Automated Software Engineering (2023) 30:31
https://doi.org/10.1007/s10515-023-00392-y

Tips: towards automating patch suggestion for vulnerable smart contracts

Qianguo Chen[1] · Teng Zhou[1] · Kui Liu[1] · Li Li[2] · Chunpeng Ge[1] · Zhe Liu[1] · Jacques Klein[3] · Tegawendé F. Bissyandé[3]

Received: 19 March 2022 / Accepted: 25 June 2023

Abstract
Smart contracts are slowly penetrating our society where they are leveraged to support critical business transactions of which financial stakes are high. Smart contract programming is, however, in its infancy, and many failures due to programming defects exploited by malicious attackers and have made the headlines. In recent years, there has been an increasing effort in the literature to identify such vulnerabilities early in smart contracts to reduce the threats to the security of the accounts. Automatically patching smart contracts, however, is a much less investigated research topic. Yet, it can provide tools to help developers in fixing known vulnerabilities more rapidly. In this paper, we propose to review smart contract vulnerabilities and specify templates that will serve to automate patch generation. We implement the TIPS pipeline with 12 fix templates and assess its effectiveness on established smart contract datasets such as SmartBugs and ContractDefects. In particular, we show that TIPS is competitive against the state-of-the-art automated repair approach (SCRepair) in the literature. Finally, we evaluate the impact of the code changes suggested by TIPS in terms of gas usage.

Keywords Smart contract vulnerability · Automated repair · Fix template

1 Introduction

"*Even the wisest man occasionally falls prey.*"

Blockchain technology has attracted great interests from the research and industry in the last decade due to its brought opportunities and challenges w.r.t. anonymity, trust, immutability, etc. Beyond the largely covered application of crypto-currencies, there are increasingly other applications that are sought-after, such as decentralized databases and smart contracts. The latter are increasingly

Extended author information available on the last page of the article

Published online: 13 September 2023

deployed to blockchain and realize automated transactions in a traceable, transparent, and irreversible way. Smart contracts are programmed in a specific language, the most used being the Solidity programming language, which is supported by the Ethereum platform (Ethereum 2021).

Just like traditional programs, smart contract programs include defects that can make them vulnerable to attacks. In the context of smart contracts, however, the involved financial stakes entice attackers to be diligent in finding and exploiting vulnerabilities. Because of the huge currency transactions, the vulnerable smart contracts have been keenly perceived by attackers. In June 2016, a reentrancy vulnerability in the smart contract of the DAO project (for an investment fund) was exploited in an attack that led investors to lose 3.5 million Ether (over 60 million US dollars at that time) (del Castillo 2016). A year later, a critical code flaw on the Parity Ethereum client software resulted in the freezing of $160 million worth of Ether (O'Leary 2017). These examples are not isolated. A recent survey by Wan et al. (2021) revealed that 40% of respondents admit facing vulnerability problems with their smart contracts.

To address vulnerabilities in smart contracts, various approaches have been proposed in the literature (Luu et al. 2016; Liu et al. 2018b; Jiang et al. 2018a; Li et al. 2019; Gao et al. 2020; Brent et al. 2020). While these approaches provide automated support for exposing vulnerabilities (i.e., what kinds of vulnerabilities and their locations), fixing such vulnerabilities still requires substantial manual effort. Indeed, smart contract programming is still not mature and developers therefore often require help that is not yet available in online forums: at the time of writing, among the ~2100 questions on stack-overflow about Solidity smart contract programming, half of them were not answered.

With the momentum in automated program repair (APR), the community is now exploring approaches for repairing smart contracts. Recently, Yu et al. (2020) proposed to investigate genetic programming search for the repair of smart contracts. Nguyen et al. (2021) leveraged symbolic execution and run-time information of smart contracts towards fixing four kinds of common vulnerabilities.

This paper. Our work investigates the possibility of fixing vulnerabilities with a classical approach in the repair of programs written in general programming languages: *template-based patch generation*. Templates have enabled recent approaches such as FixMiner (Koyuncu et al. 2020) to produce state-of-the-art results in the literature for Java programs. In the APR literature, template-based approaches often serve as a baseline (Liu et al. 2019c) to discuss the improvements that sophisticated approaches can bring. Therefore we explore an approach where smart contract code represented at the abstract syntax tree (AST) level is manipulated based on match and transform fix templates towards suggesting patches for smart contracts. Our approach is inspired by the findings of Wan et al. (2021): smart contract developers frequently reuse code from reliable sources in smart contract development to address their vulnerabilities. Therefore, our insight is that experts' understanding (Chen et al. 2020; DASP 2021; Ethereum Smart Contract Security Best Practices 2021) of smart contract vulnerabilities can form a reliable source for deriving fix templates. The TIPS approach, therefore, aims at characterizing and instantiating actionable fix templates for a variety of well-known vulnerabilities in the literature. Overall, we manage to take into account in our extensible TIPS finally eight types of smart contract vulnerabilities, which are addressed by 12 crafted fix templates.

This paper makes the following contributions:

- We systematically analyze concrete vulnerable smart contract samples in online blogs (DASP 2021; Ethereum Smart Contract Security Best Practices 2021) and overview literature (Chen et al. 2020) on smart contract defects to characterize the repair actions that are applied to fix them. Subsequently, we summarize the repair templates.
- We develop TIPS: a template-based smart contract repair framework, where match and transform operations at the AST level are encoded, finally taking into account 12 repair templates for 8 common vulnerability types.
- We perform a series of experimental validations of TIPS. We apply TIPS on samples from the SmartBugs*curated* (Durieux et al. 2020), ContractDefects (Chen et al. 2020), and SCRepair (Yu et al. 2020) datasets. For our experiments, we use well-known tools such as Slither and Mythril for defect detection. Overall, TIPS exhibits excellent performance metrics not only in terms of repairability but also in terms of effectiveness.

2 Background

This section presents background information about smart contract vulnerabilities and automated program repair.

2.1 Smart contract vulnerabilities

The practical execution of smart contracts could be threatened by the vulnerabilities from three aspects: the blockchain system, the virtual machine of the executing platform, and the smart contract programs themselves. This work focuses on the vulnerabilities in smart contract programs. Vulnerabilities in smart contract programs can be caused by flaws that are inherent to the used programming language or by programming mistakes from smart contract developers. We further circumscribe our study on the latter cases.

Figure 1 presents an example, in a single function, of the well-advertised reentrancy vulnerability. In the vulnerable code written in the Solidity programming language, the user's balance `userBalances` is not set to "`0`" until the very end of the function (i.e., line 7). When the caller's code at line 5 is executed by the attacker, it can call the `withdrawBalance` function repeatedly in the fallback[1] function of the attacker's contract before the first invocation of the `withdrawBalance` function is finished. Thus, eventually, the attacker is able to receive more tokens while

[1] A fallback function in Solidity is executed when the function identifier does not match any of the available functions in a smart contract or if there was no data supplied at all.

