Aroc: An Automatic Repair Framework for On-Chain Smart Contracts

Hai Jin, *Fellow, IEEE*, Zeli Wang, Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou

Abstract—Ongoing smart contract attack events have seriously impeded the practical application of blockchain. Although lots of researches have been conducted, they mostly focus on off-chain vulnerability detection. However, smart contracts cannot be modified once they have been deployed on-chain, thus existing techniques cannot protect those deployed contracts from being attacked. To mitigate this problem, we propose a general smart contract repairer named Aroc, which can automatically patch vulnerable deployed contracts without changing the contract codes. The core insight of Aroc is to generate patch contracts to abort malicious transactions in advance. Taking the three most serious bug types (i.e., reentrancy, arithmetic bugs, and unchecked low-level checks) as examples, we present how Aroc automatically repairs them on-chain. We conduct abundant evaluations on four kinds of datasets to evaluate the effectiveness and efficiency of Aroc. In particular, Aroc can repair 95.95% of the vulnerable contracts with an average correctness ratio of 93.32%. Meanwhile, Aroc introduces acceptable additional overheads to smart contract users and blockchain miners. When compared with the state-of-the-art techniques, Aroc introduces either fewer execution overheads or contract codes.

Index Terms—Smart contract, vulnerability, repair, on-chain protection

1 Introduction

Applying smart contracts in blockchain is an epoch-making milestone in blockchain development history. Etherum, a popular platform, is the first smart contract blokchain, so we focus on ethereum in the paper. Since then, blockchain can support diverse business requirements in widespread fields [1], [2]. Consequently, more and more digital assets have been linked to smart contracts. However, they have suffered heavy losses from widespread attacks [3]. The slow mist community [4] reports the money lost by merely hacking ethereum DApps have reached to 531,300,756.56 dollars [5]. What is even worse is that people have gradually lost confidence in blockchain. To prevent contracts from being hacked, many vulnerability detection tools have been proposed, such as Osiris [6], Zeus [7], EthRacer [8], and Solar [9]. Osiris can effectively dig out diverse integer bugs [6]; Zeus can transfer source codes embedded with assertion predicts into LLVM intermediate language, and then verify their safety [7]; EthRacer focuses on event-ordering bugs by combining dynamic symbolic executions and fuzzing technologies [8]; Solar can automatically synthesize attack contracts for a given vulnerable contract [9]. However, they are all off-chain tools and cannot guarantee deployed contracts from being attacked. Besides, merely relying on such off-chain audit technologies is very risky since smart contracts cannot be modified on chain. Therefore, there is an emergent need for on-chain contract protection countermeasures.

Several approaches have been proposed to protect deployed contracts [10], [11], [12]. Sereum [10] dynamically prevents reentrancy attacks through monitoring transaction executions, and throws exceptions when the executions are consistent with predefined vulnerability patterns. However, Sereum can only handle reentrancy bugs. Hydra [11], based on n-version programming, deploys multiple versions of contracts with the same functions and determines execution correctness by comparing the results of different versions. However, it cannot support all opcodes such as `CREATE` and `DELEGATECALL`. The contract upgrade model [12] separates data access codes from logic process codes. When a logic contract becomes vulnerable, it can be substituted by a new repaired version through the proxy contract. But this method cannot handle the vulnerable data contracts. To sum up, current proposals are neither general enough to protect diverse vulnerability types nor flexible enough towards random vulnerability discovery locations. An ideal approach should deal with diverse bug types without assuming that specific locations are bug-free.

To realize this goal, we propose Aroc, an on-chain smart contract repair framework, which can generate patches to plug up loopholes in deployed vulnerable smart contracts. Since anything stored in blockchain cannot be tampered, we must repair contracts without modifying their codes, which

- *Hai Jin and Zeli Wang are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hjin, zeliwang}@hust.edu.cn.*
- *Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {mwenaa, wqdai, zy_hust, deqingzou}@hust.edu.cn.*

Manuscript received 11 January 2021; revised 12 August 2021; accepted 26 September 2021. Date of publication 27 October 2021; date of current version 14 November 2022.
This work was supported by the National Key Research and Development Program of China under Grant 2020YFB1006000, the National Natural Science Foundation of China under Grants 62072202 and 62002125, the Science and Technology Program of Guangzhou, China under Grant 201902020016, and the Fundamental Research Funds for the Central Universities under Grant HUST: 2020JYCXJJ068.
(Corresponding author: Ming Wen.)
Recommended for acceptance by X. Zhang.
Digital Object Identifier no. 10.1109/TSE.2021.3123170

is extraordinarily challenging and tricky. The core idea of this work is to leverage independent smart contracts embedded with secure rules (patches) to block malicious transactions in advance. Specifically, we enhance *Ethereum Virtual Machine* (EVM) to support contracts binding and transactions verification. Given a vulnerable smart contract, Aroc first automatically generates a patch contract containing secure rules based on the repair template, and deploys the patch on blockchain as a regular contract. Second, the owner of the vulnerable contract needs to send a special transaction through our enhanced EVM to wrap the contract with the patch. Finally, any transaction that invokes the vulnerable contract will be first verified by the submitted patch. Then those transactions that trigger potential vulnerabilities will be blocked. Therefore, the original vulnerable contract can be protected from being attacked. Be noted that patches in this work are different from the traditional ones, which do not involve complete functionalities of the vulnerable contracts. Since smart contracts that are deployed on blockchain cannot be modified, one cannot directly apply traditional patches (i.e., in the form of code changes) on them. Essentially, our patches are actually transaction verifiers which contain secure constraints that vulnerable transactions must obey. The constraints should have been programmed in the vulnerable contracts to guard safety. Since deployed contracts cannot be modified, we put the constraints in a separate contract to enforce them work, indirectly repairing the contracts. Since our generated patches serve for the same functionalities as the traditional patches, we referred to them as "patches" in our paper.

Designing Aroc has the following key challenges: *1) How to efficiently access state variables of the vulnerable contracts in patches?* In smart contracts, plenty of vulnerabilities are caused by contract state variables (i.e., global variables). For example, contracts always use the state variable `balance` to record the contract balances. Therefore, the generated patches must fetch these data to perform necessary verifications. Moreover, to restrict the overheads (i.e., contract runtime gas consumptions) within an acceptable range, such a process must be efficient. *2) How to generate the solidity path constraints reaching the vulnerable statements?* Blocking transactions by examining whether they violate secure rules may introduce false positives. We should first examine whether they can pass path constraints to reach the target locations. However, deriving constraints is a non-trivial task. It is because conditions can be enforced in diverse syntactic structures in smart contracts such as `require`, `assert`, `if`, and many other hybrid syntactic structures. *3) How to define effective secure rules for diverse vulnerability types?* Different contract vulnerabilities have distinct features, and even one bug type has diverse manifestation patterns. Hence, using a uniform standard to identify and repair them is hard. *4) How to automatically synthesize and deploy a patch contract to make it effectively work?* Code synthesis is generally recognized as a challenging task. Especially in our task, we need to automatically create demand-driven and brand-new contracts to protect vulnerable smart contracts.

For the first challenge, Aroc utilizes the execution principles of the `delegatecall` [13] to meticulously design the structure of patches. Namely, patches first make the same state variable declarations as the vulnerable contracts. Then Aroc enforces that patches are delegate called by the vulnerable contracts with the verifying transactions in EVM. So patches can directly access the state variables of the vulnerable contracts according to `delegatecall` principles.

For the second challenge, one may think it easy to obtain target path constraints via symbolic execution. Yet, the technology will introduce substantial symbolic variables, which are hard to be mapped to the variables declared in the contracts. Therefore, the path constraints involving such intermediate variables are difficult to be embedded into patch contracts. We elude this by extracting conditional expressions as path constraints from coarse-grained program slices derived through backward analysis on the *Control Flow Graph* (CFG). Since slices range from the starting point of the function to the target line, the target line control flow depends on the conditional expressions in the slices. However, raw conditions may involve local variables of the vulnerable functions, and the patches cannot identify them since only state variables are declared in patches as mentioned above. Besides, we enforce that a patch has the same function head (i.e., function name and parameters) as the vulnerable function head. Therefore, we further rewrite raw conditional expressions by function parameters and state variables as path constraints through data flow analysis. Compared with symbolic execution, we avoid path explosion by backward static analysis and symbolic constraints transformation by data flow analyzing on source codes.

For the third challenge, bearing the core idea that most vulnerabilities are posed by incomplete permission control, we design appropriate access control rules for diverse vulnerability types. Specifically, we take arithmetic bugs, reentrancy, and unchecked low-level calls as examples, since they are the most common vulnerabilities that lead to serious consequences [14]. We present the detailed secure rules and automatic patch generation processes in the following sections. Finally, we design repair templates for the above three bug types. Each template gives an outline of the patches for a specific bug type, which specifies the code entities that are required to be generated. Based on the template and information of the vulnerable contracts such as *Abstract Syntax Tree* (AST), path constraints and bug type, Aroc first generates the patches' AST and then converts it to a solidity contract. Meanwhile, we provide an enhanced EVM to help bind the patches with the targeted vulnerable contracts. Whenever any transaction tries to call the vulnerable contracts, the associated patches will verify the transaction's legality based on inputs and contract storage states. Only those transactions that will not trigger vulnerabilities can be executed, thus protecting vulnerable contracts.

We evaluate Aroc on four kinds of datasets provided by smartBugs [15], EVMPatch [16], ContractGuard [17], and Sereum [10] respectively. In particular, Aroc can automatically patch 95.95% of the contracts with a high correctness ratio of 93.32%. Meanwhile, Aroc outperforms existing the state-of-the-art techniques to some extent. For instance, Aroc usually takes less execution overheads than ContractGuard [17], and introduces less contract codes than EVMPatch [16]. Moreover, Aroc not only can repair traditional reentrancy vulnerabilities but also is effective on two new reentrancy patterns proposed in Sereum [10]. In summary, our contributions are as follows:

- We are the first to propose an automatic on-chain contract protection countermeasure for various vulnerabilities without modifying original contracts.
- We present an efficient way to automatically synthesize patch contracts based on backward slicing and data flow analysis.
- We implement a framework, named Aroc, which consists of an automatic patch synthesis system and an on-chain contract protection system. Given contract source codes, the vulnerable lines, and the bug types, Aroc can automatically protect vulnerable deployed contracts.
- We perform abundant evaluations on four kinds of datasets to demonstrate that our proposal is practical and useful.

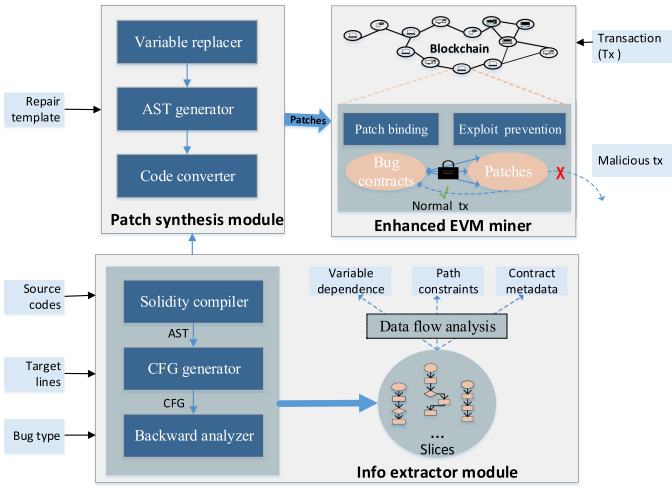


Fig. 1. Architecture of Aroc.

2 Background

Smart Contracts. Blockchain is a distributed decentralized ledger, which is naturally tamper-proof, transparent, and decentralized. Smart contracts are executable codes on blockchain, so contracts also inherit the above features. Currently, the most popular contract programming language is solidity (the main target of this study). Because any execution of contracts is consented by all peers and exposed to the public, contracts can be fully trusted if free of vulnerabilities. However, smart contracts are frequently attacked because of their particular features. For example, tamper-proofing prevents vulnerable deployed contracts from being patched directly; transparency facilitates attackers in identifying fragile targets. Consequently, there is an urgent requirement to mitigate such problems. Next, we introduce some bases of smart contracts that our paper involves.

