VFIX: Facilitating Software Maintenance of Smart Contracts via Automatically Fixing Vulnerabilities

Pengcheng Fang [2], Peng Gao [3], Yun Peng [4], Qingzhao Zhang [5], Tao Xie [6], Dawn Song [7]
Prateek Mittal [8], Sanjeev Kulkarni [8], Zhuotao Liu [9], Xusheng Xiao [1]*
[1]Arizona State University, [2]Case Western Reserve University, [3]Virginia Polytechnic Institute and State University
[4]The Chinese University of Hong Kong, [5]Shanghai Jiaotong University [6]Peking University
[7]University of California, Berkeley, [8]Princeton University [9]Tsinghua University
Email: xusheng.xiao@asu.edu

***Abstract*—The increased adoption of smart contracts in many industries has made them an attractive target for cybercriminals, leading to millions of dollars in losses. Thus, continuously fixing newly found vulnerabilities of smart contracts becomes a routine software maintenance task for running smart contracts. However, fixing the vulnerabilities that are specific to the smart contract domain requires security knowledge that many developers lack. Without effective tool support, this task can be very costly in terms of manual labor.**

To fill this critical need, in this paper, we propose VFIX, which automatically generates security patches for vulnerable smart contracts. In particular, VFIX provides a novel program analysis framework that can incorporate different fix patterns for fixing various types of vulnerabilities. To address the unique challenges in accurately fixing smart contract vulnerabilities, VFIX innovatively combines template-based repair with a set of static program analysis techniques specially designed for smart contracts. Specifically, given an input smart contract, VFIX conducts ensemble identification based on multiple static verification tools to identify vulnerabilities for an automatic fix. Then, VFIX generates patches using template-based fix patterns, and conducts static program analysis (e.g., program dependency computation, pointer analysis) for smart contracts to accurately infer and populate the parameter values for the fix templates. Finally, VFIX performs static verification to ensure that the patched contract is free of vulnerabilities. Our evaluations on 144 real smart contracts containing different types of vulnerabilities show that VFIX can successfully fix 94% of the vulnerabilities and preserve the expected normal behaviors of the smart contracts.

I. INTRODUCTION

As a paradigmatic application of blockchain [1], smart contracts enable the creation of decentralized general-purpose applications and have received wide adoption [2], [3], [4], [5]. However, it is challenging to create smart contracts without security vulnerabilities, partly due to the lack of security knowledge by developers in the new ecosystem of smart contract languages (*e.g.,* Solidity [6]) and platforms (*e.g.,* permissionless blockchains such as Ethereum [2], [7]). Over the past few years, the blockchain community has witnessed a number of critical vulnerabilities in smart contracts being exploited by attackers, leading to millions of dollars in losses [8], [9], [10], [11], [12], [13]. For example, the reentrancy attack on TheDAO contract [14] in 2016 resulted in $50M worth of Ether being stolen [12], [15].

*Corresponding Author

Despite considerable research efforts [16], [17], [18], [19], [20], [21], [22] of tool support for detecting vulnerabilities in smart contracts, fixing these vulnerabilities is highly critical, yet lacking of effective tool support, for two main reasons. First, new vulnerabilities are continuously being discovered with the rapid development of smart contracts, and thus fixing new vulnerabilities and redeploy fixed smart contracts with upgradable support [23] becomes a routine maintenance task to improve the security of smart contracts. However, this maintenance task requires security knowledge that many developers lack and is in a dire need of effective automated tool support. Second, manually fixing a smart contract is often challenging and error-prone (see Section II-C). For example, the best practice to avoid reentrancy vulnerability is to ensure all internal state changes are performed before the external call is executed (*i.e.,* the Checks-Effects-Interactions pattern) [24], [15]. Hence, the patch for *reentrancy vulnerability* requires (1) reordering multiple statements to ensure that all updates to contract state variables occur before the external call, and (2) creating temporary variables to store the values of these state variables for eliminating data dependencies on the external call (see Figure 1).