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
  uint amountToWithdraw = userBalances[msg.sender];
  (bool success, ) = msg.sender.call.value(amountToWithdraw)
    ("");
  require(success);
  userBalances[msg.sender] = 0;
}
```

Fig. 1 Example of code fragment with the reentrancy vulnerability in a single function

his/her balance is not changed until the execution runs out of gas (see following) or the call stack limit is reached. In the given example, the best way to prevent this attack is to make sure that any external function will be called only after all the needed internal work is done.

Due to the immutability of Blockchain, once a smart contract program is deployed, it can no longer be modified. Discovering and fixing smart contract vulnerabilities early is therefore a major concern in smart contract programming. Towards coping with this concern, practitioners regularly characterize and publicly share knowledge about smart contract vulnerabilities that Solidity programmers must be aware of. Following up on the success of the OWASP (Open Web App Security Project), DASP (Decentralized Application Security Project), an open and collaborative project, has released the first enumeration of its top-10 smart contract vulnerabilities in 2010, providing hints to how to resolve them (DASP 2021). There are also smart contract solution providers or auditing firms which, just like anti-virus providers, showcase in their blogs some sampled discovered vulnerabilities to demonstrate their expertise. Such one firm[2] is maintaining an online forum (Ethereum Smart Contract Security Best Practices 2021) maintains a blog on “Ethereum Smart Contract Best Practices” where some known smart contract attacks (as well as proposed solutions) are discussed. In the peer-reviewed literature, Chen et al. (2020) have recently presented the findings of their empirical study on the characteristics of smart contract defects based on a deep dive into smart-contract-related posts within Ethereum StackExchange[3] and real-world smart contracts. Their study summarized 20 types of smart contract defects that were validated based on practitioners’ opinions. Overall this study provides a rich source of information on smart contract defects based on the practitioner’s understanding.

[2] ConsenSys Diligence - https://consensys.net/diligence/.

[3] https://ethereum.stackexchange.com.

2.2 Automated program repair

Automated program repair (APR) (Goues et al. 2019; Monperrus 2018; Gazzola et al. 2017) is a research field that develops techniques for automatically generating corrective changes (on source code Liu et al. 2020 or bytecode Ghanbari et al. 2019) to address bugs in programs. It holds the ambition and promise of alleviating the manual effort involved in program debugging (Liu et al. 2021) by reducing the time-to-fix delays and the downtime caused by program bugs. Since smart contracts must be repaired in development settings before they are deployed on immutable blockchain networks, their repair should be performed on source code.

In general APR, the basic requirement of repair is that the generated patch should fix the targeted bug **and** must not introduce any new issue. Smart contract repair has similar constraints. We consider the following validation criteria as minimal: the generated patches can resolve the related vulnerability and do not introduce new (known) vulnerabilities. Furthermore, it should be noted that the normal execution of smart contracts on the widely-used Ethereum blockchain platform involves gas consumption: it refers to the fee or pricing value required to successfully conduct a transaction or execute a contract on the platform. Therefore, an extra (but important) validation criterion for smart contract patches is to take potential changes in the gas consumption by the execution of the patched smart contract into consideration (Yu et al. 2020).

In the community of automated program repair, various automated program repair approaches have been studied, which can be summarized into four categories: heuristic-based approaches, constraint-based approaches, learning-based approaches, and template-based approaches (Liu et al. 2021). In 2013, Kim et al. (2013) found that fix templates summarized from human-written patches can be used to fix program bugs automatically, which built a milestone of automatically fixing bugs with fix templates. Since then, various automated program repair tools based on fix templates have been proposed (Koyuncu et al. 2019, 2020; Liu et al. 2019b, c; Le et al. 2016; Saha et al. 2017, 2019; Wen et al. 2018; Jiang et al. 2018b; Yuan and Banzhaf 2018). Liu et al. (2020) systematically investigated the efficiency of different automated program repair approaches, of which results show that template-based APR approaches present overwhelming performances than heuristic-based APR approaches and constraint-based ones. More recently, Liu et al. (2022) also reported that templated-based APR tools can achieve comparing results against learning-based APR tools relying on a big number of bug-fixing data for training, templated-based APR tools even can fix some bugs that cannot be solved by learning-based APR tools. In addition, fix templates can help practitioners understand the code issue of bugs and the related fixing behavior at the level of code (Liu et al. 2019c). Therefore, in this work, we propose to investigate the possibility of fixing smart contract vulnerabilities with fix templates. To this end, we summarized fix templates from practitioners' knowledge shared in the community.

3 Towards a fix template taxonomy

For this study, we systematically review the community (cf. the third paragraph in Sect. 2.1) to identify approaches that leverage fix patterns to carefully understande code samples available in the literature and datasets (cf. Sect. 2). With the characteristics of smart contract vulnerabilities and the given potential solutions that are released in the community by experts (Chen et al. 2020; DASP 2021; Ethereum Smart Contract Security Best Practices 2021), we manually summarized 12 fix templates of eight common vulnerability categories with respect to the vulnerable code of Solidity smart contracts. Note that, we reference the knowledge of smart contract defects defined by Chen et al. (2020) to summarize fix templates for smart contract vulnerabilities. In addition, it is worth mentioning that these templates are not always correct, but their correctness is further verified in experiments in Sect. 5. Below, we initiate a fix template taxonomy with 12 fix templates (FT) described in terms of simplified GNU diff format for easy understanding, which is the first contribution of this work.

FT-1: Fixing unchecked external calls. Inserting a Boolean value check for the unchecked external calls in smart contracts, or replacing the external calls with the function `address.transfer()`.

```
FT-1.1 Inserting call checking:
- address.external_calls(ethers);
+ if(!address.external_calls(ethers)) { throw; }

FT-1.2 Replacing the external call:
- address.external_calls(ethers);
+ address.transfer(ethers));
```

where "`external_calls`" represents the unchecked external calls, i.e., `address.send()`, `address.call()` and `address.delegatecall()`. In Solidity smart contracts, the external calls for raw addresses could fail to execute because of network errors, out-of-gas error, etc. The call that fails during execution returns a Boolean value (i.e., `false`) without handling any exception. If their return values are not checked, the correctness of the executing code logic cannot be guaranteed. When they return `false` value, while the remaining code will still be executed, which would lead to unexpected outcomes. For example, when the code at line 3 in Fig. 2 returns the boolean `false` value, line 4 will still be executed and the value of the variable `valuePaid` will be changed. Therefore, the fix template **FT-1.1** is to check the return value of the external calls, and throw an exception when the return value is `false`, while the other fix template **FT-1.2** tries to replace the external calls with the `transfer` function, which will throw an exception while failing to execute.

```
if(valuePaid > currentClaimPrice) {
  uint excessPaid  = valuePaid - currentClaimPrice;
  msg.sender.send(excessPaid);
  valuePaid = valuePaid - excessPaid;
}
```

Fig. 2 Example of the unchecked external call vulnerability (https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol)

FT-2: Fixing reentrancy. Replacing the vulnerable reentrancy call with the function `send` or `transfer` in smart contracts, or moving the statements of changing state variables to the position before external calls.

```
FT-2.1 Replacing the reentrancy call:
- address.call.value(ethers)
+ address.send/transfer(ethers)

FT-2.2 Moving the statement with state variables:
+ statements with state variables;
 ...msg.sender.call.value(ethers)...
- statements with state variables;
```

A reentrancy attack happens when the external contract calls can make new calls to the called contract before completing the initial execution. It means that the state of the contract may change in the middle of the function call due to an untrusted external call that leads to the repeated balance withdrawal. Therefore, the fix template **FT-2.1** uses the function `send` or `transfer` to make a transfer within the limited gas cost. In such a case, the attack can be interrupted because of the gas limitation. The fix template **FT-2.2** prevents this attack by setting contract states changes (like the user's balance here) before the Ether transfer to make sure that an external function won't be called until all needed internal work is done.

FT-3: Fixing access control. The access control vulnerability can be introduced by three kinds of issues that should be addressed in three different ways: (1) Replacing the authenticating valuable `tx.origin` with `msg.sender`, (2) Replacing the incorrect function name with the corresponding constructor function, and (3) Inserting the missing protection before the caller accesses the high authority.