The Operation Mechanism of Smart Contracts. There are two types of accounts in ethereum, *External Owned Accounts* (EOAs) and contract accounts, which can both interact with the blockchain. A public-private key pair controls EOAs, and the contract codes manage their own contract accounts. EOAs send a transaction attached with bytecodes to deploy a contract. Miners will then create a corresponding contract account holding its vital information such as the creator and bytecodes, and compute a unique ID (the contract address) as its identity in the blockchain. To invoke a function, the contract address, executing function signature (equal to the ID of function in the contract), and the parameters should be explicitly provided. The miners will load the bytecodes from the address and input data (including function signature and parameters) from the transaction into EVM, which is an isolated execution environment for ethereum smart contracts.

EVM will find the bytecodes of targeted function based on the function signature and execute them. Precisely, each byte in the bytecodes corresponds to a low-level instruction, and there is a detailed definition for all instructions, including the operation semantics and gas costs [18]. After executing each instruction, EVM will accumulate all consumed gases from the first executed instruction. EVM will throw exceptions if all gases provided in the transaction are consumed out. Consumed gases will be rewarded to miners. The gas mechanism can protect ethereum from distributed denial of service attacks such as endless computations and also motivate miners to participate in the blockchain consensus.

The Storage Layout and Call Ways of Smart Contracts. Smart contracts have three ways to store data: stack, memory, and storage. The stack stores simple local variables of smart contracts. It is free to use, but has a length limit - 1,024. The stack will be cleared after finishing executing the contract. The memory stores temporary variables such as function parameters and return values. It will be recycled after executing the functions. The storage stores state variables and complex local variables, which will be kept in the blockchain. State variables are global variables in smart contracts.

The costs of storage in EVM are expensive. Each contract account has an isolated storage space, keeping its codes, state variables, transaction history, etc. This space is identified by a unique 120-bit address, i.e., contract address. Contract accounts can only access their own storages. One exception is using the `delegatecall` or `callcode` call method. The `delegatecall` is similar to the library-call method. The callee executes its codes in the caller's context. State variables, `msg.sender` and `msg.value` will keep the same between the callee and the caller. The callee runs as a library. Precisely, if state variables declarations of the caller are declared in the same locations in the callee, the callee can directly access the state variables of the caller. That is, the callee can tamper with them.

Therefore, those contracts that are delegate called must be trusted enough. The `callcode` is an original version of `delegatecall`, in which the callee also uses the caller's contexts while the `msg.sender` and `msg.value` are different. Currently, `callcode` is almost deprecated. Besides, `call` and `staticcall` are common mechanisms to invoke contract functions. In these two calling executions, the callee uses his own storage context and message information. The `staticcall` prevents any modification to state variables, which cannot be used in smart contracts because there are no low-level supported APIs. But it can be invoked through calling functions decorated with keywords `pure` or `view`.

3 Automatical Contract Repair on Chain

3.1 Overview

Fig. 1 shows the overview of Aroc, which mainly consists of three modules: (1)*Info Extractor Module* (*IEM*, see Section 4.1), which extracts the fundamental information necessary for

synthesizing patches given vulnerable contract source codes, target lines, and bug types. Such information includes the *Variable Dependence Relationships* (VDR), path constraints and the contract metadata (e.g., state variable declaration statements). (2) *Patch Synthesis Module* (*PSM*, see Section 4.2), which generates patches based on repair templates and the information provided by IEM; (3) An enhanced EVM (see Section 5), which binds patches with the vulnerable contracts, and transactions that violate the patches' verification will be aborted. The first two modules work off-chain while the last module works on-chain. Hence, Aroc can be divided into two sub-systems: an off-chain automatic patch generation system responsible for patch generation (see Section 4), and an on-chain exploit prevention ethereum responsible for blocking malicious transactions (see Section 5).

IEM focuses on analyzing vulnerable contracts to provide necessary information for PSM. The most significant part is to generate VDR and path constraints. To achieve these goals, IEM first locates the vulnerable functions containing target lines and generates their AST. Then Aroc creates a CFG based on the AST of the vulnerable function. IEM will adopt a backward analysis method to generate program slices based on the CFG. Each slice represents a path from the function entrance to the target line, and the branch conditions in each slice can form a group of path constraints. To derive such constraints, IEM performs data flow analysis on each slice. In this process, IEM iteratively records VDR and re-expresses branch conditions. Finally, all rewritten conditions are regarded as path constraints.

Given the path constraints and VDR provided by IEM, vulnerable lines and bug types supplied by any existing static analysis tool, PSM will choose the corresponding repair templates and determine the transaction verification rules based on the bug types. Then for each vulnerability, PSM will organize the required information in order based on its repair template. After generating a complete AST of the patch, PSM will convert it to a concrete solidity contract.

After patches are deployed to blockchain, the vulnerable contract owner can send a special transaction to specify which vulnerable function should be protected by which patch. Special transactions are the same as common transactions, but their receivers are fixed at an address so that EVM can recognize special transactions through the receiver address. After EVM receives special transactions, it will parse out messages attached in the transaction and link the specified function of the vulnerable contract to the assigned patch. When the contract's vulnerable function is called, our enhanced EVM first launches the patch contract to verify transactions. Any malicious transaction that will trigger vulnerabilities of the vulnerable contracts will be aborted.

3.2 Example

We now show an example to demonstrate how Aroc can automatically repair reentrancy vulnerabilities. To simplify representation, we use A/B to stand for line number A of Listing B. Listing 1 presents the source codes of a vulnerable contract, and (12/1) is the vulnerable line. This contract realizes a simplified transfer function: a user can deposit assets in the contract by calling the `addCredit` function. He can later withdraw his deposits by calling the `withdraw` function. At this time, if the `fallback` function of the user contract is similar to the codes shown in Listing 2, it can recall the `withdraw` function of the `Example` contract. Since credits will not be deducted (13,14/1) until finishing assets transfer (12/1), constraints (9,11/1) are still satisfied in the re-calling process. Transfer operation (12/1) can be executed successfully. The user's `fallback` function can recall the `withdraw` function in the `Example` iteratively until the balances are less than that the user withdraws.

Listing 1. Example of the Reentrancy Vulnerability

```
1  pragma solidity ^0.4.0;
2  contract Example{
3    mapping (address => uint) credit;
4    uint sum = 1000;
5    function addCredit() payable{
6      credit[msg.sender] += msg.value;
7    }
8    function withdraw() {
9     require(credit[msg.sender] > 0);
10    uint value = credit[msg.sender];
11    require(sum >= value);
12  if ((msg.sender.call.value(value)())) {
13        credit[msg.sender] = 0;
14        sum -= value;
15        }
16 }
```

Listing 2. The Fallback Function in the Attack Contract

```
1   function() payable{
2      Example(ExampleAddr).withdraw();
3    }
```

Intuitively, directly swapping the order between (12/1) and (13,14/1) can address this problem. However, deployed contracts are not allowed to be modified. To enable repairing this vulnerability on-chain, we design a repair template of such reentrancy vulnerabilities as shown in Listing 3. It mainly consists of three parts: the variable declaration (3-8/3), the shadow variables initialization (11-15/3), and the verification part (16-22/3). Hereafter, we refer to shadow variables in patches as SVs and state variables in the vulnerable contracts as GVs. Patches generated by the template can directly access GVs of the corresponding vulnerable contracts without modifying their values through part 1. In part 2, we initialize SVs with the values of GVs. Therefore, SVs can represent GVs since their values are equal. In part 3, we can directly verify SVs to judge whether transactions obey the secure states of GVs. Consequently, the verification in part 3 is executed before transferring assets of the vulnerable contracts. Therefore, the patch implements the same functionality as swapping the vulnerable code's order in the deployed vulnerable contract, thus indirectly repairing the contract. The following explains the three parts in detail.

In part 1, `var_X` and `var_Y` are abstract representations of variable declaration statements, which are completely identical with the state variable declarations of vulnerable contracts. As Section 2 describes, if contract A delegate calls contract B, B will be executed in the context of A. Our enhanced EVM forces target contracts to delegate call their bonded patches before they are executed. At this time, patches can directly

access state variables of vulnerable contracts efficiently. To avoid affecting them, we insert shadow variables to represent them, namely `var_X1` and `var_Y1`.

Listing 3. Repair Template of Reentrancy Bug

```
1  pragma solidity verXX ;
2  contract contractName {
3     var_X; // GVs of map type
4     var_Y; // GVs of basic type
5     var_X1; //SVs
6     var_Y1;
7     //create variable update flag
8     mapping (address => bool) flag;
9     function funcName (param1, param2) public {
10    //update shadow variables
11       if(flag[msg.sender] == false){
12        var_X1_name[msg.sender] = var_X_name[msg.sender] ;
13        var_Y1_name = var_Y_name ;
14        flag[msg.sender] = true;
15       }
16       if( path_cons_exprs ){//path constrains
17          Expr_x; //storage variables evoluations
18       }else{
19        var_X_name[msg.sender] = var_X1_name[msg.sender] ;
20        var_Y_name = var_Y1_name ;
21        require(false);
22        }
23   }
24 }
```

In part 2, `flag` marks whether the sender is new. The part 2 will synchronize SVs with GVs for each new sender. Hence, SVs are equal to GVs. In theory, maybe only synchronizing once when deploying patches is also feasible. However, to improve reliability, the initialization is in the account's granularity.

In part 3, we verify transactions, conduct variable evolutions, and update GVs. Aroc guarantees that path constraints (16/3) and evolution expressions (17/3) are equivalent to (9,11/1) and (13-14/1) in vulnerable contracts. One difference between them is that SVs re-express expressions in patches. But SVs reflect the correct values of GVs at any time (see Fig. 2). Hence, conducting verifications on SVs in patches is equivalent to performing verifications on GVs before transferring assets in vulnerable contracts. This method indirectly realizes preventing reentrancy attacks. As Fig. 2 shows, GVs may not reflect real changes (case 2), so part 3 will update GVs based on SVs (19-20/3).

Fig. 2 uses swim lane diagrams to show three possible cases in executing vulnerable contracts. In case 1, the transaction does not satisfy verification rules in the first call, and it is aborted in the patch. SVs keep equal to GVs since initialization. In case 2, SVs reflect correct changes, but GVs keep unchanged. The multiple re-calls in case 3 are legal because all the calls pass verification. Namely, the attacker has enough balances to withdraw. In n times of re-calls, each change of SVs follows an asset transfer in the victim contract. SVs have correct changes. However, GVs keep unchanged until the end. Hence, SVs reflect correct values of GVs at any time. Therefore, SVs of patches can represent GVs of vulnerable contracts, and we should update GVs after blocking malicious transactions.

For the `Example` contract, Aroc generates the patch for it as shown in Listing 4. As explained above, `credit1` and `sum1` can represent `credit` and `sum` of the `Example` contract in real-time. For each call, the patch will first verify whether the path constraints (how the path constraints are generated is shown in the next paragraph) are satisfied (16/4) to prevent reentrancy. Moreover, the value of GVs will be corrected following assets transfer (20-21/4). In summary, the patch helps the `Example` contract prevent the reentrancy attack without affecting its original functions.

Listing 4. Patch of the Example Contract

```
1  pragma solidity ^0.4.16;
2  contract Example{
3    mapping (address => uint) credit;
4    uint sum = 1000;
5    mapping (address => uint) credit1;
6    uint sum1;
7    mapping (address => bool) flag;
8
9    function withdraw() public {
10     if(flag[msg.sender] == false){
11       credit1[msg.sender] = credit[msg.sender];
12       sum1 = sum;
13       flag[msg.sender] = true;
14     }
15
16    if (credit1[msg.sender] > 0 &&
   sum1 >= credit1[msg.sender]){
17      credit1[msg.sender] = 0;
18      sum1 -= credit1[msg.sender];
19    }else{
20      credit[msg.sender] = credit1[msg.sender];
21       sum = sum1;
22      require(false);}
23  }}
```

Next, we will show how Aroc automatically generates a patch for the `Example` contract. Aroc only needs to extract information asked to fit (highlighted lines of Listing 3) as specified in the repair template. Aroc first uses the solidity compiler to generate the AST of `Example`. Then the AST is traversed to collect information. The information mainly consists of two parts. One part is straight information that can be reused in the patch, such as compiler version, contract name, the head of red the function, and state variable declarations. The other part is evaluated information for embodying above three critical parts of the reentrancy repair template. Since the first part information of `Example` contract is the same as the patch, Aroc will reuse them in patches.

