack elements occurs when the instruction in arithmetic, comparison, duplication and swap type is executed. We take ADD as an example: ADD takes the top two elements in the stack as addend and augend, and then the sum of the addition is pushed back to the stack. The taints of addend and augend are merged as a new taints list for sum.

The taint propagates from stack to memory when memory store instructions execute. We take MSTORE as an example: MSTORE takes top two elements in the stack as offset and value. The taint of value will be propagated to memory[offset:offset+32].

The taint propagates from memory to stack when MLOAD or SHA3 execute. We take SHA3 as an example: SHA3 takes top two elements in the stack as offset and length. The hash is calculated by keccak256(memory[offset:offset+length]) and is pushed back to the top of the stack. The taints in memory[offset:offset+length] propagate to hash.

The taint propagates from stack to storage when SSTORE executes. SSTORE takes top two elements in the stack as key and value, which will be recorded as storage[key] = value. The taints of value will be propagated to storage[key].

The taint propagates from storage to stack when SLOAD executes. After new taint created, SLOAD takes the top element in the stack as key and load storage[key] to push back to stack. The new taint, taints of key and taints of storage[key] are merged as a taint list for the element pushed back.

The taint will not continue to propagate but will be checked as a taint sink for instructions in types of Message call, Log, Create and Jump.

Some instructions neither introduce taints nor propagate taints, thus we do not include them into the table. Such as transaction/message parameters loading instructions (ORIGIN, CALLDATALOAD, etc.), block information (BLOCKHASH, etc.) and pop instructions.

References

Albert, E., Grossman, S., Rinetzky, N., Rodríguez-Núñez, C., Rubio, A., Sagiv, M., 2020. Taming callbacks for smart contract modularity. Proc. ACM Program. Lang. 4 (OOPSLA), 1–30.

Bartoletti, M., Galletta, L., Murgia, M., 2019. A minimal core calculus for solidity contracts. In: Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Luxembourg, September 26–27, 2019, Proceedings 14. Springer, pp. 233–243.

Cai, J., Li, B., Zhang, J., Sun, X., Chen, B., 2023. Combine sliced joint graph with graph neural networks for smart contract vulnerability detection. J. Syst. Softw. 195, 111550.

ChainSecurity, 2019. Constantinople enables new reentrancy attack. https://medium.com/chainsecurity/constantinople-enables-new-reentrancy-attack-ace4088297d9.

Chen, T., Cao, R., Li, T., Luo, X., Gu, G., Zhang, Y., Liao, Z., Zhu, H., Chen, G., He, Z., et al., 2020. SODA: A generic online detection framework for smart contracts. In: NDSS.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 337–340.

Diligence, C., 2020. Cross-function reentrancy. https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/#cross-function-reentrancy.

Ferreira, J.F., Cruz, P., Durieux, T., Abreu, R., 2020. Smartbugs: A framework to analyze solidity smart contracts. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 1349–1352.

Ferreira Torres, C., Jonker, H., State, R., 2022. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 115–128.

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: A survey. IEEE Trans. Softw. Eng. 45 (01), 34–67.

Ghaleb, A., Pattabiraman, K., 2020. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 415–427.

Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y., 2017. Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. 2 (POPL), 1–28.

Guo, H., Chen, Y., Chen, X., Huang, Y., Zheng, Z., 2023. Smart contract code repair recommendation based on reinforcement learning and multi-metric optimization. ACM Trans. Softw. Eng. Methodol..

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., et al., 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium. CSF, IEEE, pp. 204–217.

Jiao, J., Kan, S., Lin, S.-W., Sanan, D., Liu, Y., Sun, J., 2020. Semantic understanding of smart contracts: Executable operational semantics of solidity. In: 2020 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 1695–1712.

Jin, G., Song, L., Zhang, W., Lu, S., Liblit, B., 2011. Automated atomicity-violation fixing. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 389–400.

Jin, H., Wang, Z., Wen, M., Dai, W., Zhu, Y., Zou, D., 2021. Aroc: An automatic repair framework for on-chain smart contracts. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/TSE.2021.3123170.

Jin, G., Zhang, W., Deng, D., 2012. Automated concurrency-bug fixing. In: Presented As Part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12). pp. 221–236.

Kalra, S., Goel, S., Dhawan, M., Sharma, S., 2018. ZEUS: Analyzing safety of smart contracts. In: NDSS.

Kelk, D., Jalbert, K., Bradbury, J.S., 2013. Automatically repairing concurrency bugs with ARC. In: Multicore Software Engineering, Performance, and Tools: International Conference, MUSEPAT 2013, St. Petersburg, Russia, August 19-20, 2013. Proceedings. Springer, pp. 73–84.

Liu, H., Chen, Y., Lu, S., 2016. Understanding and generating high quality patches for concurrency bugs. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 715–726.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A., 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1186–1189.

Mueller, B., 2018. Smashing ethereum smart contracts for fun and real profit. HITB SECCONF Amsterdam 9, 54.

Nguyen, T.D., Pham, L.H., Sun, J., 2021. sGUARD: Towards fixing vulnerable smart contracts automatically. In: 2021 IEEE Symposium on Security and Privacy (SP). SP, IEEE, pp. 1215–1229.

Paganini, P., 2020. Uniswap and lendf.me hacked, attacker stole $25 million worth of cryptocurrency. https://securityaffairs.co/wordpress/101895/cybercrime/uniswap-lendf-me-hacked.html.

Rodler, M., Li, W., Karame, G.O., Davi, L., 2019. Sereum: Protecting existing smart contracts against re-entrancy attacks. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society, URL https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/.