```
contract MyContract {
    address owner;
    function MyContract() public {
        owner = msg.sender;
    }
    function sendTo(address receiver, uint amount) public {
        require(tx.origin == owner);
        receiver.transfer(amount);
    }
}
```

Fig. 3 Example of the tx.origin access control vulnerability (https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-txorigin)

```
FT-3.1 Replacing the authenticating variable:
- tx.origin == owner
+ msg.sender == owner

FT-3.2 Replacing the incorrect constructor:
- otherFuncName()
+ constructor/contractName()

FT-3.3 Inserting the missing protection:
+ require(msg.sender == contractOwner);
   Authority-sensitive statements
```

In smart contracts, the execution of some code requires a the high level of autorization through private or internal functions. While the straightforward way for attackers to dominate the owner's account and balance is opened due to the negligence of authenticating the insecure visibility. When contracts use the deprecated `tx.origin` (a global variable in the smart contract that represents the original address the transaction initiate) to validate callers, it will make the contracts suffer from phishing attacks like Fig. 3. If the attacker invokes the function `sendTo` of `MyContract` in his contract, the owner of `MyContract` could be phished to transfer Ether to the attacker until exhausting the balance in `MyContract` account. Thus, the fix template **FT-3.1** prevents this attack by replacing the vulnerable `tx.origin` with `msg.sender`.

In smart contracts, the constructor function can be used to initialize the global variables. The correct constructor will be executed when the contract is deployed on the blockchain and cannot be called anyway in the next life cycle. In some versions of Solidity, developers could be recklessly misled by the function names that are similar to constructors. It will degenerate the constructor into an externally callable function, which could be attacked to take control of the contract account by calling such a fake constructor function. In Fig. 4, the function `missing` is to initialize the state variable *owner*. Actually, this function is an external function that can

```
contract Missing{
    address private owner;
    ...
    function missing() public {
        owner = msg.sender;
    }
    ...
}
```

Fig. 4 Example of the access control vulnerability (https://smartcontractsecurity.github.io/SWC-registry/docs/SWC-118#incorrect-constructor-name1sol)

```
contract SimpleSuicide {
  function sudicideAnyone() {
    selfdestruct(msg.sender);
  }
}
```

Fig. 5 Example of the lack of protection (https://github.com/SmartContractSecurity/SWC-registry/blob/master/test_cases/unprotected_critical_functions/simple_suicide.sol)

be invoked and the value of *owner* can be modified by everyone. So, the fix template **FT-3.2** fixes the attack by replacing the wrong-used function with the correct constructor.

In smart contracts, code statements could include the authority-sensitive operations of which execution requires that the contract owner is the executor. Once developers neglect the needed protection, it will make the contract vulnerable. The contract shown in Fig. 5 can be destroyed by anyone since it is lack of protection. Thus, the fix template **FT-3.3** adds the missing protection to validate that the `msg.sender` is the contract owner `contractOwner`.

FT-4: Fixing arithmetic issue. Inserting the missing protection before/after the arithmetic operation.

Arithmetic overflows and underflows are dangerous in smart contracts, where unsigned integers are prevalent and most developers usually use simple uint types. If overflows occur, many benign-seeming codepaths become vectors for theft or denial of service, such as the careless programming BEC[4] token with the arithmetic overflow vulnerability. To resolve the arithmetic overflow/underflow issues, the fix templates add the related protection for each specific arithmetic operation.

[4] https://etherscan.io/address/0xc5d105e63711398af9bbff092d4b6769c82f793d#code

```
Inserting the missing protection:
 a = b + c;
+ require(a >= b || a >= c);
OR
+ require(b >= c);
a = b - c;
OR
a = b * c;
+ require(c = a / b);
OR
a += b;
+ require(a >= b);
OR
+ require(a >= b);
a -= b;
OR
+ uint tmp = a;
a *= b;
+ require(b = a / tmp);
```

FT-5: Fixing strict balance equality. Replacing the strict equal comparison of balance value with the range check for the related variable.

```
Replacing the authenticating variable:
- this.balance == val
+ this.balance >= val && this.balance < val + 1
```

Normally, developers are used to writing the strict balance equality (i.e., `this.balance == var`) when the contract code requires checking the balance value. It could be attacked by forcibly sending Ethers to the contract with the selfdestruct (victim address) instruction.[5] In this case, the fallback function will not be triggered, and the victim contract cannot reject the Ethers. Therefore, the logic of checking equal balance will fail to work due to the unexpected ethers send by attackers, which leads the code in the logic will not be executed. The fix template replaces the strict balance equality with the range check (inner one Ether) of the balance.

FT-6: Fixing unmatched type assignment. Replacing the int-related data type with `uint` or `uint256`.

[5] http://solidity.readthedocs.io

```
Replacing the int type:
- for(uintN i = 0; i < val; i++)
+ for(uint/uint256 i = 0; i < val; i++)
```

Solidity has different types for integer numbers (e.g. uint8, uint256). The integer type supports a smaller ranger, it takes less memory. However, when an integer value exceeds the maximum value of the related integer type, Solidity will lead to overflow without throwing any exception that will affect the normal execution of smart contracts. Checking the range of integer value will increase the gas assumption, the fix template simply replaces the integer type with `uint` or `uint256`.

FT-7: Inserting a suicide function. This fix template is designed for the **missing interrupter** and the **greedy issue** in smart contracts.

```
Inserting the missing interrupter:
+ function suicide() {
+   require(contractOwner == msg.sender);
+   selfdestruct(this.balance);
+ }
```

If a vulnerable contract is deployed in the blockchain, the developers cannot modify the code anymore, but watch the attacking until out-of-gas when it is attacked. To reduce the economic losses, developers can find a way to stop these contracts and deploy a new one. The fix template adds a suicide function for such contracts. With such a suicide function, ethers on the contracts can be withdrawn and the contracts are destroyed when attacks happen with the missing interrupter of the greedy issue.

A contract can withdraw Ethers by sending Ethers to another address or using self-destruct function. Without these withdraw-related functions, Ethers in contracts can never be withdrawn and will be locked forever. Such contracts are defined as greedy contracts (Chen et al. 2020). One example of greedy contracts is Parity contract (O'Leary 2017), after the Parity library contracts are killed, the wallet contracts could no longer access the library. Finally this defect resulted in the locking of 200 M dollars indefinitely. The fix template adds a `suicide` function to prevent ether locking.

FT-8: Fixing hard coded address. Removing the hard coded addresses and setting the addresses as function parameters.

```
Making the hard coded address(es) as the parameter(s) of a function
- function method() {
-   ...0xeeeeeeeee...;
+ function method(param0) {
+   ...param0...;
```

```
- function hardencode() {
-   address dest = 0xeeeeeeeeeeeeeeeeeeee;
+ function hardencode(address param0) {
+    address dest = param0;
     ...
  }
```

Fig. 6 Example of a hard encode address and the related fix

The security and scalability of smart contracts could be impacted by the address variables hard coded in them. For example, if an address variable is an illegal address hard coded by a mistake, it can cause that the amount of the contract cannot be withdrawn by anyone. For the other example, if an address variable is a hard coded suicide address, the Ether will be lost forever when others send Ether to the suicide contracts. Thus, the fix template removes the hard coded address and use function parameters to replace the addresses, so makes the hard coded addresses flexible (Fig. 6).

4 Setup for smart contract repair

To investigate the effectiveness of our fix templates set (from the initial taxonomy presented in Sect. 3), we design experiments of fixing smart contract vulnerabilities using the fix templates. The produced smart contract repair tool, TIPS, is then assessed on three datasets of smart contract vulnerabilities to allow the feasible investigation of repairing smart contract and the reliable comparison against the state-of-the-art smart contract repair tools.

4.1 TIPS: a baseline smart contract repair tool

We implement a smart contract repair pipeline TIPS with the identified fix templates. Given a vulnerable smart contract, TIPS leverages vulnerability detection tools (i.e., Mythril Mueller 2018 and Slither Feist et al. 2019) to identify the vulnerability and its position in the code. TIPS generates patches for vulnerable smart contracts by modifying code at the AST level. To this end, the smart contract will be parsed into AST before the patch generation with the solidity compiler solc (Solidity 2021) in nodejs. It then selects the adequate fix template with the identified vulnerability category identified by vulnerability detection tools, and heuristically generates patches with the code change actions specified by the selected fix template. Finally, the generated patch is validated with the compiling process, vulnerability detection tools, and manual checking.

Patch generation: To generate patch candidates for fixing the detected vulnerabilities, TIPS leverages the information provided by the related detection tools and code change actions of fix templates to guide the automated process of synthesizing patch candidates. More specifically, given a vulnerability detected by the detection tools, TIPS can automatically select the adequate fix template for it with its category identified mainly by Slither Feist et al. (2019) and the related vulnerability