Part 1 is straightforward. `var_X/Y` (3-4/3) are declarations of GVs in the vulnerable contract, and `var_X1/Y1` are their corresponding SVs. During traversing the AST, Aroc can identify GVs declaration statements (3-4/4). Aroc then generates the associated SVs declaration statements (5-6/4) based on the GVs'. They will be inserted into the patch AST following the GVs declaration statements AST nodes.

Part 2 is closely related to part 1. After locating the GVs declaration statements as mentioned above, Aroc can determine SVs' names based on GVs' names. According to the fixed formats predefined in the template (12-13/3), Aroc can generate SVs' initialization expressions (11-12/4). Especially for map type variables whose value is address type,

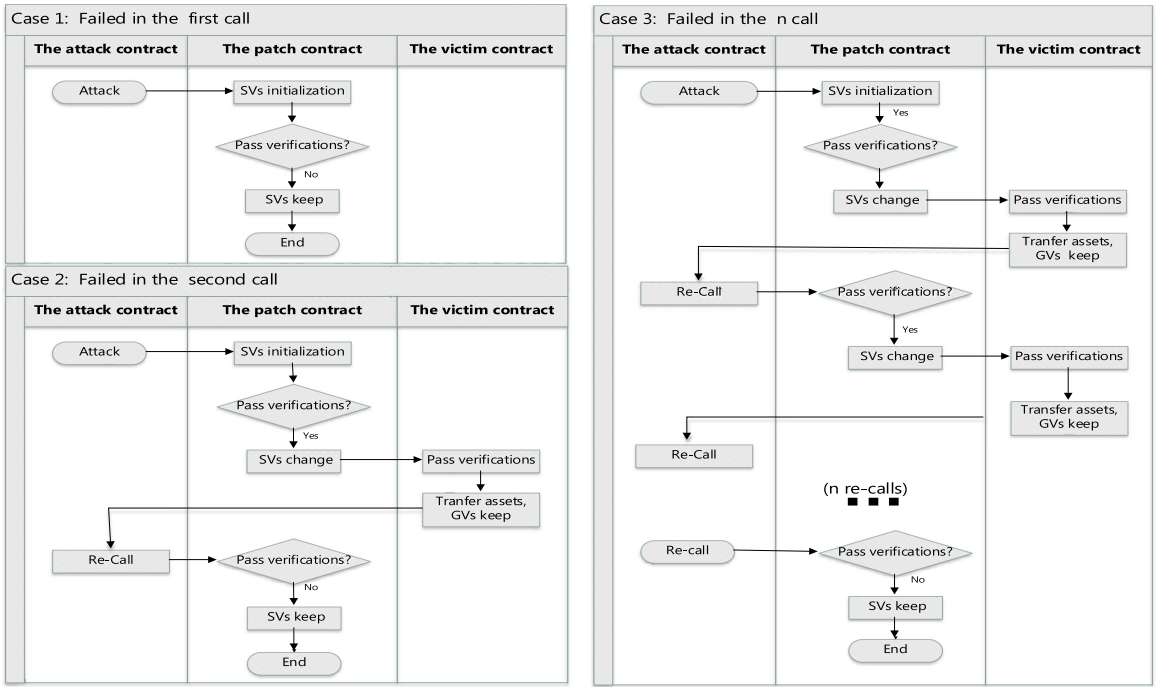


Fig. 2. Three reentrancy cases in Aroc.

Aroc will attach `[msg.sender]` to the tail of the variable names. The most challenging part is to generate path constraints and re-express them by state variables and function parameters. Specifically, Aroc will first generate the CFG (see Fig. 3) based on the AST. CFG presents data and control dependence relationships between variables. Given the target line (12/1), IEM can locate its subordinate function `withdraw` and basic block 4.

Starting from block 4, IEM will analyze each basic block and get all program slices reaching the target block 4. In the `Example` contract, only a slice 4-3-2-1-0 flows from the function's start point to the target line. Then IEM will analyze each slice to obtain path constraints and VDR. Specifically, IEM forward traverses the AST of the generated slices and iteratively records VDR and re-expresses branch conditions.

In `Example` contract, IEM starts from block 1 and gets the first branch expression '`credit[msg.sender]` > 0'. Next, block 2 is an assignment expression. IEM will record this variable dependence '`value` $-$ `credit[msg.sender]`'. The block 3 is a branch expression. IEM will first search existed VDR to find whether `sum` and `value` have mapped value expressions in VDR. As `value` is mapped to `credit[msg.sender]`, the condition in block 3 is re-expressed as `sum` $>=$ `credit[msg.sender]`. IEM ends extracting path constraints when meeting block 4 that contains the target line. Consequently, IEM gets the path constraints: '`credit[msg.sender]` > 0, `sum` $>=$ `credit[msg.sender]`'. After replacing their state variables with the shadow ones, Aroc generates the final path constraints (16/4) for the patch. As for `Expr_x`, IEM will explore GVs-related assignment expressions (13-14/1) following asset transfer operations and also re-express them as mentioned above to generate `Expr_x` (17-18/4). GVs update expressions (20-21/4) can be directly derived by reversing SVs initialization expressions of part 2. (22/4) aims to inform EVM that current transaction is malicious by throwing exceptions. Then EVM will abort this transaction by the `require(false)` statement.

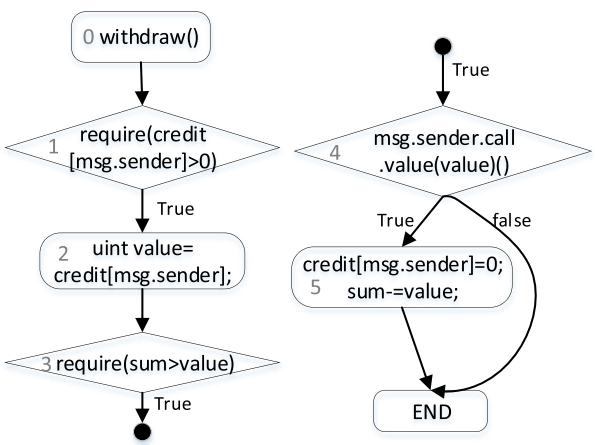


Fig. 3. Control flow graph of the example contract.

4 Off-Chain Automatic Patch Generation

This section illustrates how Aroc designs and implements the off-chain patch generation system, so that Aroc can automatically generate patches for vulnerable smart contracts. We will describe it in two parts: the info extractor module 4.1 and the patch synthesis module 4.2.

4.1 Info Extractor Module

IEM runs on the vulnerable contracts, mainly focusing on extracting path constraints and VDR. Meanwhile, IEM also identifies certain contract metadata that will be used in patches. We will next detailedly describe the design and workflow of IEM.

Given contract source codes, multiple pairs of the vulnerable line and bug type, IEM will first locate the function of each vulnerable line by traversing the contract codes, and

group these lines based on their enclosing functions. IEM then leverages the solidity compiler to generate the AST of the original contract, which contains the detailed syntax structures such as contract definitions, function declarations, statement type, etc. Finally, IEM will traverse the AST to derive information that can be reused in the patch, such as the compiler version and the contract name for GVs.

For each vulnerable function, IEM extracts the function's AST from the original vulnerable contract. Subsequently, IEM performs the following operations to generate path constraints, which is the most challenging part in generating patches. The basis to achieve such a goal is creating a CFG based on the AST. However, a significant challenge in generating CFG is that Aroc needs to identify and handle diverse branch statements in solidity such as `require` and `assert`. Although they do not have explicit `then` and `else` statements, it is implicit that the former contains subsequent blocks and the latter represents program termination according to their semantics. We overcome this challenge by analyzing their representations on source codes and summarizing the location of conditions in different statements.

Based on the CFG, IEM can generate program slices efficiently given a target line. Specifically, IEM backward traverses the CFG from the target location to the entry point of the function. All statements of a basic block flowing into the target line will be recorded into slices. Generating such coarse-grained slices helps Aroc work more efficiently. Since slices are mainly used to generate path constraints, which is closely related to branch conditions, and several irrelevant expressions in basic blocks do not have effects on final results. Therefore, it is not necessary for our slices to extract statements that only have dependence relationships with those variables in the target line as adopted by traditional slice generation methods. If n basic blocks point to the same basic block that flows into the target line, the current part slice will be further to form n slices following different paths. Each slice represents a unique path from the entry point of the function to the target location. Each path is identified by a series of sequence numbers of its contained basic blocks, hereafter referred as the path identifier. IEM will finally obtain all slices reaching the target line. Similarly, IEM will derive all slices for the remaining target lines.

After obtaining all slices for all target lines within a function, to reduce potential static analysis overheads in analyzing the same slices redundantly and reduce the size of patches, IEM will extract the longest common path prefix by comparing the path identifiers of all slices. The common path prefix means a string of continuous path identifiers that is shared by multiple paths, which also represents the common path consisting of the same CFG basic blocks. Those VDG and path constraints that are extracted from the common path are referred as the common VDG and common path constraints hereafter. By contrast, paths that are unique to a slice are referred as the personal path, whose VDG and constraints are denoted as the personal VDG and personal path constraints. For paths that have a common prefix (referred as the first level prefix), IEM will further measure whether some paths have a common prefix following the first prefix.

The above processes are repeated until no paths that have a common high-level prefix have a low-level common prefix. Considering these common paths, in the subsequent analysis IEM can try to avoid analyzing the same path segments, lessen patches size and further reduce patch execution consumptions. To sum up, each slice is a CFG path starting from the function entry point to the target line, each path has a path identifier, and some paths may have multiple common path prefixes.

For each slice, IEM executes the following operations. First, it examines whether the current slice involves common path prefixes. If the slice contains a common prefix that has not been analyzed, IEM will flag the last basic block in the common path as the 'breakpoint'. IEM will then start traversing each slice. Specifically, IEM analyzes each basic block of the slice from the entry point of the vulnerable function. When this traverse process reaches the 'breakpoint', IEM will record the path constraints and the VDR of the current common path prefix, namely, the path starting from the function start point to the breakpoint. By default, the function entry point is the first 'breakpoint'. On the contrary, if the slice contains a common prefix that has already been analyzed, IEM can directly fetch the recorded information of this path prefix, and jump to the basic block subsequent to the common prefix in the slice to proceed analyzing. If the slice does not contain a common prefix, IEM analyzes it to get personal path constraints and VDG.

Meanwhile, in analyzing basic blocks, IEM focuses on the assignment and conditional expressions due to the following reasons. First, the control flow dependence relationships are established by conditional expressions, and thus we need the relationships to derive path constraints. Second, the assignment expressions can directly affect VDR. To examine whether transaction inputs obey path constraints, we need to remove local variables in the path constraints based on VDR. Therefore, IEM highlights these two expressions.

For each assignment expression, IEM will record the VDR between the left value of the assignment (i.e., denoted as `lvalue`) and the right value of the assignment (i.e., denoted as `rvalue`). Specifically, IEM will identify all variables in `rvalue`. For each variable, it searches existing VDR to check whether it has mapped value expressions. If any mapping exists, the `rvalue` will be re-expressed by replacing the variable with its mapped expressions. Finally, the mapping relationship between the `lvalue` and rewritten `rvalue` will be recorded in VDR. In this case, all local variables will be rewritten in terms of the function parameters and state variables.

For each conditional expression, IEM first identifies all variables involved in it. Similarly, for each variable, IEM searches existing VDRs to determine whether it has mapped value expression, and thus the conditional expressions can also be rewritten. Finally, all local variables in conditions can be removed. These re-expressed conditions are regarded as path constraints. In particular, if target lines contain reentrancy vulnerability, after finishing analyzing slices, IEM will continue traversing the remaining paths in the complete CFG to search GVs change statements. At last, each target vulnerable line has multiple pairs of personal path constraints and VDG (together with GVs change statements if the bug type is reentrancy). Besides, a mapping between the common path prefix, constraints and VDG will be recorded.