Schwartz, E.J., Avgerinos, T., Brumley, D., 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: 2010 IEEE Symposium on Security and Privacy. pp. 317–331. http://dx.doi.org/10.1109/SP.2010.26.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M., 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.

Vitalik Buterin, M.S., 2020. EIP-2929: Gas cost increases for state access opcodes.

Wood, G., 2023. Ethereum: A secure decentralised generalised transaction ledger. https://ethereum.github.io/yellowpaper/paper.pdf.

Wu, L., Wu, S., Zhou, Y., Li, R., Wang, Z., Luo, X., Wang, C., Ren, K., 2020. Ethscope: A transaction-centric security analytics framework to detect malicious smart contracts on ethereum. arXiv preprint arXiv:2005.08278.

Wu, H., Zhang, Z., Wang, S., Lei, Y., Lin, B., Qin, Y., Zhang, H., Mao, X., 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 378–389.

Xue, Y., Ma, M., Lin, Y., Sui, Y., Ye, J., Peng, T., 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1029–1040.

Yu, X.L., Al-Bataineh, O., Lo, D., Roychoudhury, A., 2020. Smart contract repair. ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (4), 1–32.

Yu, L., Lu, J., Liu, X., Yang, L., Zhang, F., Ma, J., 2023. PSCVFinder: A prompt-tuning based framework for smart contract vulnerability detection. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 556–567.

Zhang, Z., Lei, Y., Yan, M., Yu, Y., Chen, J., Wang, S., Mao, X., 2022. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–13.

Zhang, Y., Ma, S., Li, J., Li, K., Nepal, S., Gu, D., 2020b. Smartshield: Automatic smart contract protection made easy. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 23–34.

Zhang, M., Zhang, X., Zhang, Y., Lin, Z., 2020a. {Txspector}: Uncovering attacks in ethereum from transactions. In: 29th {USENIX} Security Symposium ({USENIX} Security 20). pp. 2775–2792.

Zhou, X., Chen, Y., Guo, H., Chen, X., Huang, Y., 2023. Security code recommendations for smart contract. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering. SANER, IEEE, pp. 190–200.

Zhou, S., Möser, M., Yang, Z., Adida, B., Holz, T., Xiang, J., Goldfeder, S., Cao, Y., Plattner, M., Qin, X., et al., 2020. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2793–2810.

Zhuang, Y., Liu, Z., Qian, P., Liu, Q., Wang, X., He, Q., 2021. Smart contract vulnerability detection using graph neural networks. In: Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence. pp. 3283–3290.

Ruiyao Huang received her B.S. in software engineering from Beijing Jiaotong University, Beijing, China, and her M.S. in software engineering from Peking University, Beijing, China, in 2018 and 2021. She is currently working toward the Ph.D. degree in software engineering with Peking University, Beijing, China. Her research interests include blockchain, smart contract security, and program repair.

Qingni Shen received the Ph.D. degree from the Institute of Software Chinese Academy of Sciences, Beijing, China, in 2006. She is currently a professor with the School of Software and Microelectronics and the director of PKU-OCTA Laboratory of Blockchain and Privacy Computing, Peking University, Beijing, China. Her research interests include operating system security, cloud security and privacy, blockchain and privacy computing, and trusted computing. She is a senior member of CCF and an ACM/IEEE member. She is serving as the reviewer for Computers & Security, Journal of Parallel and Distributed Computing, Information Science, the Computer Journal, ACM MM, ICICS and so on, ever as the general chair of ICICS 2019 and as the steering committee member of ICICS since 2019.

Yuchen Wang received his B.S. in software engineering from Harbin Institute of Technology, Harbin, China. He is currently working toward the Ph.D. degree in software engineering with Peking University, Beijing, China. His current research interests include cloud security and blockchain security.

Yiqi Wu received his B.S. in computer science and technology from Peking University, Beijing, China, in 2019. He is currently working toward the master's degree with Peking University, Beijing, China. His research interests include cloud security and system security.

Zhonghai Wu received the PhD degree from Zhejiang University, Hangzhou, China, in 1997. Previously, he worked as a postdoctoral researcher with the Institute of Computer Science and Technology, Peking University, Beijing, China. Since then, he has been involved both in research and development of distributed system, service, and security. He is currently the dean of the School of Software and Microelectronics, Peking University, Beijing, China. His research interests include cloud computing, data intelligence, and system security.

Xiapu Luo is a professor at the Department of Computing and the director of the Research Centre for Blockchain Technology of the Hong Kong Polytechnic University. His research focuses on Blockchain and Smart Contracts Security, Mobile and IoT Security, Network Security and Privacy, and Software Engineering with papers published in top-tier security, software engineering, and networking venues. His research led to more than ten best/distinguished paper awards, including ACM SIGSOFT Distinguished Paper Awards in ICSE'24, ISSTA'22 and ICSE'21, Best Paper Award in INFOCOM'18, etc. and several awards from the industry. He received the BOCHK Science and Technology Innovation Prize (FinTech) for his contribution to blockchain security. He regularly serves in the program committees of top security and software engineering conferences and received Top Reviewer Award from CCS'22 and Distinguished TPC member Award from INFOCOM'23 and INFOCOM'24. He is currently an associate editor for IEEE/ACM Transactions on Networking (ToN), IEEE Transactions on Dependable and Secure Computing (TDSC), and ACM Transactions on Privacy and Security (TOPS).

Anbang Ruan received his B.S. in Computer Science from Hohai University and M.S. in software engineering from Peking University. He obtained his Ph.D. degree in computer science from the Department of Computer Science, University of Oxford. His research interests include research in cyber security and system security for years, and is currently focusing on researching and building the trusted public cloud infrastructure. He has played an important role in Oxfords system security team in participating European-wide projects, funded by FP7 and EPSRC.