```
1 contract Token {
2    address owner;
3    ...
4    function suicide() {
5 +      require(owner == msg.sender);
6       selfdesturct(this.balance);
7    }
8 }
```

Fig. 7 Example of fixing the access control vulnerability

information supplemented by Mythril Mueller (2018) appropriately. If there are several fix templates defined for one vulnerability category (i.e., FT-1, FT-2, FT-3, and FT-4), we follow the defined sort of those templates in Sect. 3 to generate patch candidates for the given vulnerability one by one. In other words, for the FT-1 vulnerability, TIPS will first generate patch candidates with FT-1.1, then with FT-1.2.

Note that, the positions of vulnerabilities are defected at the function level, TIPS thus automatically figures out the vulnerable code with the context information of the vulnerability specified in the related fix template in a heuristic way. Once the vulnerable code is identified, and then heuristically modifies the vulnerable code with the code change actions to generate patches. It follows the prior implementations done in general template-based program repair (Liu et al. 2019b, c). As shown in Sect. 3, for fixing some vulnerabilities, TIPS needs the adequate donor code to synthesize the patch. For example, with FT3.3 and FT8, TIPS needs the donor code (i.e., a variable of the contract owner) to synthesize the patch. TIPS searches all variables from the contract that has the same context with the required donor code (i.e., a state variable). These variables will be prioritized based on the semantic similarity between their names and the required context. For FT3.3 and FT8, variables named with “owner”, “creator” or “admin” will be prioritized over other variables. If TIPS fails to find the related variable, it will randomly use an *address* variable. For example, Fig. 7 shows a vulnerability fixed by TIPS. The code at Line 5 is generated by heuristically finding the *owner* variable from the *Token*’s state variable.

Patch validation: In the normal program repair, patches are validated with the regression test. However, the smart contracts in the dataset do not contain such tests. To validate the patches for vulnerable smart contracts, each patch is first validated with Remix (Remix 2021), an open-sourced Solidity smart contract IDE. If the patched contract cannot be successfully compiled by Remix, it is considered as a failure case. When the patched contract passes the compiling process, we further leverage Mythril (Mueller 2018) and Slither (Feist et al. 2019) to check whether any above vulnerability can be detected in it. Mythril relies on analysis techniques such as concolic analysis and taint analysis to detect vulnerabilities. It will search for the real values that can exploit vulnerabilities to reduce the false positive of the detected vulnerabilities. Slither is a static analysis framework that uses various detectors to detect different types of vulnerabilities. Durieux et al.’s empirical study shows that the combination of Mythril and Slither is the best trade-off between accuracy and execution costs (Durieux et al. 2020). Thus, Mythril and Slither are selected to validate the vulnerable smart contracts patched by TIPS. Besides these two techniques, we also consider a machine learning based tool SmartEmbed (Gao et al. 2020) to validate the patched smart contracts. Finally, the patched smart contracts without any detected vulnerabilities are further validated by authors manually. With the aforementioned steps, if a correct patch is generated, other patch candidates will not validated any more. In the practical process of automated program repair (Xiong et al. 2017), once a correct patch is generated, the automated program repair will stop generating and validating any other patch candidates. Therefore, in this work, we follow it for the patch validation of fixing smart contract vulnerabilities.

4.2 Environment and dataset

Our experiment was carried on Ubuntu 18.04 with 32GB memory and four cores. TIPS is implemented using Python 3.6 and nodejs 12.18. In the experiments of this work, three datasets of vulnerable smart contracts (i.e., SmartBugs^{curated}Durieux et al. 2020, ContractDefects Chen et al. 2020, and the contracts used by SCRepair Yu et al. 2020) are used to assess the patch suggestion ability of TIPS. Table 1 illustrates the number of vulnerable smart contracts and vulnerabilities in each dataset that are used in the evaluation of this study. For more information about the three datasets, please reference the related works (Durieux et al. 2017; Chen et al. 2020; Yu et al. 2020).

SmartBugs^{curated} is a dataset of vulnerable smart contracts collected from Github repositories (e.g., not-so-smart-contracts Smart Contracts 2021), SWC Registry (SWC-registry 2021), and the blog such as Positive.com (ICO Security 2021) that analyzes the contract and the Ethereum network, which has been used to empirically review the automated analysis tools for smart contracts (Durieux et al. 2020). ContractDefects contains 20 kinds of smart contract defects defined in Chen et al. (2020), which stresses smart contract security and emphasizes the availability, performance, maintainability and reusability of smart contracts. The two datasets contain the vulnerabilities related to smart contracts. TIPS however focuses on resolving vulnerable Solidity code of smart contracts (i.e., excluding others related to system, etc.). Then, after also removing redundant vulnerability cases, 148 vulnerabilities in 113 smart contracts are selected from SmartBugs^{curated} and 147 vulnerabilities in 54 smart contracts are selected from ContractDefects for the experiments in this study. From the evaluation dataset of the SCRepair state of the art smart contract repair tool, we considered all the 48 vulnerabilities in 17 smart contracts.

5 Assessment

5.1 Patch suggestion ability of TIPS

Our first experiment focuses on assessing the ability to suggest patches for vulnerable smart contracts with TIPS, which is conducted on the vulnerable smart contracts

Table 1 Dataset information

Vul type	# Contracts	# Vul
SmartBugs^{curated} (*Durieux et al.* 2020)		
Unchecked external call	52	75
Reentrancy	31	32
Access control	16	18
Arithmetic	15	23
Total	113*	148
ContractDefects (*Chen et al.* 2020)		
Unchecked external call	12	28
Reentrancy	4	10
Strict balance equality	4	4
Unmatched type assignment	20	38
Missing interrupter	41	41
Hard coded address	12	20
Greedy	6	6
Total	54*	147
SCRepair dataset (*Yu et al.* 2020)		
ED (Unchecked external call)	14	28
Reentrancy	5	6
IO (Arithmetic)	3	12
Transaction order dependency	2	2
Total	17*	48

*The total number of smart contracts is not equal to the sum of smart contracts in the corresponding column since some smart contracts contains different types of vulnerabilities. “vul”: vulnerability

of dataset SmartBugs^{curated} (Durieux et al. 2020) and ContractDefects (Chen et al. 2020). The patch suggestion results are shown in Table 2.

In this experiment, we found that 148 vulnerabilities (5+2 *Unchecked external call* vulnerabilities, 16 *Access control* vulnerabilities, all of the *Arithmetic*, *Missing interrupter*, *Unmatched type assignment*, and *Hardcoded address* vulnerabilities) from 64 smart contracts cannot be detected by Mythril or Slither. To assess the effectiveness of resolving these vulnerabilities for the related fix templates, we manually check all smart contracts and identify the positions of these 148 vulnerabilities. When only considering the detectable vulnerabilities, TIPS suggests correct patches for 95.4% (=104/109) vulnerable smart contracts and 96% (=144/150) vulnerabilities. If considering all vulnerabilities, 89% (=154/173) vulnerable smart contracts and 88.1% (=260/295) vulnerabilities are successfully suggested with correct patches. Additionally, none of the patched smart contracts were checked with the above vulnerabilities by Mythril, Slither, and SmartEmbed. These results suggest that the most vulnerable smart contracts can be suggested with correct patches by TIPS.

Table 2 Number of smart contracts and vulnerabilities in SmartBugs^{curated} and ContractDefects fixed by TIPS

Dataset	Defect type	# Contracts	# Vul
SB^{curated}	Unchecked external call	51/52	69/70
	Unchecked external call*	5/5	5/5
	Reentrancy	29/31	29/32
	Access control	2/2	2/2
	Access control*	12/14	14/16
	Arithmetic*	12/15	19/23
ContractDefects	Unchecked external call	11/11	26/26
	Unchecked external call*	2/2	2/2
	Reentrancy	3/4	9/10
	Strict Balance Equality	4/4	4/4
	Unmatched type assignment*	14/20	28/38
	Greedy	5/6	5/6
	Missing interrupter*	37/41	37/41
	Hard coded address*	9/12	11/20
Total	detectable only detectable & Non-detectable	104/109 154/173*	144/150 260/295*

'SB' represents SmartBugs. 'x/y' represents the number of fixed/total smart contracts and vulnerabilities

*The corresponding vulnerabilities cannot be detected by Mythril and Slither, so their positions are manually identified by the authors of this paper

We further assess to what extent vulnerabilities can be fixed by each fix template in TIPS. Experimental results are presented in Table 3. Three fix templates (FT-3.1, FT-3.2, FT-5) can be used to fix all related vulnerabilities in the two datasets. FT-1.1 and FT-1.2 are two fix templates for the same vulnerable issue of *unchecked external call*, where FT-1.1 fixed more cases than FT-1.2. One *Unchecked external call* vulnerability is not fixed by FT-1.1 or FT-1.2 due to a character encoding problem. 29 vulnerabilities cannot be resolved by FT-1.2 because FT-1.2 cannot be applied to them. It implies that adding the missed checking (FT-1.1) is more actionable than replacing the call (FT-1.2) for *Unchecked external call* vulnerabilities.

For the *Reentrancy* vulnerabilities, FT-2.1 repaired 38 out of 42 vulnerabilities, while FT-2.2 only successfully repaired 12 of them. FT-2.1 relies on the function `send` or `transfer` to fix the *Reentrancy* vulnerable `call.value` function to make a transaction, it can limit the gas usage for the patched smart contract to avoid complex operations in the fallback functions. FT-2.2 moves the state statement with respect to the related state variables before the statement of transferring Ether. One vulnerability cannot be fixed because of the character encoding issue. Three unfixed *Reentrancy* vulnerabilities are out of the scope of the summarized fix templates. In addition, the *Reentrancy* vulnerable code of 29 vulnerabilities is not located with the same code block of the statement with respect to the related state variables, TIPS thus failed to find the statement with FT-2.2. So, FT-2.2 is not as actionable as FT-2.1 for the *Reentrancy* vulnerabilities.