Although in this procedure, it is complicated to handle diverse syntax structures, Aroc can still work efficiently by analyzing slices to extract path constraints and reusing

information in common path prefixes. Moreover, since slices derived by backward analysis consider all paths flowing to the target line without exploring irrelevant paths, the extracted path constraints are more sound and preciser comparing with forward program analysis.

4.2 Patch Synthesis Module

PSM is responsible for synthesizing the patch smart contracts. It mainly performs three steps: (1) Integrating path constraints - a function may have multiple vulnerable lines, and a line may also have multiple paths flowing into it. For a function, PSM integrates the common constraints of these paths, to reduce the size of patches and further reduce patch deployment costs together with execution consumptions. Specifically, PSM processes all constraints simultaneously and does not distinguish them between different target lines since those paths flowing into the same lines or different lines are distinctive. Consequently, if paths share common path prefixes, they will be classified into one group. Then PSM iteratively organizes path constraints of all slices following two principles. First, for each group of path constraints, the first part is the common constraints while the second part is the customized path constraints. Second, if several paths in the second part still share common path prefixes, the prefix will be extracted as the second common path prefix for the paths, that is, extracting the common constraints is an iterative process. (2) Extracting secure rules - following the customized path constraints, PSM will generate and place secure rules in related locations of the repair template. Different bug types have different patterns of secure rules, and the detailed generation process is illustrated in the following subsections. Finally, the structure of those integrated path constraints is similar to that as shown in Listing 5. (3) Synthesizing information - PSM first generates a patch contract framework, which contains almost all the required information of the patch except for the function bodies. Specifically, PSM first chooses a repair template based on the detected bug type. Second, PSM feeds the information provided by IEM, such as the compiler version and state variable declarations, to the corresponding locations of the template. In this step, if the bug type is reentrancy, PSM will create shadow variable declarations as mentioned in Section 3.2; otherwise, PSM will not create these. A contract framework has been created in this step. Next, PSM aims to to create the body of the patch functions. Since different bug types have diverse processing procedures, we introduce the repair procedures of our target vulnerabilities as follows.

Listing 5. The Structure of Integrated Path Constraints

```
if(common_path_cons1){
  if(personal_path_consA){
        repair_rulesA;}
  if(common_path_cons2){
    if(personal_path_consB){
        repair_rulesB;}
    if(personal_path_consC){
        repair_rulesC;} }
  ...
}
if(common_path_cons2){...}
...
```

4.2.1 Reentrancy Vulnerability Repair

Reentrancy is the culprit of the notorious DAO attack caused by the wrong order between asset transfer and state variables change operations [19]. Specifically, state variables are changed after asset transfer operations. In this case, if the receiver is a contract with a well-designed `fallback` function, it can reenter (call back) the sender contract to drain assets iteratively. Because expressions of state variable changes are not executed in this iterative process, such a process will not stop until the contract balances are less than the withdrawal amount. Repairing reentrancy through our patches encounters two significant challenges: 1) Aroc cannot swap the incorrect operations directly, since vulnerable deployed contracts cannot be modified; 2) patches cannot get the transaction execution traces of vulnerable contracts to protect against reentrancy like Sereum [10], since patches are executed before vulnerable contracts. Nevertheless, our core idea of repairing reentrancy vulnerabilities is to swap the order between state variables change and asset transfer expressions without modifying vulnerable contracts. As introduced in Section 3.2, the reentrancy repair template (Listing 3) mainly consists of three parts (variable declaration, shadow variable initialization, and verification).

For part 1, to enable patches efficiently accessing GVs of the original vulnerable contracts, the template takes full advantage of the `delegatecall` principles. That is, the template declares the same GVs (3-4/3) as those of the original. Under such an organization, when vulnerable contracts delegate call patches, patches can directly get values of GVs. To avoid impacts on GVs of vulnerable contracts, Aroc introduces the shadow mapping mechanism. That is, each GV has a corresponding SV, which will keep consistent to the corresponding GV's real value through part 2, and thus representing the corresponding GV. In part 2, patches initialize the value of SVs with GVs' value to make them equivalent. To try to reduce intervention between contract users, for each new sender, SVs will be synchronized to the value of GVs. Part 3 is responsible for implementing secure rules and verifying transactions. Because (17/3) in patches is derived by rewriting original storage variables change statements with function parameters and state variables, it is equal to state variables change statements following asset transfer of vulnerable contracts.

According to the design of Aroc architecture, patches are executed before vulnerable contracts, so (17/3) in patches is executed before asset transfer in vulnerable contracts, which indirectly swaps the order, i.e., secure rules. Path constraints (16/3) generated by Aroc for patches are equivalent to path constraints in vulnerable contracts. Path constraints involving checks on state variables of state variables change expressions such as (9,11/1), so (16/3) in patches can check the legality of transactions. Moreover, as Fig. 2 shows, GVs may not reflect correct values in blockchain sometimes when reentrancy attacks are launched. Luckily, SVs keep consistent with correct values of GVs so that the repair template will enforce updating GVs with SVs after each failed call.

4.2.2 Arithmetic Vulnerability Repair

Arithmetic errors are pervasive in smart contracts. Since contracts involve many asset transfers, there are plenty of

TABLE 1
Repair Rules of Arithmetic Bugs

Result = LHD op RHD	Repair rules
op = '+'	Result >= RHD & Result >= LHD
op = '-'	LHD >= RHD
op = '*'	if(RHD !=0) Result / RHD == LHD
op = '/'	RHD != 0

modifications on account balances. Arithmetic bugs will be triggered when the computed results excess the data type range, causing that the real computed results are not equal to the intended one. Hence, patches here need to verify whether arithmetic results on the transaction inputs are correct. We should define verification/secure rules for different arithmetic operations. According to the popular library (SafeMath) for securing smart contracts [20], we summarize four rules (see Table 1). In the table, 'Result = LHD op RHD' stands for vulnerable arithmetic expressions, 'Result' means the computing expression, 'RHD' and 'LHD' means the right and left operand, respectively. If the operator is +, Aroc asks the sum should be larger than any addend. If the operator is −, Aroc asks the difference should be less than the minuend. If the operator is *, Aroc asks the quotient of the product and one multiplier is equal to the other multiplier if no multiplier's value is zero. If the operator is /, Aroc asks the divisor is not equal to 0.

Listing 6 shows the repair template for arithmetic bugs. The compiler version, contract name, the head of vulnerable functions, and state variable declarations (1-4/6) are directly copied from the corresponding vulnerable contracts. The same state variable declarations help patches directly access the value of state variables, and the same function head helps patches directly parse transaction input data. The remaining of the template mainly consists of two parts: path constraints (5/6) and secure rules of transaction verification (6/6). Part 1, consisting of path constraints, aims to judge whether transactions can reach the vulnerable target location. Path constraints are rewritten branch conditions of the vulnerable contracts provided by IEM.

Listing 6. Repair Template of Arithmetic Bug

```
1 pragma solidity verXX;
2 contract contractName {
3   var_X;//original storage variables
4   function funcName(param1, param2) public {
5    path_cons_exprs;//path constraints
6    require(repair rules);//transaction verification rules
7   }
8 }
```

Part 2 is the template's core part, focusing on verifying whether transactions are legal through secure rules. As shown in Table 1, PSM should first extract operators and operands of vulnerable arithmetic statements to construct secure rules. However, this task is a big challenge. On the one hand, arithmetic operations have diverse representations. They can be combined with assignment operations, such as +=, −=, *=, and they may also be tangled with other complicated expressions such as correct arithmetic operations or logic expressions. On the other hand, arithmetic operations can exist in diverse code elements, including conditional expressions, block statements, assignments, and so on. Consequently, splitting and identifying target arithmetic expressions are not easy. We tackle this challenge by presenting each syntax structure as a 'class', which includes member attributes to store data type of return values, operators, operands, etc, and member functions to set and get these attributes. When IEM traverses the AST of a vulnerable contract, it will set values of different attributes.

Given a vulnerable arithmetic expression, PSM locates its AST node and gets its operators and operands. Then PSM can choose appropriate repair rules based on operators, further constructing secure rules based on operands and operators. The operands in the rules are rewritten based on the VDR provided by IEM. Feeding all such information as mentioned above into the arithmetic repair template, the associated patches can be generated.

4.2.3 *Unchecked Low-Level Call Vulnerability Repair*

All exceptions in traditional languages are thrown, but some functions (`call`, `callcode`, `delegatecall`, `send`) in solidity only return a boolean value when exceptions occur. If contracts lack checks on such execution results, subsequent operations will continue even when these functions failed, thus causing hazardous consequences if the following operations are sensitive. One famous example of this bug type is found in the smart contract game - 'The King of the Ether Throne' (KotET) [21]. Gamers can pay KotEt to get the rights of the King, but attackers may deliberately fail to pay, such as attaching insufficient gases. However, because of lacking checks, the attackers can still obtain the rights.

Repairing this kind of bug is tricky since we can get the call function execution results only after the vulnerable contracts have been executed. But patches are called before them. However, we can execute them under an identical environment in the patch to check the execution results. Therefore, we design the repair template for these bugs as shown in Listing 7. Except for the basic contract metadata (1-4/7) replicated from the vulnerable contracts, it mainly consists of two parts: path constraints (5/7) and the verification on the function call (6-10/7). The first part is the same as part 1 of the arithmetic repair template provided by IEM and can be directly placed in the fixed location.

The second part is the core of the template, which focuses on verifying whether the call will succeed for each transaction. The low-level call including the call expression semantics and execution context in (6/7) should be entirely equal to the vulnerable function call of the vulnerable contract. The expression 'external_call_statement' is an equivalent sentence obtained by rewriting the vulnerable call expression based on the VDR provided by IEM. Patches are delegate called by the vulnerable contracts, so execution contexts, including `msg.sender`, `msg.value`, and storage states, are identical to the vulnerable contracts. In a word, the call statement in patches (6/7) is equivalent to that in the vulnerable contracts. Any transaction that fails in the call will be blocked.

Moreover, to elude effects on the original contracts, Aroc rolls back all state changes caused by patches. Implementing this is a challenging project. Fortunately, solidity provides

two keywords, `revert` and `assert`, both of which reset the current execution and throw exceptions when assertions are not satisfied or the `revert` statement is executed. Besides, the differences between them make Aroc easier to distinguish whether transactions pass checks. That is, `revert` refunds remaining gases to users while `assert` confiscates all gases. In other templates such as the arithmetic repair template, the `require` condition expressions, responsible to deal with malicious cases, also terminate execution but refund the remaining gases . Therefore, in the underlying EVM implementation, we will modify EVM to abort exception-throw transactions with remaining gases (`require`). But exception-throw transactions without remaining gases (`assert`) are regarded as benign.

Listing 7. Repair Template of Unchecked Low-Level Call Bug

```
pragma solidity verXX;
contract contractName {
  var_X;
  function funcName(param1, param2) public {
    path_cons_exprs; //path constraints
    if(external_call_statement){
      assert(0==1);
    }else{
      revert();}
  }
}
```

In summary, patch contracts will execute the same low-level call as the target vulnerable line in advance. As mentioned above, due to the `delegatecall` principles and contract storage layouts, patches share the same execution contexts with the vulnerable smart contracts. So execution results of the low-level call in patches must be identical with the vulnerable contracts. If the executions in patches fail, the transaction will also fail to execute in related vulnerable contracts. Consequently, we can judge execution results of vulnerable calls in advance and timely abort transactions that will fail in the vulnerable call.

5 On-Chain Contract Protection System

Patches are deployed as normal smart contracts. After they have been deployed, users need to send special transactions to link patches to the related vulnerable contracts. Special transactions have the same structures with the normal ones, but their receiver addresses are fixed at a predefined address and the 'data' field carries customized messages according to different application scenarios. Ethereum miners can distinguish them through the receiver address, and parse messages delivered through the 'data' field of transactions. These messages help Aroc protect deployed contracts in a secure controlled way. Specifically, to prevent malicious bindings, Aroc checks the identity of special transaction senders based on the message. Only target contracts deployers have such a right. When EOAs or contracts call vulnerable contracts, the input data will be enforcedly forwarded to the related patches by EVM. EVM will check execution results of patches. Only transactions that pass checks can call target vulnerable contracts. Otherwise, they will be aborted. So the vulnerabilities in contracts cannot be exploited, patches repair them indirectly.