```
contract Token {
    function num() public returns(uint8) {
-       for(uint8 i = 0; i < rounds.length; i++){
+        for(uint i = 0; i < rounds.length; i++){
            if((now > rounds[i].start) && (now <= rounds[i].end)){
                return i+1;
            }
        }
        return 0;
    }
}
```

Fig. 8 A patched smart contract that fails to compile

Table 3 Number of vulnerabilities in SmartBugs^{curated} and ContractDefects fixed by each fix template

Fix template	# Vul	Fix template	# Vul	Fix template	# Vul
FT-1.1	102/103	FT-3.2	6/6	FT-7-MI	37/41
FT-1.2	73/103	FT-3.3	9/11	FT-7-G	5/6
FT-2.1	38/42	FT-4	19/23	FT-8	11/20
FT-2.2	12/42	FT-5	4/4		
FT-3.1	2/2	FT-6	28/38		

‘FT-7-MI’ and ‘FT-7-G’ represent the fix template FT-7 working on *Missing Interrupter* and *Greedy* vulnerabilities, respectively

For the fix templates FT-3.3, FT-4 and FT-7, eight unfixed vulnerabilities (two access control vulnerabilities, four arithmetic issues, one greedy issue and one missing interrupter vulnerability) cannot be fixed by them as these fix templates indeed do not cover all cases of related vulnerabilities. The remaining three missing interrupter vulnerabilities cannot be correctly repaired due to the failed compile. The vulnerabilities patched with the fix template FT-7 are disappeared, but adding a suicide function for a smart contract is a dangerous operation. So more effective fixing approaches should be exploited for the corresponding vulnerabilities.

For the fix templates FT-6 and FT-8, eight *Unmatched type assignment* vulnerabilities and eight *Hard coded address* vulnerabilities cannot be suggested with correct patches by TIPS as these patched smart contracts are failed to be compiled, Different from other fix templates, FT-6 and FT-8 are arisen with the context-aware problem (the context information should be considered for the patch generation) when they are used to fix corresponding vulnerabilities. Two *Unmatched type assignment* vulnerabilities and one *Hard coded address* vulnerability are not fixed because of the context-aware problem. Figure 8 shows an example of unfixed *Unmatched type assignment* vulnerability due to the context-aware problem. The function `num` returns an `uint8` value of which data type is consistent with the data type of the variable `i` in the vulnerable `for` statement. The fix template FT-6 replaced the data type `uint8` of the variable `i` with `uint`, but ignored the return data type of the function `num`. It leads to the patched smart contract cannot pass the compilation process. We infer that the context-aware problem could be solved with a data-flow analysis for the context

```
1   contract Token {
2       address public owner;
3
4   -    constructor() public {
5   -        owner = 0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee;
6   +     constructor(address param0) public {
7   +         owner = param0;
8       }
9   }
10
11  contract readToken {
12      Token public token;
13      constructor() public {
14          token  = new Token();
15      }
16  }
```

Fig. 9 Context-aware problem in patched hard coded address vulnerability

information of the vulnerable code element before repairing the related vulnerabilities. For the other instance, Fig. 9 presents the *Hard coded address* vulnerability unfixed by TIPS. An address is hard coded to the address variable `owner` in the constructor of the smart contract. FT-8 generates a patch for it by refactoring the constructor with a parameter that is used to transfer the address variable. However, the original constructor is invoked at line 14, which leads the patched smart contract cannot pass the compilation process.

In brief: TIPS is effective in fixing most vulnerabilities in smart contracts with the fix templates summarized based on experts' understanding of smart contract vulnerabilities. TIPS presents the limitation of fix templates, the same as template-based program repair (Liu et al. 2019c), which cannot help solve the vulnerabilities that are not covered by them. In addition, TIPS suffers from the character encoding problem for repairing some vulnerable smart contracts. And TIPS ignores the analysis the context information of vulnerabilities for generating related patches. These shortages should be explored in the future for template-based smart contract repair to improve the performance of generating correct patches for vulnerable smart contracts.

5.2 Comparing against state-of-the-art

Our second experiment compares the patch suggestion ability of TIPS against the state-of-the-art smart contract repair tool, SCRepair (Yu et al. 2020). SCRepair fails to execute with the dataset SmartBugs^{curated} and ContractDefects. For a fair comparison, we conduct this experiment with the dataset (collected from Ethereum) used by SCRepair. To avoid potential bias related to the detection of vulnerabilities (Liu et al. 2019a), TIPS leverages the same tools as SCRepair to detect the vulnerabilities and their positions, namely Slither (Feist et al. 2019) and Oyente (Luu et al. 2016). We originally planed to perform regression testing

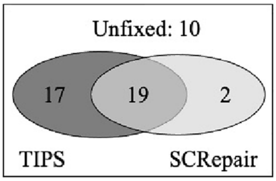

Fig. 10 Number of vulnerabilities fixed/unfixed by TIPS and SCRepair

Table 4 Results of vulnerable smart contracts fixed by SCRepair and TIPS

Contract Name	Vul type (# Vul)	# Vul fixed by SCRepair		# Vul fixed by TIPS	
Autonio ICO	ED(1)	ED(0)	✗	ED(0)	✗
Airdrop	ED(4)	ED(3)	(✓)	ED(4)	✓
Banana Coin	ED(1), RE(1)	ED(1), RE(1)	✓	ED(1), RE(1)	✓
XGold Coin	ED(2)	ED(2)	✓	ED(2)	✓
Hodbo Crowdsale	ED(2)	ED(2)	✓	ED(2)	✓
Lescoin Presale	ED(2)	ED(1)	(✓)	ED(2)	✓
Classy Coin	ED(1), RE(1)	ED(0), RE(0)	✗	ED(1), RE(1)	✓
Yobcoin Crowdsale	ED(2), RE(1)	ED(1), RE(1)	(✓)	ED(2), RE(0)	(✓)
Classy Coin Airdrop	ED(2)	ED(1)	(✓)	ED(2)	✓
OKO Token ICO	ED(4), RE(2)	ED(1), RE(1)	(✓)	ED(4), RE(0)	(✓)
ApplauseCash Crowdsale	ED(2), RE(1)	ED(1), RE(0)	(✓)	ED(2), RE(1)	✓
HDL Presale	ED(3)	ED(3)	✓	ED(3)	✓
Privatix Presale	ED(1)	ED(1)	✓	ED(1)	✓
MXToken Crowdsale	ED(1)	ED(1)	✓	ED(1)	✓
dgame	IO(3), TOD(1)	IO(0), TOD(0)	✗	IO(0), TOD(0)	✗
Easy Mine ICO	IO(6), TOD(1)	IO(0), TOD(0)	✗	IO(6), TOD(0)	(✓)
Siring Clock Auction	IO(3)	IO(0)	✗	IO(0)	✗
Total	ED(28), IO(12), RE(6), TOD(2) sum: 48	ED(**18**), IO(0), RE(3), TOD(0) sum: 21		ED(**27**), IO(**6**), RE(3), TOD(0) sum: 36	

*✗, (✓), and ✓ represent that the vulnerable contract is unfixed, partially fixed, and fully fixed, respectively. “ED”, “RE”, “IO” and “TOD” represent the exception disorder(Unchecked external call), reentrancy, integer overflow(arithmatic), and transaction order dependence vulnerabilities, respectively

to perform the correctness of the patch like SCRepair. However, SCRepair did not provide the corresponding regression test data, thus we just discuss the repair ability of both. The comparison results on fixing vulnerable smart contracts between SCRepair and TIPS are presented in Table 4.

Overall, SCRepair fixed 21 vulnerabilities while TIPS fixed 36 ones. The overlap between fix vulnerabilities is shown in Fig. 10. SCRepair fixed two vulnerabilities that TIPS cannot fix, but TIPS fixed 17 vulnerabilities that SCRepair cannot fix. SCRepair fully fixed six vulnerable smart contracts, while TIPS fully fixed 11 out of 17 vulnerable smart contracts where one smart contract and four smart contracts are unfixed/partially fixed by SCRepair. In addition, TIPS fixed six integer overflow vulnerabilities that are not fixed by SCRepair.

From the perspectives of fixed vulnerable smart contracts and vulnerabilities, TIPS outperforms the state-of-the-art smart contract repair tool, SCRepair.

We also closely check the vulnerabilities that cannot be fixed by TIPS. The *Reentrancy* and *Integer overflow* vulnerabilities are not fixed by TIPS since it does not have the related fix templates. Similarly, TIPS does not have any fix template for the transaction dependency vulnerabilities. It implies that experts' knowledge on resolving vulnerable smart contracts cannot catch up with the newly emerged vulnerabilities.

Some vulnerable smart contracts include different types of vulnerabilities in the same code line. To fix the *ClassyCoin* smart contract, TIPS faces conflicts among the fix templates in the generation of patches. We made different trials to solve this problem. Eventually, TIPS was able to generate correct patches by prioritizing the *reentrancy* vulnerability over the exception disorder vulnerability, which is used in the final experiment. This case study example suggests that the fixing order of different vulnerabilities should be prioritized when they are located in the same code line.

5.3 Efficiency of suggesting patches

5.3.1 Gas variation of patched smart contracts

Different from the bug repair of traditional programs, patching smart contracts should consider the gas usage (Yu et al. 2020). If the patched smart contract takes more gas than the original one, it might trigger the out-of-gas exception. Nevertheless, the gas of executing a smart contract is given by the initiator of a transaction. In theory, the patched smart contract should not spend too much gas causing the out-of-gas exception. Moreover, if the patched smart contract needs more gas, the initiator of a transaction will spend more fee to finish the task. Therefore, the patched smart contract should not cost too much gas compared with the original one. To assess the gas variation of the patches smart contract against the original one, we leverage the compiler solc, which can estimate the gas usage of a smart contract when it compiles the smart contract. The gas variation is then calculated with the following formula:

$$Gas\_Variation = \frac{Gas_{patched} - Gas_{origin}}{Gas_{origin}} \times 100\% \tag{1}$$

Table 5 presents results about the gas variation of 165 smart contracts from the three datasets (i.e., SmartBugs^{curated}, ContractDefects, and SCRepair_Dataset) fixed by TIPS. Overall, the gas variation of 138 (83.6%) patched smart contracts is negative or slightly positive (up to 5%) when compared with the vulnerable smart contracts: 55 (33.3%) of patches increase the gas usage between 0 and 1%, while 14 patches do not induce any gas consumption variation. Finally, we found that 33 patches led to the gas consumption decrease. The Mann-Whitney-Wilcoxon tests ($pvalue$ = 0.425 >0.05) confirm that the gas usage of patched smart contracts is not statistically significantly higher than the original ones, but strongly similar to each other.

These results definitively suggest that the smart contracts patched by TIPS will not spend much higher gas usage than the original ones.

5.3.2 Time cost of fixing

The time cost is always one criterion for evaluating the performance of automated program repair in the community. Smart contracts are tuning-complete simple programs without too much code. In theory, patch suggestions or automated repair of vulnerable smart contracts should not take too much time, though the developers have difficulty in debugging vulnerable smart contracts manually. We thus investigate the time cost of suggesting patches for vulnerable smart contracts with TIPS, of which results are illustrated in Fig. 11.