5.1 Patch Binding

To support the contract binding, we add new variables in the account struct of EVM, which record binding relationships for the current contract. Each smart contract can be bound to a patch, and the patch is deployed like normal contracts. The bound patch can be replaced with a new one at any time through special transactions. The receiver of those special transactions is fixed as '0x1000000000000000000000000000000000000000'. Payloads in the 'data' field point out the address of the patch and the vulnerable contract, the vulnerable function's signature, and the nonce of vulnerable contract deployment transaction. EVM identifies special transactions through the recipient address and parses out above messages. To avoid malicious binding, EVM only allows the vulnerable contract deployers to patch. In ethereum the contract address is computed based on the sender address and the contract deployment transaction nonce as the Identity 1 [18] shows, where RLP is the RLP encoding function and KEC is the Keccak 256-bit hash function, and $\mathscr{B}_{96..255}$ extracts a binary value ranging the indices [96, 255]. Hence, Aroc can determine whether the current sender is the deployer of the vulnerable contract by comparing the address computed from `msg.sender` and nonce in special transaction with the real vulnerable contract address. If the special transaction sender has the authority, Aroc can record the binding relationship between the patch and the vulnerable function specified in the special transaction into the vulnerable contract account.

$$a \equiv \mathscr{B}_{96..255}(KEC(RLP(sender, nonce))). \tag{1}$$

Since patches are independent smart contracts, they will not be confined by bound vulnerable contracts. They run like libraries by the `delegatecall` mechanism. As long as vulnerable functions have identical vulnerabilities and path constraints, they can share the same patches. This feature of high reusability facilitates reducing costs of vulnerable contract owners.

5.2 Exploit Prevention

For each transaction, our enhanced EVM will search the blockchain's state to determine whether the target contract has bound patches. Meanwhile, by extracting the first four bytes function signature from transaction payloads, Aroc can further determine whether the target function has bound patches. If there are, Aroc will first repack the transaction to delegate call patches. Since the raw EVM limits that `delegatecall` can only be triggered by corresponding EVM opcodes, we change the original EVM to support `delegatecall` to be called externally. EVM will delegate call corresponding patches with repacked input payloads. `delegatecall` provides patches with the same execution environment as the original contracts. They share same storage space, state variable values, `msg.sender`, etc. Hence, patches can verify transaction input data and original contract storage states in the almost same environment as the vulnerable contract's. Moreover, the repair function in a patch has the same head as the target vulnerable function, so the patch can directly parse input payloads.

If inputs pass patches' verification, their subordinate transactions can then be transferred to call the vulnerable

contracts. If EVM throws exceptions and the remaining gases are not zero, transactions will be aborted. Since all transactions that trigger bugs are blocked by patches, contracts with vulnerabilities can run securely. Moreover, because contracts can be called by EOAs through transactions and also contracts through message calls, we add the exploit prevention function in both the transaction processing module and call instructions (`CALL`, `CALLCODE` `DELEGATECALL`) implementation module of EVM. Since we enhance EVM to enforce that transactions are verified by related patches before triggering vulnerable contracts, attackers cannot bypass patch verifications. As long as they call contract functions by transactions or message calls, the exploit prevention function in EVM will be triggered.

6 Evaluation

In this section, we evaluate the effectiveness and efficiency of Aroc, including the usability of patches and the performance of Aroc system. Specifically, we evaluate the performance of Aroc with respect to three types of bugs: reentrancy, arithmetic bugs, and unchecked low-level checks.

Experimental Setup. The off-chain patch generation system is implemented in C++ based on SIF [22]; and the on-chain protection system is based on go-ethereum 1.9.0. All experiments are conducted on the Linux server with 2.40 GHz 64-bit Intel Xeon CPU E5-2630 v3 processor with 8-cores, 64 GB RAM, and the 18.04 Ubuntu operating system. We apply Aroc on abundant datasets that consist of the following diverse data sources:

1) 98 smart contracts with the location of bugs provided by [23], in which 15 contracts contain arithmetic bugs, 31 contracts contain reentrancy bugs, and the other 52 contracts have unchecked low-level call bugs. This dataset organized by the DASP taxonomy has been widely used by existing studies [15], [24].

2) 500 smart contracts that are randomly selected from the EVMpatch dataset [16]. Although EVMPatch provides the bytecodes of 14,107 ethereum mainnet smart contracts, Aroc requires the source codes. Therefore, we crawl the source codes from etherscan [25], a famous ethereum blockchain explorer. Finally, we collect the corresponding sources for 4,018 contracts. We utilize Osiris [6] to get the vulnerable lines for these contracts. Moreover, the evaluations on these contracts need benign and malicious transactions to determine whether patches affect the original functionalities and prevent bugs from being exploited. Such a process requires huge manual efforts. Therefore, we randomly sampled 500 contracts out of the 4,018 contracts.

3) Three contracts with the new reentrancy patterns that are provided by Sereum [10]. Sereum only provides three contracts for the new patterns in its evaluations, and thus we utilize the three contracts to measure whether Aroc can repair other reentrancy patterns.

4) Six contracts provided by ContractGuard [17]. ContractGuard presents the detailed analysis on six contracts in its paper while the tool is not open-sourced. Therefore, we evaluate Aroc on the six contracts, to compare the experimental results with contractGuard presented results.

6.1 Usability of Patches

The usability is to measure whether our patches can be effectively and efficiently used in practice. If the usability is low, the generated patches will not be accepted and used by widespread users, and thus the practical usefulness of our system will be significantly compromised. Specifically, the usability mainly includes two parts, which are the *correctness* and *gas costs*, and the latter involves deployment gases and execution gases. If patches have low correctness, the protected vulnerable contracts can still be attacked. If patches have high gas costs, they will be hardly adopted. To evaluate such usability of our generated patches, we conduct experiments on two kinds of datasets, including one provided by [15] and the other randomly selected from the EVMpatch dataset [16] as mentioned above. The first dataset can verify whether Aroc can effectively work on diverse vulnerabilities, and the second dataset can verify whether Aroc works on real-life large and complicated smart contract projects.

6.1.1 Correctness

Since Aroc includes off-chain patch generation and on-chain contract protection, we evaluate the correctness from these two parts. The correctness of the off-chain process is examined by whether the generated patches contain complete path constraints and correct secure rules, while that of on-chain means the correctness of blocking malicious transactions and passing benign transactions. Table 2 presents the evaluation results. The columns of *Dataset properties* in Table 2 describe features of the dataset, in which '*Contract amount*' and '*Bug amount*' mean the number of vulnerable contracts and the corresponding number of bugs respectively. The columns of '*Off-chain analysis*' in Table 2 measure the correctness of off-chain patch generation. Specifically, '*Imprecise patches*' indicates the number of patches that include correct secure rules and incomplete path constraints; '*Precise patches*' shows the number of total correct patches. Those patches with incorrect secure rules are regarded as failed cases. The '*Correctness*' is used to measure how many bugs' patches are correctly generated, which is computed by the amount of precise patches divided by the amount of all bugs. The '*Patch ratio*' is computed by the sum of '*Imprecise patches*' and '*Precise patches'* divided by the '*Bug amount*', aimed to measure how many bugs can be protected by Aroc. The columns of '*On-chain protection*' in Table 2 measure the correctness of on-chain contracts protection. Specifically, '*Failed rep.*' (FR) indicates the number of bugs that we cannot attack successfully; '*False pos.*' (FP) indicates the number of patches that block benign transactions; '*False neg.*' (FN) indicates the number of patches that pass malicious transactions; '*True pos.*' (TP) indicates the number of patches that block malicious transactions without affecting benign transactions; and '*Correctness*' indicates the proportion of TP to all patches that can be tested on-chain. We will introduce the evaluation processes and results next.

Correctness of the Off-Chain Patch Generation System. In this assessment, Aroc first generates patches for all the 598 contracts. As the columns in 'Off-chain analysis' of Table 2 shows, Aroc can successfully generate correct patches for most vulnerable contracts, achieving the correctness ratio of

TABLE 2
The Correctness of Aroc

Dataset properties				Off-chain analysis				On-chain protection				
Data source	Bug type	Contract amount	Bug amount	Imprecise patches	Precise patches	Correctness	Patch ratio	*Failed rep.*	*False pos.*	*False neg.*	*True pos.*	Correctness
smartBugs [15]	Arithmetic	15	23	0	23	100%	100%	1	0	0	22	100%
	Reentrancy	31	31	2	20	64.52%	70.97%	0	2	0	20	90.91%
	Unchecked low-level call	52	78	5	65	83.33%	89.74%	13	5	0	52	91.23%
	Total	98	132	7	110	83.33%	88.64%	14	7	0	94	93.07%
EVMPatch [16]	Arithmetic	500	1252	63	1148	91.69%	96.72%	264	63	0	884	93.35%
Total	-	598	1384	70	1258	90.90%	95.95%	278	70	0	978	93.32%

90.90%. Moreover, since imprecise patches contain correct secure rules, they can also protect vulnerabilities from being exploited. By taking them into account, Aroc can patch 88.64% of the bugs in the smartBugs dataset and 96.72% bugs in the EVMPatch dataset, with the total patching ratio of 95.95%. In this off-chain analysis, comparing with another two vulnerabilities, Aroc needs to search for storage variables change expressions following asset transfers for reentrancy vulnerabilities. But the locations of the change expressions are diverse, they can even locate in different basic blocks of CFG with respect to the vulnerable lines. It is more difficult for Aroc to overcome this problem, so Aroc achieves a relative low correctness for repairing reentrancy bugs.

Aroc cannot generate patches for all bugs. It is because that the vulnerable lines can exist in diverse syntactic combinations while Aroc works on the AST of programs. Therefore, Aroc may fail in generating patches, when many paths flow into the target lines or the structure of target lines is too complicated. Listing 8 shows an example that Aroc fails to generate patches, whose address in the ethereum mainnet is 0x4f8144764a115b868cb14d71576ccf961f943452. The target line is highlighted at line 19, where `nextCorIndex` is a contract storage variable. In total, there are six paths flowing into the target line. Each path has its own particular VDG and path constraints, and thus they should be handled individually. However, to reduce the patch size, we try to integrate these paths. Due to the abundant paths and complicated path combinations, Aroc fails to generate a patch. Especially, if a function has too many bugs, the number of paths will sharply increase. Consequently, there are more kinds of path combinations, and Aroc may fail. Sometimes, if the target line is too complicated, Aroc may fail to generate patches neither. For example, the 'BattleOfTitansToken' contract, whose address is 0xdbd23bde88d4169fb60b0d9966fa1bef8eb74179 in the ethereum mainnet, has the vulnerable line '`require (balances[msg.sender]>=_value+frozenAccount[msg.sender]*forbiddenPremine/(86400*1))`'. This line has four arithmetic operations and the two contain bugs in the operand of the parameter of the `require` statement. Moreover, six paths flow into this vulnerable line. Due to the complicated syntax, Aroc also fails to generate the patch for this vulnerable line. Similarly, Aroc-generated patches may be imprecise due to the tangled paths, complicated target lines, or the subordinate function with multiple bugs.

The fifth column of Table 2 presents the number of bugs that do not be patched precisely. To obtain the correctness of patches, we manually verify whether they involve correct path constraints and secure rules of all the bugs. The precision loss of patches directly influences its effectiveness in blocking transactions. As introduced above the imprecision of patches means the patches do not contain complete path constraints. Thus, some benign transactions that violate secure rules while cannot reach the target lines may be blocked by mistakes on-chain, causing false positives. This also explains why the amount of false positives on-chain (the tenth column of Table 2) is equal to the number of imprecise patches off-chain (the fifth column of Table 2). To conclude, evaluation results on the two kinds of datasets demonstrate that Aroc can repair most of the contracts for the three bug types and real-world contracts.