As presented in Fig. 11, the time cost of suggesting patches with TIPS as a whole is in proportion to the increased size (i.e., the number of characters) of smart contract programs. SmartBugs^{curated} dataset contains simpler smart contract programs than the other two datasets, most of them are suggested with correct patches by TIPS within 15 ms. Compared with SmartBugs^{curated}, the ContractDefects, and SCRepair_Dataset smart contracts have higher complexity, most of them can be suggested with correct patches with TIPS within 150 ms and 200 ms, respectively.

These results indicate that TIPS can suggest patches for vulnerable smart contracts in a short waiting time for developers.

6 Threats to validity

The external threats include the objective addressed by this study, which limits eight-category smart contract vulnerabilities at the source code level. The vulnerabilities of smart contracts at the blockchain are not considered in the scope of TIPS, and other vulnerabilities that existed in smart contract code are not covered by the eight categories. To alleviate this threat, this work thus considers the common smart contract vulnerabilities collected from the industry and studied in the community. The second external threat is from the smart contract vulnerability detection tool, which cannot ensure that each vulnerability can be correctly and precisely detected. To address this limitation, we consider two state-of-the-art smart contract vulnerability detection tools (Mythril and Slither) to identify vulnerabilities.

The first internal threat to validity is the fix templates summarized from experts' understanding of smart contract vulnerabilities, it highly relies on experts' knowledge. To address this threat, we systematically summarized the fix templates from one published TSE paper, which defines lots of smart contract defect patterns, and two forums of smart contract vulnerabilities that are publicly maintained and contributed by practitioners. The other internal threat is from the

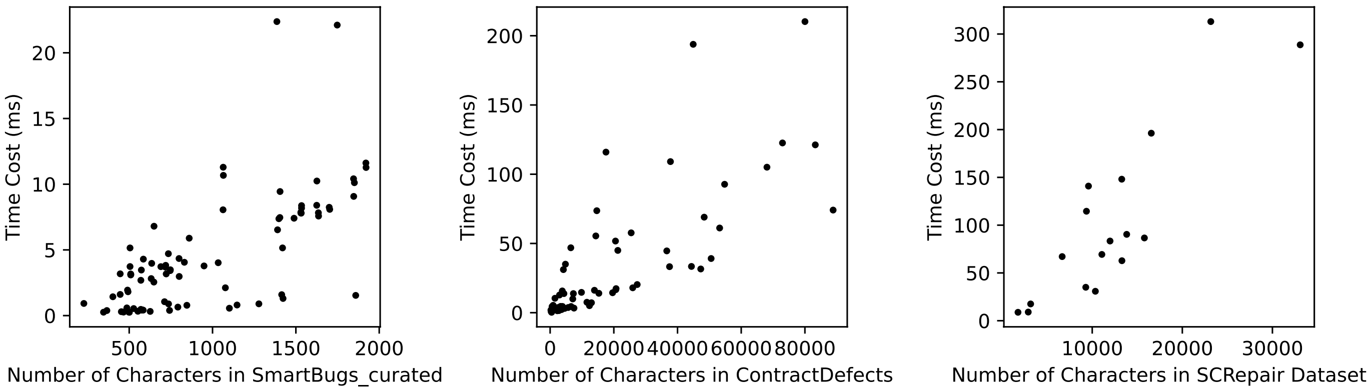

Fig. 11 Distribution of the time cost for fixing vulnerable smart contracts in three datasets with TIPS

Table 5 Gas variation for 165 patched smart contracts

Gas variation	# Smart contracts	Ratio
< 0%	33	20%
0%	14	8.5%
(0%, 1%]	55	33.3%
(1%, 2%]	21	12.7%
(2%, 3%]	5	3%
(3%, 4%]	6	3.6%
(4%, 5%]	4	2.4%
(5%, 6%]	5	3%
(6%, 7%]	2	1.2%
(7%, 8%]	2	1.2%
(8%, 9%]	6	3.6%
(9%, 10%]	0	0%
> 10%	12	7.3%

patch validation that is not conducted with regression tests since smart contracts lack the related test cases. To address this threat, the patches generated by TIPS are checked with several vulnerability detection tools and manually identified by authors. As future work, automated tests should be integrated into the patch validation process. The construct threat to validity is that the context information for patch generation is ignored by TIPS. To boost smart contract repair, contexts of vulnerabilities are considered as a future task to improve TIPS.

7 Related work

7.1 Detecting and fixing smart contract vulnerabilities

Smart contracts are written in a domain-specific programming language, which also face the same security problem as the programs written by traditional programming language. Because smart contracts are immutable once deployed and are related to economic transactions, a subtle defect may result in huge financial losses. Therefore it is significant to solve defects before releasing. To enhance the robustness of smart contracts, lots of approaches (Zhang et al. 2020a, b; Huang et al. 2020; Hartel and Schumi 2020; Ashraf et al. 2020) have been proposed to detect the vulnerabilities.

Oyente (Luu et al. 2016) is one of the first smart contract analysis tools. It uses symbolic execution on EVM bytecode to detect the vulnerabilities. Maian (Nikolić et al. 2018) and Osiris (Torres et al. 2018) borrowed its method to enhance the effectiveness of such a detection. SmartCheck (Tikhomirov et al. 2018) uses static analysis(lexical and syntactical analysis) to search the vulnerability patterns. Mythril (Mueller 2018), developed by ConsenSys, relies on concolic analysis, taint analysis, and control flow to check the EVM bytecode and look for values that can exploit vulnerabilities in the smart contracts. Jiang et al. (2018a) proposed to use fuzzing to build seed inputs within the valid input domain to test smart contracts. In this research line, tools like ReGuard (Liu et al. 2018a), Harvey (Wüstholz 2019) and sFuzz (Nguyen et al. 2020) are all proposed by leveraging the fuzzing approach to discover the vulnerabilities for smart contracts.

These vulnerability detection tools can easily obtain the vulnerability information of smart contracts. We can refer to the detection results of the above tools to attempt automated repairs. However, there are a few ways to directly fix these vulnerabilities automatically. SCRepair (Yu et al. 2020) is the first work of fixing vulnerable smart contracts with genetic programming search. Contemporary, Nguyen et al. (2021) leveraged the symbolic execution traces of the smart contract and specific fix patterns to repair four kinds of smart contract vulnerabilities. This work relies on the fix templates summarized from experts' knowledge of smart contract vulnerabilities to change the AST of smart contract code to generate the related patches for vulnerabilities.

7.2 Template-based program repair

In the literature, template-based techniques have been widely studied to automatically fix program bugs in traditional software, and achieved promising performance in relieving developers from the heavy burden of manual debugging. Kim et al. (2013) first proposed the automated program repair tool, PAR, with fix templates that are manually summarized from human-written patches. At the top of PAR, ELIXIR (Saha et al. 2017) is proposed with more fix templates based on its authors' knowledge about Java program bugs. NPEFix (Durieux et al. 2017) is also an automated program repair proposed with its authors' knowledge of null pointer exception-related Java bugs.

Except for the manual summarization, statistics on recurrent code changes at the AST level of bug fixes have been explored to automated program repair (e.g., CapGen Wen et al. 2018 and SimFix Jiang et al. 2018b). Long et al. (2017) proposed Genesis to automatically infer fix templates from human-written patches for repairing three kinds of bugs. Liu et al. (2019b) and Rolim et al. (2018) leveraged the fix patterns for static analysis bugs to automatically fix the semantic bugs failing to pass some test cases. Liu et al. (2018) explored to mine fix patterns from Q &A posts in Stack Overflow for program repair. Koyuncu et al. (2020) mined the fix templates at the AST level from patches. Liu et al. (2019c) systematically summarized fix templates from the literature to build a baseline program repair system TBar to boost program repair. In this work, we leverage the experts' knowledge to repair vulnerable smart contracts, thus improving the robustness of smart contracts before releasing them.

8 Conclusion

With the ambition of steering research towards ensuring the robustness of smart contracts, we propose a baseline approach for automatically patching vulnerable smart contracts. TIPS is a template-based system that employs an automatic program repair scenario that is now mature in the literature. TIPS leverages 12 fix templates (addressing 8 vulnerability types) that are summarized from experts' knowledge of smart contract vulnerabilities. The patch suggestion ability of TIPS is evaluated on three datasets (i.e., SmartBugs^{curated}, ContractDefects, and SCRepair dataset) of smart contract vulnerabilities collected from the real world. The experimental results show that the basic pipeline by TIPS can effectively generate patches for vulnerable smart contracts and outperform the state-of-the-art smart contract repair tool SCRepair. Our replication package of TIPS is publicly available at: https://github.com/CVbluecat/TIPS.

References

Ashraf, I., Ma, X., Jiang, B., Chan, W.K.: GasFuzzer: fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities. IEEE Access **8**, 99552–99564 (2020). https://doi.org/10.1109/ACCESS.2020.2995183

Brent, L., Grech, N., Lagouvardos, S., Scholz, B., Smaragdakis, Y.: Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, pp. 454–469. ACM (2020). https://doi.org/10.1145/3385412.3385990

Chen, J., Xia, X., Lo, D., Grundy, J., Luo, X., Chen, T.: Defining smart contract defects on ethereum. IEEE Trans. Softw. Eng. (2020)

del Castillo, M.: The DAO attacked: code issue leads to \$60 million ether theft (2016)

Durieux, T., Cornu, B., Seinturier, L., Monperrus, M.: Dynamic patch generation for null pointer exceptions using metaprogramming. In: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering, pp. 349–358 (2017)

Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 530–541 (2020)

Ethereum smart contract security best practices. https://consensys.github.io/smart-contract-best-practices/ (Last Accessed: July 2021)

Ethereum. https://ethereum.org/ (Last Accessed: July 2021)

Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8–15 (2019)

Gao, Z., Jiang, L., Xia, X., Lo, D., Grundy, J.C.: Checking smart contracts with structural code embedding. IEEE Trans. Softw. Eng. 1–1 (2020)

Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. IEEE Trans. Softw. Eng. **45**(1), 34–67 (2017)

Ghanbari, A., Benton, S., Zhang, L.: Practical program repair via bytecode mutation. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 19–30. ACM (2019). https://doi.org/10.1145/3293882.3330559

Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)

Hartel, P.H., Schumi, R.: Mutation testing of smart contracts at scale. In: Proceedings of the 14th International Conference on Tests and Proofs. Lecture Notes in Computer Science, vol. 12165, pp. 23–42 (2020). https://doi.org/10.1007/978-3-030-50995-8_2

Huang, Y., Jiang, B., Chan, W.K.: EOSFuzze: fuzzing EOSIO smart contracts for vulnerability detection. CoRR (2020) arXiv:2007.14903

ICO Security. https://blog.positive.com (Last Accessed: July 2021)

Jiang, B., Liu, Y., Chan, W.K.: ContractFuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 259–269. ACM (2018a). https://doi.org/10.1145/3238147.3238177

Jiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 298–309. ACM, (2018b)

Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 802–811 (2013)

Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Monperrus, M., Klein, J., Le Traon, Y.: ifixr: bug report driven program repair. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 314–325 (2019)

Koyuncu, A., Liu, K., Bissyandé, T.F., Kim, D., Monperrus, M., Klein, J., Le Traon, Y.: FixMiner: mining relevant fix patterns for automated program repair. Empir. Softw. Eng. **25**(3), 1980–2024 (2020). https://doi.org/10.1007/s10664-019-09780-z

Le, X.B.D., Lo, D., Le Goues, C.: History driven program repair. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 213–224. IEEE (2016)

Li, Z., Wu, H., Xu, J., Wang, X., Zhang, L., Chen, Z.: Musc: a tool for mutation testing of ethereum smart contract. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, pp. 1198–1201. IEEE (2019). https://doi.org/10.1109/ASE.2019.00136

Liu, X., Zhong, H.: Mining stackoverflow for program repair. In: Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, pp. 118–129 (2018)

Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B.: ReGuard: finding reentrancy bugs in smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp. 65–68. ACM, (2018a). https://doi.org/10.1145/3183440.3183495

Liu, H., Liu, C., Zhao, W., Jiang, Y., Sun, J.: S-gram: towards semantic-aware security auditing for ethereum smart contracts. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 814–819. ACM (2018b). https://doi.org/10.1145/3238147.3240728

Liu, K., Koyuncu, A., Bissyandé, T.F., Kim, D., Klein, J., Le Traon, Y.: You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 102–113 (2019a)

Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1–12 (2019b)

Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: TBar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 31–42 (2019c). https://doi.org/10.1145/3293882.3330577

Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T.F., Kim, D., Wu, P., Klein, J., Mao, X., Traon, Y.L.: On the efficiency of test suite based program repair: a systematic assessment of 16 automated repair systems for java programs. In: Proceedings of the 42nd International Conference on Software Engineering, pp. 615–627. ACM (2020). https://doi.org/10.1145/3377811.3380338

Liu, K., Li, L., Koyuncu, A., Kim, D., Liu, Z., Klein, J., Bissyandé, T.F.: A critical review on the evaluation of automated program repair systems. J. Syst. Softw. **171**, 110817 (2021). https://doi.org/10.1016/j.jss.2020.110817

Liu, K., Zhang, J., Li, L., Koyuncu, A., Kim, D., Ge, C., Liu, Z., Klein, J., Bissyandé, T.F.: Reliable fix patterns inferred from static checkers for automated program repair. ACM Trans. Softw. Eng. Methodol. (2022). https://doi.org/10.1145/3579637

Long, F., Amidon, P., Rinard, M.: Automatic inference of code transforms for patch generation. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, pp. 727–739 (2017)

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

Monperrus, M.: Automatic software repair: a bibliography. ACM Comput. Surv. **51**(1), 17–11724 (2018)

Mueller, B.: Smashing ethereum smart contracts for fun and real profit. In: 9th Annual HITB Security Conference (HITBSecConf), p. 54 (2018)

Nguyen, T.D., Pham, L.H., Sun, J., Lin, Y., Minh, Q.T.: sFuzz: an efficient adaptive fuzzer for solidity smart contracts. CoRR (2020). arXiv:2004.08563

Nguyen, T.D., Pham, L.H., Sun, J.: sGUARD: towards fixing vulnerable smart contracts automatically (2021). arXiv preprint arXiv:2101.01917

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 653–663 (2018)

(Not So) Smart Contracts. https://github.com/crytic/not-so-smart-contracts (Last Accessed: July 2021)

O'Leary, R.-R.: Parity team publishes postmortem on $160 million ether freeze (2017)

Remix. https://remix.ethereum.org/ (Last Accessed: July 2021)

Rolim, R., Soares, G., Gheyi, R., D'Antoni, L.: Learning quick fixes from code repositories (2018). arXiv preprint arXiv:1803.03806

Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R.: ELIXIR: effective object-oriented program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 648–659 (2017)

Saha, S., Saha, R.K., Prasad, M.R.: Harnessing evolution for multi-hunk program repair. In: Proceedings of the 41st International Conference on Software Engineering, pp. 13–24. IEEE (2019). https://doi.org/10.1109/ICSE.2019.00020

Solidity. https://github.com/ethereum/solidity (Last Accessed: July 2021)

SWC-registry. https://smartcontractsecurity.github.io/SWC-registry (Last Accessed: July 2021)

This is the very first iteration of the Decentralized Application Security Project (or DASP) Top 10 of 2018. https://dasp.co/ (Last Accessed: July 2021)

Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, pp. 9–16 (2018)

Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, pp. 664–676. ACM (2018). https://doi.org/10.1145/3274694.3274737

Wan, Z., Xia, X., Lo, D., Chen, J., Luo, X., Yang, X.: Smart contract security: a practitioners' perspective (2021). arXiv preprint arXiv:2102.10963

Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C.: Context-aware patch generation for better automated program repair. In: Proceedings of the 40th IEEE/ACM International Conference on Software Engineering, pp. 1–11 (2018)

Wüstholz, V., Christakis, M.: Harvey: a greybox fuzzer for smart contracts. CoRR (2019). arXiv:1905.06944

Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., Zhang, L.: Precise condition synthesis for program repair. In: Proceedings of the 39th IEEE/ACM International Conference on Software Engineering, pp. 416–426. IEEE (2017). https://doi.org/10.1109/ICSE.2017.45

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

Yuan, Y., Banzhaf, W.: ARJA: automated repair of java programs via multi-objective genetic programming. IEEE Trans. Softw. Eng. (2018). https://doi.org/10.1109/TSE.2018.2874648

Zhang, Q., Wang, Y., Li, J., Ma, S.: EthPloit: from fuzzing to efficient exploit generation against smart contracts. In: Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, pp. 116–126 (2020a). https://doi.org/10.1109/SANER48275.2020.9054822

Zhang, P., Yu, J., Ji, S.: ADF-GA: data flow criterion based test case generation for ethereum smart contracts. CoRR abs/2003.00257 (2020b). arXiv:2003.00257

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Qianguo Chen[1] · Teng Zhou[1] · Kui Liu[1] · Li Li[2] · Chunpeng Ge[1] · Zhe Liu[1] · Jacques Klein[3] · Tegawendé F. Bissyandé[3]

✉ Kui Liu
kui.liu@nuaa.edu.cn

Qianguo Chen
cqgboy@163.com

Teng Zhou
tengzhou@nuaa.edu.cn

Li Li
li.li@monash.edu

Chunpeng Ge
gecp@nuaa.edu.cn

Zhe Liu
zhe.liu@nuaa.edu.cn

Jacques Klein
jacques.klein@uni.lu

Tegawendé F. Bissyandé
tegawende.bissyande@uni.lu

[1] Nanjing University of Aeronautics and Astronautics, Nanjing, China

[2] Monash University, Melbourne, Australia

[3] University of Luxembourg, Luxembourg City, Luxembourg