Listing 8. Example of the Failed Patch Peneration

```
1  function createTokens(address _cor) payable {
2   uint _amount = msg.value;
3   uint cAmount = _amount;
4   uint returnAmount = 0;
5   if (_amount > (maxCap - ethRaised)) {
6    cAmount = maxCap - ethRaised;
7    returnAmount = _amount - cAmount;
8   }
9   if (ethRaised + cAmount > minCap
10   && minCap > ethRaised) {
11   MinCapReached(block.number);
12   }
13  if (ethRaised + cAmount == maxCap
14  && ethRaised < maxCap) {
15   MaxCapReached(block.number);
16  }
17  if (corList[_contributor].cAmount == 0){
18   corIndexes[nextCorIndex] = _contributor;
19   nextCorIndex += 1; }
20   ....
21  }
```

Correctness of the On-Chain Exploit Prevention System. We assess whether patches can effectively protect the bugs from being exploited without affecting the original functionalities. As the ninth column presents, we cannot conduct on-chain evaluations on all bugs with precise patches due to

the three following reasons: 1) we cannot launch attacks on the bugs. For example, if the caller of the unchecked low-level calls is a constant, we cannot assign it as the contract attacker; if path constraints or variables that trigger bugs are related to the future block timestamp, these conditions can hardly to be established. 2) Patches cannot pass compiling due to undefined variables. For example, patches include calling other functions of the vulnerable contract or storage variables inherited from parent contracts. 3) The parent contracts inherited by the vulnerable contracts include global variables. Since Aroc generates patches based on the AST of the original vulnerable contracts, patches do not contain declarations of the state variables of the parent contracts. Due to the inconsistent order of global state variables, the values of the state variables between patches and vulnerable contracts are not identical. For the remaining patches, we generate 20 test cases with benign and malicious inputs respectively for each bug.

For each vulnerable contract, we deploy it and the associated patches. We first run the vulnerable contracts without patches, and then check whether they can execute all the generated test cases successfully. We then send a designed transaction to establish the relationship between the patch and the vulnerable contract. We again test the vulnerable contract using the above test cases to verify whether the patch can block malicious transactions while allowing benign transactions to execute successfully. As the tenth and eleventh columns of Table 2 shows, patches can block all malicious transactions but may abort some benign transactions by mistakes. Since the inaccuracy of patches generated by Aroc is reflected in that the patches contain incomplete path constraints, all successfully-generated patches contain correct path constraints and secure rules, as introduced in the analysis of patch generation correctness. Consequently, patches may pose false positives but not false negatives. All in all, patches can block all malicious transactions with a few false positives, achieving approximately a correctness ratio of 93.32% for all the selected datasets, which is computed by the following equation.

$$\begin{aligned} Correctness &= \frac{\#patches - FR - FP - FN}{\#patches - FR} \\ &= \frac{TP}{\#patches - FR}. \end{aligned} \tag{2}$$

Correctness of Preventing New Reentrancy Patterns. To further evaluate the effectiveness and correctness of Aroc, we verify whether Aroc can block three new reentrancy patterns, which are cross-function, create-based, and delegated reentrancy, as proposed by Sereum [10]. Smart contracts have multi-entrances because each public function is a contract entrance. All the functions of a contract share the same contract state variables. Based on these features, to launch the cross-function reentrancy, attackers pose the inconsistent states by invoking different victim functions in a carefully constructed way. For delegated reentrancy, a library is delegate-call by a victim contract function. The attack leverages those libraries that share the same runtime contexts as the contract. Therefore, executions in libraries may cause inconsistent state-update, causing the victim function to be re-invoked in obsolete states. Created-based reentrancy is similar to the delegated reentrancy. When a victim function creates a new contract that invoke other malicious contracts, attackers can re-enter the victim.

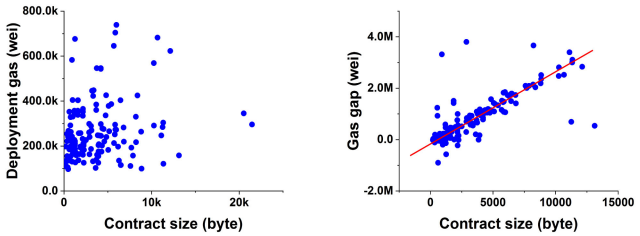
Aroc successfully generates correct patches for all test cases provided by Sereum. We then deploy victim contracts, and the attack contracts and patches. After depositing several ethers into the victim contracts, we launch attacks. The experimental results show that Aroc can successfully block malicious transactions for created-based and delegated reentrancy, but cannot prevent cross-function reentrancy. It is because that for the first two patterns, their re-entered functions are the same as the initial victim functions. Therefore, Aroc can block the malicious transactions for them similar to the common reentrancy vulnerabilities. However, for the cross-function pattern, the re-entered function is different from the initial victim function.

Unfortunately, our repair templates require that the signatures of the patch functions and the vulnerable functions are same. In such cases, patch functions can directly parse transaction data towards the vulnerable functions and verify whether the transactions are secure. Therefore, patches generated by Aroc cannot detect transactions towards functions that have different function signatures from patch functions. Besides, to reduce impacts on the vulnerable contract's state variables, we use shadow variables in the patches to represent the original storage variables in the vulnerable contracts. Although we have changed the shadow storage variables and verified them in patches, the vulnerable contract state variables do not change until the reentrancy process terminates, and thus the executions in the re-entered function can pass. Therefore, Aroc cannot repair cross-function reentrancy but is able to repair the remaining two kinds of reentrancy patterns.

6.1.2 Gas Costs

Many vulnerabilities in the EVMPatch dataset are repeated. Therefore, in this evaluation, we ignore the contracts containing identical bugs to improve the evaluation efficiency. Specifically, we conduct evaluations on all those unique contracts whose vulnerabilities can be attacked and patches can be generated. We consider two parts to evaluate the gas costs: patch deployment and execution. By deploying patches through remix, we can directly obtain their gas costs from the remix console. The remix is a popular ethereum contract IDE, which can compile, deploy, and debug contracts [26]. Moreover, to investigate whether the patch deployment gases have relationships with the contract complexity, the contract size is computed from runtime bytecodes to represent its complexity. Fig. 4 presents the experimental results. Specifically, Fig. 4a shows the deployment costs of patches corresponding to the different sizes of vulnerable contracts. Fig. 4b shows the deployment costs gap between the vulnerable contracts and patches.

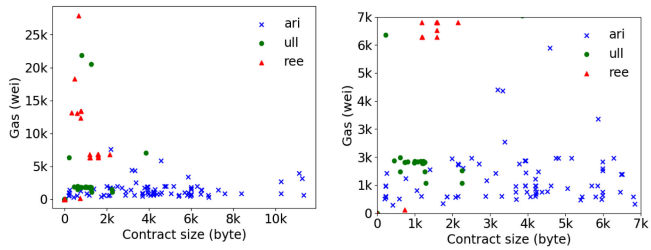
As Fig. 4a illustrates, patches of larger contracts do not necessarily cost more deployment gases. It is because patches mainly contain state variables declarations and several conditional expressions (i.e., path constraints and secure rules), and most operations of the vulnerable contracts are irrelevant to the generation of patches. Further, since several small contracts contain multiple bugs, their patches cost



(a) Patches deployment costs

(b) The deployment costs gap between the generated patches and related vulnerable contracts

Fig. 4. Deployment costs evaluations of patches for vulnerable contracts of different sizes.



(a) All dataset with contract size ranging from 0 to 15k bytes

(b) Partial dataset with contract size ranging from 0 to 7k bytes

Fig. 5. Execution gas costs evaluations of patches for vulnerable contracts of different sizes.

more gases in deployment. If these vulnerabilities exist in different functions, the patch contract will have multiple patch functions. The patches' size will be larger, thus requiring more deployment gases. Moreover, it is almost negligible even for the upper limit of patches deployment costs, which 8,000,000 wei ($\approx \$6.22 \times 10^{-5}$, computed from average gas price ($0.21/27Gwei) showing in etherscan [25].

When the deployed contracts contain vulnerabilities, users can fix the vulnerable contracts off-chain based on source codes, and then redeploy them. To study whether our proposal is more cost-efficient than this common strategy, we evaluate the gap of the deployment gas between patches and vulnerable contracts. This gap is the difference between the deployment costs of the vulnerable contracts and the associated patches.

Fig. 4b shows evaluation results. The value of most gas gap is higher than zero. That is, the original vulnerable contracts cost more deployment gases than that of the related patches in general. It implies that our strategy is more cost-efficient comparing with directly repairing vulnerable contracts off-line. Moreover, contracts directly repaired usually will be instrumented with more guarded rules, which are always larger than the original vulnerable contracts. Therefore, they may cost more deployment gases than our generated patches. Besides, as the trend line shows, directly repaired contracts will cost more deployment gases as the contract size grows. Thus, patches generated by Aroc have more outstanding advantages for large contracts.

However, the gas gap of several contracts is less than zero as shown in Fig. 4b, which means that these contracts' patches cost more deployment gases than them. The size of most of these contracts is less than 2,500 bytes as Fig. 4b shows. Because patches need to add new statements such as (11-15/3) in the reentrancy repair template, these additional operations may cause that patches are bigger than original contracts when vulnerable contracts are too small. But such cases are rare because lines of these contracts are within 10, which is too small to realize most real-world requirements. Several contracts with nearly 5,000 bytes codes, also cost fewer deployment gases than our patches. It is because that certain simple contracts have multiple bugs, and thus the size of patches is large than the original contracts. However, such cases are rare as the figure shows. In general, the gas consumption of deploying patches is in an acceptable range. In most cases, generated patches are more cost-efficient than directly repairing vulnerable contracts.

As for the overheads of contract executions, we invoke contracts using remix [26], and the execution costs will be computed and shown in the transaction details of the remix console. Specifically, we use the same test cases to call contracts before and after patching to obtain the additional runtime gases introduced by the patches. For the reentrancy vulnerabilities, their patches need to access the storage variables such as "balances" that may cost many gases because the storage operations are expensive in EVM. To avoid out of gas exceptions, we temporarily cancel the gas consumption of their patch executions in EVM, and obtain their gas consumptions via printing the associated logs in EVM. Meanwhile, since patches of the unchecked low-level call dataset cost sharply different for various called external functions, we cannot compute an average execution cost. But due to the few operations in patches (see Listing 6), patches cost fewer gases than the original contracts.

Finally, we only conduct evaluations for arithmetic and reentrancy patches. For the arithmetic dataset, the gas overheads that patches introduce are approximately 939 Wei and 1434.37 Wei respectively for smartBugs and EVMPatch dataset. Since there are only several condition expressions to verify transactions in the function body of arithmetic patches, they cost a few overheads even the contracts are large. Meanwhile, because contracts in the EVMPatch dataset are usually large than that in the smartBugs dataset, the corresponding patches are also larger. Therefore, patches for the EVMpatch dataset cost more. For the reentrancy dataset, the average gas overheads of patches are 10062.59 Wei ($\approx \$7.83 \times 10^{-8}$). Since reentrancy patches involve accessing storage variables repeatedly and the read and write operations on stored variables are expensive in EVM, small contracts with reentrancy will cost more execution gases. Nevertheless, the overheads are nearly ignorable considering the gas price and the amount of the additional gases.

To give a clear view on the execution overheads of patches, we draw a picture of all recorded results as shown in Fig. 5a. Since the size of the most contracts is less than 7,000 bytes, we present a figure on the partial dataset, whose size is less than 7,000 bytes, as shown in Fig. 5b to make the results more claer. In this picture, we exclude several experimental results on testing contracts with unchecked low-level call bugs. Because the invoking functions of these unconsidered contracts cost too many gases, and results cannot be plotted into the same picture with the remaining results. As the figure shows, the execution overheads of patches do not increase as the contract size increases. It is because Aroc will

TABLE 3
Comparisons Between ContractGuard[17] and Aroc With Real-Life Reported Vulnerabilities

Program	Vulnerability	Code size (bytes)	Deployment overhead		Runtime overhead		Recall		False alarms	
			CG	Aroc	CG	Aroc	CG	Aroc	CG	Aroc
DAO	Reentrancy[19]	1774	11.95%	49.83%	20.51%	41.26%	100%	100%	4.48	0
MultiSig	Unchecked low-level call [27]	764	29.58%	33.50%	28.27%	-	100%	-	2.0	-
KoETH	Unchecked low-level call[21]	3617	19.62%	20.22%	23.82%	2.36%	100%	100%	13.57	0
BecToken	Overflow[28]	5963	9.52%	5.72%	20.51%	1.77%	100%	100%	8.2	0
OwlWallet	Tx.origin authentication [27]	561	27.27%	72.19%	26.50%	7.17%	100%	100%	3.0	0
MerdeToken	Underflow[27]	1013	21.03%	56.03%	18.74%	2.24%	100%	100%	4.0	0

try to reduce the patches' size by integrating path constraints. Moreover, except for the cases that the low-level called functions are exorbitant, reentrancy patches have the highest overhead, second by unchecked low-level calls, and arithmetic vulnerabilities are the cheapest. It is because that reentrancy patches usually contain frequent state variables access, and low-level call patches often involve function calls, while arithmetic patches usually only have several judgment statements.

6.1.3 Comparisons With Existing Work

In this subsection, we compare Aroc with two recent state-of-the-art work, ContractGuard [17] and EVMPatch [16]. ContractGuard provides an on-chain smart contract protection approach by embedding safe paths into contracts and identifying whether executions obey safe paths. EVMPatch [16] can patch contracts automatically by rewriting off-chain contracts to upgradable contracts. Since ContractGuard and EVMpatch are not open-sourced, we directly compare Aroc with their provided results in their papers, namely, Table 2 of [17] and Table 3 of [16]. That is, we do not conduct experiments on them, but evaluate Aroc on the datasets provided by them to obtain the values of related metrics.

Table 3 presents the comparison results between ContractGuard and Aroc. The column *'Code size'* means the runtime code size of the vulnerable contracts, *'Deployment overhead'* is calculated by dividing the additional deployment gases by the original deployment gases, *'Runtime overhead'* is derived by the additional execution gases divided by that of vulnerable contract, *'Recall'* means the percentage of test cases where malicious transactions are successfully blocked, *'False alarms"* means the percentage of test cases where benign transactions are falsely blocked, and *'CG'* represents ContractGuard. The six contracts presented in this table are projects. Although we have not handled the Tx.origin bug in the *OwlWallet* contract, we patch them following the repairing rules of unchecked low-level calls.

For most contracts, Aroc introduces more deployment overheads than ContractGuard, since Aroc needs to redeclare global variables and vulnerable functions for creating patches while ContractGuard inserts safe paths into the original contracts to conduct verification. When the vulnerable contracts are simple, the size of variable and function declarations are smaller than that of safe paths. But for large contracts, Aroc may perform better. For example, for 'BecToken' with the largest code size, ContractGuard costs more deployment gases than Aroc. Since the large contracts are usually more complicated, safe paths that ContractGuard needs to insert more complicated and larger.

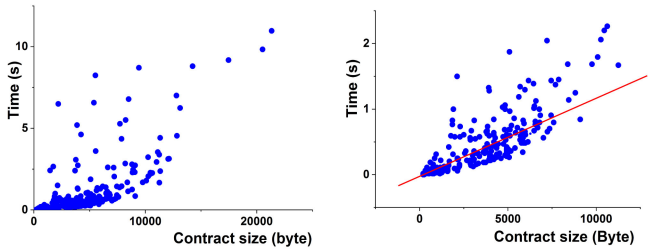
For the runtime overheads, Aroc always costs less than ContractGuard, except for the 'DAO' program with a reentrancy bug. It is because all secure rules of arithmetic bugs are simple judgment statements, thus consuming a few gases; the rules of the unchecked call only involve a judgment statement on the function call, which are also not expensive. However, the security rules of reentrancy bugs include manipulations on storage variables, whose related instructions are expensive as described in the ethereum yellow paper [18]. Therefore, the patch for the 'DAO' program takes more execution overheads. However, Aroc-generated patches can block all malicious transactions in the ContractGuard-provided dataset. We cannot evaluate Aroc on the 'MultiSig' program on-chain. Because its target line is in a fallback function that does not have a function name, we cannot bind it with the patch based on our binding strategy.

Table 4 presents the comparison results between EVMPatch and Aroc. *'Overhead'* and *'Code size increase'* refer to the additional execution consumptions and codes introduced by patches respectively. All contracts in this table include CVEs, whose IDs are shown in the *'CVE'* column. As the table illustrates, Aroc introduces less codes than EVMPatch for 'UET', 'SCA', and 'HXG', but more for 'BEC' and 'SMT' contracts. It is because EVMPatch requires duplicating certain codes of vulnerable contracts and inserting new fixed codes.

Moreover, the size of the duplicating codes increases as the number of vulnerable lines grows and the depth of bugs locations is deeper. Both the 'BEC' and 'SMT' contracts have only one bug, and the bugs are at the first basic block of their functions' CFG. Consequently, EVMPatch costs fewer for the two contracts. For the remaining contracts, the basic blocks that their vulnerable lines locate are deeper. Besides, UET and SCA have multiple bug lines. Therefore, EVMpatch costs more gases. It is noted that EVMPatch needs to re-deploy the repaired vulnerable contracts, but our patches only include several judgment statements. Aroc may cost less gases than experimental results when deploying patches.

TABLE 4
Comparisons Between EVMPatch[16] and Aroc With Arithmetic CVEs

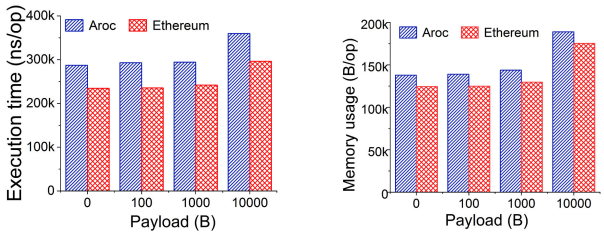
Program	CVE	Overhead (gas)		Code size increase (B)	
		EVMPatch	Aroc	EVMPatch	Aroc
BEC	2018-10299	83	596	117 (1.0%)	310 (2.65%)
SMT	20148-10376	47	479	191 (0.8%)	314 (1.32%)
UET	2018-10468	225	695	1,299(18.2%)	905 (12.68%)
SCA	2018-10706	47	7,740	3,811(1.77%)	696 (0.32%)
HXG	2018-11239	120	1,250	997 (28.10%)	334 (9.41%)



(a) Experimental results on completed dataset whose contract size ranges from 0 to 25000

(b) Experimental results on partial dataset whose contract size ranges from 0 to 15000

Fig. 6. Time of generating patches varies with the contract size.



(a) Execution time (b) Memory usage

Fig. 7. Comparisons of normal transfer transaction overheads between *Aroc* and *Ethereum*.

However, Aroc costs more runtime overheads than EVMPatch. Aroc integrates all repair rules into one patch contract. To protect contracts, the patch needs to be invoked. However, EVMPatch directly inlines secure arithmetic operations into the vulnerable contracts bytecodes. Hence, Aroc introduces more execution gases. However, as the gas price is low ($0.21/27 Gwei as mentioned above), the largest overhead (7,740 gas of 'SCA') is only around $\$6.02 \times 10^{-17}$, which is nearly negligible.

6.2 Performance of Aroc

Since patches should be deployed timely to avoid bugs being exploited, the speed of generating patches should be fast. Moreover, to support patch binding and transaction verification, we add operations in EVM. It is crucial to measure additional overheads introduced by these operations. Thus, next we will assess Aroc's performance from two parts: the time of generating patches and additional overheads on EVM.

6.2.1 Time of Generating Patches

Here, we run off-chain patch synthesis system of Aroc implemented in C++. Based on the `gettimeofday`, a function provided by C programming language to get current time, we instrument the off-chain system to derive the start and end time of generating patches. The difference between them is the patch generation time. Fig. 6 shows the relationship between the patch generation time and contract complexity, where the x-axis represents the contract size, and the y-axis represents the patch generation time. In the figure each dot represents a test result, meaning a tested smart contract's patch generation time. Since the size of most tested contracts are clustered within 15,000 bytes as Fig. 6a shows, to show rules clearly, Fig. 6b presents results on contracts whose size ranges from 0 to 15,000.

As Fig. 6 shows, the generation time increases as the contract size increases in general. This is reasonable because larger contracts need more time to be traversed and analyzed. Nevertheless, for most contracts, the generation time is less than 2.5 second (s), which is fast. Compared with the block generation time (15s) in ethereum, this time is acceptable. Besides, we also noted that several patches of small contracts take more generation time than large ones. Because these contracts contain more bugs, it takes more time for Aroc to analyze bugs and related path constraints. In particular, for contracts in the EVMPatch dataset, most of them have no less than one bug and some even have six bugs. However, the generation time of most patches is less than 10s. This demonstrates that Aroc enjoys the obvious advantage compared with manual repair, considering the generation time.

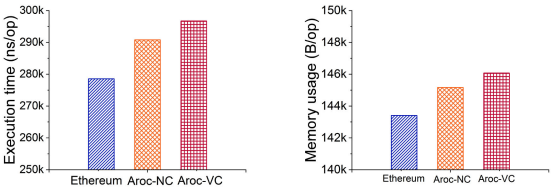
6.2.2 Performance of Enhanced EVM

In this experiment, we use the built-in benchmark to measure additional overheads on our enhanced EVM compared with raw EVM. The benchmark test results show the average time and memory allocation for executing test function once. New operations in the enhanced EVM focus on special transaction and verification on transactions that the receiver is not nil (i.e., common asset transfer transactions and contract call transactions). Since different transaction types have distinctly different proportions in transactions, for example, special transactions are sent only when contracts need repairing, hence we evaluate the overheads of special transactions, normal asset transfer transactions, and contract call transactions, respectively.

Since special transactions need to access the blockchain ledger when binding contracts, they introduce about 1.74 ms longer time and 0.73 MB higher memory overheads than transactions of raw ethereum. Nevertheless, users only send special transactions when patching their vulnerable contracts, which accounts for a tiny part of the blockchain transactions. Compared with the loss brought by vulnerability attacks, the overheads are in an acceptable range.

Then we evaluate normal transfer transaction overheads with different payload sizes ranging from 0 B, 100 B, 1000 B, and 10000 B. As Fig. 7 shows, Aroc needs more execution time and memory allocation. This is as expected because Aroc introduces additional transaction verification operations, for example, checking whether they are special transactions or contract call transactions. Moreover, as with the increase of payloads, overheads of both Aroc and ethereum grow. But clearly that the gaps of both execution time and memory usage between them do not become wider. Approximately, Aroc introduces a mean 22.47% additional time overhead and a mean 10.20% additional storage overhead.

As for contract call transactions, Aroc judges whether the receiver contract is vulnerable. If vulnerable, Aroc enforces that the receiver first delegate calls its related patches; otherwise, the transactions can be executed directly. So in our enhanced EVM, normal contract call and vulnerable contract call have different overheads. Hence, we compare contract call in raw EVM with *Normal Contract* call (NC) and *Vulnerable Contract* call (VC) in Aroc respectively. As Fig. 8 illustrates, both NC and VC cost more time and memory than the raw



(a) Execution time (b) Memory usage

Fig. 8. Comparison of contract call overheads (execution time and storage usage) with different sizes of payloads between *Aroc* and *Ethereum*.

contract call since Aroc introduces more operations to handle contract call transactions. Moreover, VC costs more overheads than NC because vulnerable contract calls are first delegated to patches and verified. In a summary, comparing with the original EVM, NC and VC introduce an average of 4.40% and 6.52% additional time overheads and an average of 1.23% and 1.86% additional storage overheads respectively.

7 Discussion

Although we only realize repair for three bug types in this paper, Aroc can play more roles in the contract ecology environment. First, Aroc has a good extensibility. Since three modules in Aroc are loosely-coupled, IEM can extract path constraints for other bug types. As long as more repair templates are placed into PSM, Aroc can repair more bugs. For instance, Aroc can also easily be extended to repair specific access control vulnerabilities such as verifying `tx.origin` and missing restrictions on sensitive operations. Specifically, suppose that Aroc finds `tx.origin` in the given target line. In that case, Aroc can get another operand in the comparison statement and add a comparison statement between the operand and `msg.sender` in the patch. For the second vulnerable case, Aroc can add a new `require` statements with `msg.sender` and the owner as operands, namely, `require (msg.sender == owner)`. Moreover, Aroc can deal with more access control vulnerable cases if users interact with it. For example, users can provide a whitelist or blacklist to help Aroc enhance access control of patches.

Moreover, our enhanced ethereum can strengthen deployed contracts. For example, to upgrade an on-chain contract, the user can develop an additional contract with more functions. Then he can bind it with the original contract by sending a special transaction. Although the existing contract upgrade model can use a new one to replace the old contract, but this method also has several limitations. For example, it needs to redeploy a contract with all functions, which will spend more costs comparing with Aroc. Moreover, considering the tamper-proofing feature of blockchain, Aroc may be an excellent supplement for this way.

Unfortunately, Aroc cannot repair all contracts yet since it gets state variables value of vulnerable contracts based on the `delegatecall` principles. But the delegatecallee can only fetch the delegatecalller's context, there is no way to obtain the context of the third contract. If the target vulnerable line depends on variables in the third contract, our generated patches cannot protect it. Besides, Aroc cannot fix vulnerabilities in `fallback` functions since it does not have function signatures, which is hard to identify transactions towards them. Moreover, to prevent malicious manipulations, Aroc requires the deployers of the vulnerable contracts to send special transactions to bind patches with the vulnerable contracts. However, smart contracts can also be deployed by contracts. Then when the deployer of a vulnerable contract is also a contract, the binding process cannot be completed because contracts cannot send transactions.

8 Related Work

Off-Chain Vulnerability Detection. Until now, most researches focus on static vulnerability detection through static analysis [6], [29], [30], fuzzing [31], [32], [33], [34], formal verification [35], [36] etc. Oyente [29] first constructs a CFG and then symbolically analyzes it to get program traces, which are leveraged to detect vulnerabilities based on predefined unsafe patterns. Osiris [6] aims at detecting integer bugs based on symbolic execution and taint analysis. Ethainter [30] can identify composite vulnerabilities that can only be exploited by a successive transactions. However, tools depending on static program analysis may not be precise enough since the blockchain execution environment is not realistically simulated. ContractFuzzer [31] fuzzes smart contracts based on randomly generated test cases in a simulated blockchain environment.

To further improve path coverage, Sfuzz [32] tries to improve the effectiveness of test cases based on the distance-based heuristic strategy. Meanwhile, EthPloit [34] can generate directed test cases towards dangerous asset transfer instructions based on taint analysis. Ethploit creates transaction sequences from the sink to sources according to variables dependence graph that taint analysis generates, so these test cases are more efficient. Echidna [33] randomly generates a sequence of transactions towards the given smart contract, and checks whether executions violate a set of given invarients. Focusing on verifying programs' safety, formal verification has also attracted attentions in smart contract field. VeriSmart [35] verifies arithmetic bugs based on transaction invariants and loop invariants. Verx [36] can automatically verify whether smart contracts have realized required functional requirements through a series of formal verification technologies. These tools are orthogonal to our proposal, and they focus on off-chain audit while our system aims at offering runtime protection.

On-Chain Vulnerability Protection. In addition to on-chain countermeasures introduced in Section 1, Soda [37] also provides a way to detect vulnerabilities on chain. Soda instruments EVM to extract transaction runtime information, and detects vulnerabilities based on checking these information against test oracles. Solythesis [38] provides a new compiler to support runtime verifications by instrumenting smart contracts with user-specified invariants. Any transaction violating the invariants will be aborted. SMACS [39] leverages a off-chain service consisting of various verification tools and *Access Control Rules* (ACRs) to verify transactions. ContractGuard [17] implements an intrusion detection system, which first collects initial safe path sets based on user-provided passable test cases in the training phase, then instruments the sets to the contract. Transactions whose execution paths are not in the safe sets will trigger alarms. Then administrator will reproduce this transaction off-chain and manually verify its security. EVM* [40] instruments EVM to

monitor transaction executions and stop dangerous transactions in real-time. EVMPatch [16] leverages the bytecode rewriting technology to generate patches automatically, then uses delegatecall-proxy technology to repair vulnerable contracts by replacing the implementation address with the address of the new repaired contract. Our system proposes a new approach to protect on-chain smart contracts without modifying them. Besides, our system and these tools are complementary to each other and can be integrated to create a securer contract ecosystem.

9 Conclusion

Smart contract vulnerabilities are the core obstacle for the development of blockchain, and a small and simple loophole may even cause colossal asset losses. Providing comprehensive countermeasures for guarding contract security is of great significance. However, most existing solutions focus on off-chain solutions, which are inapplicable to deployed vulnerable contracts. In this paper, we offer an automatic smart contract repair framework without the requirement to modify source codes by combining off-chain static program analysis and on-chain dynamical exploit prevention. Experiments show that Aroc can automatically generate patches for most vulnerable smart contracts. Moreover, these patches are precise enough to block most malicious transactions while only falsely blocking limited number of benign transactions. Further, Aroc is able to generate a patch within seconds, and the additional overheads on ethereum are also acceptable. Compared with existing outstanding studies, Aroc usually introduces less execution overheads than ContractGuard and less contract sizes than EVMPatch.

In our future work, we will improve Aroc to support repairing more kinds of vulnerabilities. Moreover, we will consider leveraging backward symbolic execution technologies to generate path constraints, and generate more sound patches without false negatives.

References

[1] S. Zhong and X. Huang, "Special focus on security and privacy in blockchain-based applications," *Sci. China Inf. Sci.*, vol. 63, no. 3, 2020, Art. no. 130100.

[2] H. Huang, X. Chen, and J. Wang, "Blockchain-based multiple groups data sharing with anonymity and traceability," *Sci. China Inf. Sci.*, vol. 63, no. 3, 2020, Art. no. 130101.

[3] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, "Ethereum smart contract security research: Survey and future research opportunities," *Front. Comput. Sci.*, vol. 15, 2021, Art. no. 152802.

[4] Slowmist, 2018. [Online]. Available: https://slowmist.com

[5] S. Zone, "Slowmist hacked," 2020. [Online]. Available: https://hacked.slowmist.io/

[6] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in Ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.

[7] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[8] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 363–373.

[9] Y. Feng, E. Torlak, and R. Bodik, "Summary-based symbolic evaluation for smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 1141–1152.

[10] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proc. 26th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2019.

[11] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels, "Enter the Hydra: Towards principled bug bounties and exploit-resistant smart contracts," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1335–1352.

[12] Contract upgrade anti-patterns, 2018. [Online]. Available: https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/

[13] Solidity, 2020. [Online]. Available: https://docs.soliditylang.org

[14] N. Group, "Decentralized application security project (or DASP) top 10 of 2018," 2018. [Online]. Available: https://www.dasp.co/

[15] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 530–541.

[16] M. Rodler, W. Li, G. O. Karame, and L. Davi, "EVMPatch: Timely and automated patching of Ethereum smart contracts," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1289–1306.

[17] X. Wang, J. He, Z. Xie, G. Zhao, and S. Cheung, "ContractGuard: Defend Ethereum smart contracts with embedded intrusion detection," *IEEE Trans. Service Comput.*, vol. 13, no. 2, pp. 314–328, Mar./Apr. 2020.

[18] G. Wood, "Ethereum yellow paper," 2018. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[19] D. Siegel, "Understanding the dao attack," 2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[20] OpenZeppelin, "openzeppelin-contracts," 2020. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts

[21] Post-mortem investigation, 2016. [Online]. Available: https://www.kingoftheether.com/postmortem.html

[22] C. Peng, S. Akca, and A. Rajan, "SIF: A framework for solidity contract instrumentation and analysis," in *Proc. 26th Asia-Pacific Softw. Eng. Conf.*, 2019, pp. 466–473.

[23] J. F. Ferreira and P. Cruz, "SB curated: A curated dataset of vulnerable solidity smart contracts," 2020. [Online]. Available: https://github.com/smartbugs/smartbugs/tree/master/dataset

[24] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: A framework to analyze solidity smart contracts," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 1349–1352.

[25] Ethereum, "The Ethereum blockchain explorer," 2015. [Online]. Available: https://cn.etherscan.com/

[26] Ethereum, "Remix, Ethereum-IDE," 2020. [Online]. Available: https://remix-ide.readthedocs.io/en/latest/

[27] Merdetoken, 2017. [Online]. Available: https://github.com/Arachnid/uscc/tree/master/submissions-2017/doughoyte

[28] B. Mueller, "Detecting integer arithmetic bugs in Ethereum smart contracts," 2016. [Online]. Available: https://media.consensys.net

[29] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[30] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2020, pp. 454–469.

[31] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 259–269.

[32] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 778–788.

[33] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 557–560.

[34] Q. Zhang, Y. Wang, J. Li, and S. Ma, "EthPloit: From fuzzing to efficient exploit generation against smart contracts," in *Proc. 27th IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2020, pp. 116–126.

[35] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for Ethereum smart contracts," in *Proc. IEEE Symp. Security Privacy*, 2020, pp. 1678–1694.

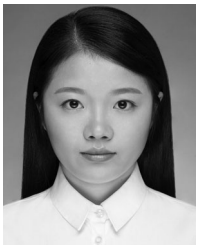
[36] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. T. Vechev, "VerX: Safety verification of smart contracts," in *Proc. IEEE Symp. Security Privacy*, 2020, pp. 1661–1677.

[37] T. Chen *et al.*, "SODA: A generic online detection framework for smart contracts," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020.

[38] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proc. 41st ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2020, pp. 438–453.

[39] B. Liu, S. Sun, and P. Szalachowski, "SMACS: Smart contract access control service," in *Proc. 50th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2020, pp. 221–232.
[40] F. Ma *et al.*, "EVM: From offline detection to online reinforcement for Ethereum virtual machine," in *Proc. 26th IEEE Int. Conf. Softw. Anal.*, 2019, pp. 554–558.

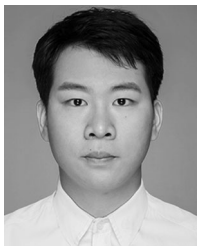
Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology, China, in 1994. He is currently a chair professor of computer science and engineering at the Huazhong University of Science and Technology (HUST), China. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz, Germany. He worked at The University of Hong Kong, Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California, Los Angeles, California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China, in 2001. He is a fellow of CCF, and a life member of the ACM. He has coauthored more than 20 books and published more than 900 research papers. His research interests include computer architecture, parallel and distributed computing, big data processing, data storage, and system security.



Zeli Wang is currently working toward the PhD degree in the School of Computer Science and Technology, Huazhong University of Science and Technology (HUST), China. Her research interest is blockchain smart contract security.

Ming Wen received the PhD degree from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology (HKUST), Hong Kong. He is currently an associate professor at the School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China. His research interests include program analysis, fault localization and repair, and software security. His research work has been regularly published in top conferences and journals in the research communities of software engineering, including ICSE, FSE, ASE, TSE and EMSE and so on. For more information: https://justinwm.github.io.

Weiqi Dai received the PhD degree in computer science and technology from the Huazhong University of Science and Technology (HUST), China. He is currently an associate professor at the School of Cyber Science and Engineering, Huazhong University of Science and Technology, China. His research interests include blockchain, data privacy, cloud computing security, trusted computing, and virtualization technology.



Yu Zhu is currently working toward the PhD degree in the School of Cyber Science and Engineering, Huazhong University of Science and Technology (HUST), China. His research interests include security in smart contract and blockchain.

Deqing Zou received the PhD degree from the Huazhong University of Science and Technology (HUST), China, in 2004. He is currently a professor at the School of Cyber Science and Engineering, Huazhong University of Science and Technology, China. He has been the leader of one 863 Project of China and three National Natural Science Foundation of China (NSFC) projects, and a core member of several important national projects, such as the National 973 Basic Research Program of China. He has applied almost 20 patents, published two books, one is Xen virtualization Technologies and the other is Trusted Computing Technologies and Principles, and published more than 50 high-quality papers. His research interests include system security, trusted computing, virtualization, and cloud security. He has served as the PC member/PC chair of more than 40 international conferences.