Although various existing techniques for automated program repair [25], [26], [27], [28], [29] can automatically generate patches to fix the given program's bugs, these techniques are often not applicable to effectively fix vulnerabilities for smart contracts for two main reasons. First, applying existing repair techniques to repair contract vulnerabilities typically requires a comprehensive test suite to assure that all detected vulnerabilities are fixed and no side effects are introduced by the generated patch. Previous works [30] show that it is highly difficult to create a comprehensive test suite that can defend against all types of exploits. Second, applying existing search-based repair techniques [31], [26], [27], [28], [29], [32], [33] (being mainstream ones) to fix contract vulnerabilities fails to generate patches for some important types of contract vulnerabilities. These techniques explore the search space of repairs based on syntactic mutators, by leveraging search algorithms such as genetic programming or random search. However, the strategies of these techniques are mostly adding conditional checks or replacing a statement with another existing statement, which is insufficient for fixing contract

```
mapping (address => uint) userBalances;
mapping (address => uint) lastPaymentDate;
...
function refund ( ) public {
      require(userBalances[msg.sender] > 0);
+     var balance = userBalances[msg.sender] ;
+     userBalances[msg.sender] = 0;
+     lastPaymentDate[msg.sender] = now;
+     msg.sender.call.value(balance);
-     msg.sender.call.value(userBalances[msg.sender]);
-     userBalances[msg.sender] = 0
-     lastPaymentDate[msg.sender] = now;
}
```

Fig. 1: Example patch for *Reentrancy* vulnerability

vulnerabilities that require temporary variable creations and statement reordering (*e.g.,* fixing reentrancy vulnerabilities). Although one can simply adapt these techniques to include more complex fixing strategies, doing so tend to (1) result in an exponential expansion of the search space [31], [34], [35], inducing patch-generation ineffectiveness.

To effectively fix vulnerabilities for facilitating software maintenance of smart contracts, in this paper, we propose vFIX, that (1) automatically detects vulnerabilities in a smart contract, (2) applies patches to multiple detected vulnerabilities, and (3) verifies the patched contract before the contract deployment. In particular, vFIX is built upon our novel static program analysis infrastructure that is specially optimized for Solidity, which can automatically fix different vulnerabilities with an extensible set of fix strategies.

vFIX is powered by three key designs. First, to avoid wasting later high cost of searching for patches of detected vulnerabilities being false positives or not amenable for automatic fix (*e.g.,* handling external method calls without source code), vFIX synergistically combines multiple static verification tools [18], [36], [37] with post-processing. In particular, vFIX first applies these static verification tools to detect vulnerability candidates and adopts majority voting to determine which candidates are more likely to be real vulnerabilities rather than false-positive ones. vFIX then conducts post-processing to extract the required information from the reported vulnerabilities (*e.g.,* identifying the types of the data dependencies for reentrancy vulnerabilities) and filter out candidates that are not amenable for automatic fix.

Second, to address the space explosion during the search for target patches and preserve expected contract behaviors, vFIX generates patches using template-based fix patterns [38]. We leverage the template-based approach as it can generate complex patches and fix *multiple* vulnerabilities efficiently. vFIX further conducts *static program analysis* to accurately infer variable values from the contract program under analysis without the need for searching a huge repair space. Most smart contracts restrain the uses of references in the language level (*e.g.,* Solidity limits references to specific types), enabling our static analysis techniques to compute precise program dependencies for generating complex patches such as moving statements without violating data dependency constraints (Section IV-B1). Particularly, our program analysis allows vFIX to support fix patterns with different performance overheads. For example, vFIX supports both adding global locks and moving statements to fix reentrancy vulnerabilities, and prefers moving statements as the resulting program requires much less gas cost (5 v.s. 25000).

Third, vFIX reapplies the static verification techniques used to detect the vulnerabilities on the patched smart contract, and further verifies that the detected security vulnerabilities are eliminated in the patched smart contract. As most static verification techniques employ sound program analysis, they produce false positives but *no false negatives*. Thus, by combining template-based fix patterns with static verification, not only vFIX can guarantee that the patched smart contract preserves the expected contract behaviors, the static verification techniques also can ensure the elimination of the patched vulnerabilities.

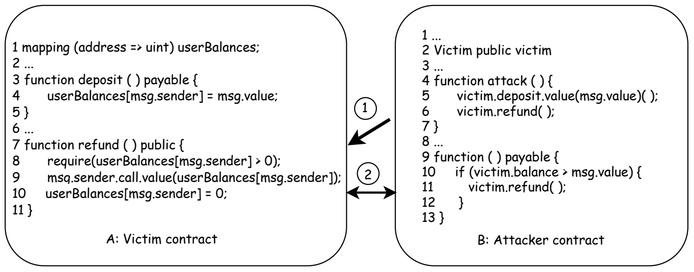
In summary, our paper makes the following contributions:

- To support software maintenance of smart contracts in eliminating detected vulnerabilities, we propose a novel framework, named vFIX, that (1) leverages the synergy of multiple static verification tools to detect vulnerabilities in a smart contract, (2) generates source code patches for the contract, and (3) performs static verification to ensure the elimination of the vulnerabilities.
- We propose a novel set of program analysis techniques that extract variable values from smart contracts to generate patches based on the fix patterns for four major types of vulnerabilities.
- We implement the prototype of vFIX to fix four major types of vulnerabilities: *Reentrancy*, *MissingInputValidation*, *LockedEther*, *UnhandledException*, and make the source code publicly available at [39]. These four types cover the majority of the vulnerability population from our measurement study of vulnerable contracts in the wild.
- We conduct systematic evaluations on (1) 50 contracts ($20,510$ LOC) selected from a widely used dataset [40] of smart contracts with injected vulnerabilities, and (2) 94 contracts ($120,894$ LOC) selected among $4,940$ real smart contracts with the largest number of transactions from Etherscan [41]. The results show that the majority voting scheme is highly precise in detecting vulnerabilities, and vFIX changes 7.97 lines on average to successfully generate patches for 565 out of 601 vulnerabilities, *achieving a high success rate ($> 94\%$)*. Additionally, we crawl $\sim 125,000$ transactions from Etherscan [41] and replay these transactions on both the original contracts and the patched contracts. The results show that the patched contracts preserve the original contract functionalities, and the increases of the gas caused by the extra security checks are negligible ($\sim$ \$0.000027).

II. BACKGROUND AND MOTIVATION STUDY

A. *Smart Contract and Ethereum*

The very first blockchain, Bitcoin [1], which supports limited scripting [42] for its transactions, can already run simple



```
1 mapping (address => uint) userBalances;
2 ...
3 function deposit ( ) payable {
4       userBalances[msg.sender] = msg.value;
5 }
6 ...
7 function refund ( ) public {
8       require(userBalances[msg.sender] > 0);
9       msg.sender.call.value(userBalances[msg.sender]);
10      userBalances[msg.sender] = 0;
11 }
```

```
1 ...
2 Victim public victim
3 ...
4 function attack ( ) {
5       victim.deposit.value(msg.value)( );
6       victim.refund( );
7 }
8 ...
9 function ( ) payable {
10     if (victim.balance > msg.value) {
11        victim.refund( );
12     }
13 }
```

Fig. 2: An exploit of *Reentrancy* vulnerability

smart contracts such as freezing funds until a time stamp in the future [42] and decentralized lotteries [43]. Ethereum [7] and other blockchains (*e.g.,* Hyperledger [44] and Corda [45]) support general-purpose computation for smart contracts, and thus it is far less complicated to build a much wider range of decentralized applications (Dapps). In Ethereum, the Ethereum Virtual Machine (EVM) is a virtual machine designed as the runtime environment for smart contracts.

B. Vulnerabilities in Smart Contracts

Recently, an increasing number of high-profile attacks resulting in huge financial losses have been reported. We next illustrate a list of representative vulnerabilities [46].

Reentrancy: In July 2016 [12], a fault in TheDAO contract allowed an attacker to steal $50M. The root cause of the attack is to re-enter a non-recursive function before its termination. Figure 2 shows an exploit of the *Reentrancy* vulnerability. First, the *attack()* function in the attacker contract is called, causing to deposit some ethers in the victim contract and then invoke the victim's vulnerable *refund()* function. Then, the *refund()* function sends the deposited ethers to the attacker contract (Line 9 in A), also triggering the unnamed fallback function in the attack contract (Line 9 in B). Next, the fallback function again calls the *refund()* function in the victim contract (Line 11 in B). Since the victim contract updates the *userBalances* variable (Line 10 in A) after the ether transfer call, *userBalances* remains unchanged when the attacker re-enters the *refund()* function, and thus the balance check (Line 8 in A) can still be passed. As a consequence, the attacker is able to repeatedly siphon off ethers from the victim contract and exhaust its balance.

Missing Input Validation: If developers forget to assign correct values to the arguments, EVM will execute the function using the default values based on the argument types. This mechanism makes smart contracts vulnerable to the attacks on function arguments.

Locked Ether: In 2017, a vulnerable contract relies on another library contract to withdraw its funds (using *delegatecall*). Unfortunately, a user accidentally removed the library contract from the blockchain (using the *kill* instruction), and thus the funds in the wallet contract could not be extraced any more [9].

Unhandled Exception: In Solidity, there are multiple situations where an exception may be raised. Unhandled exceptions can affect the security of smart contracts. In February 2016, a vulnerable contract [8] forced the owner to ask the users not to send ether to the owner because of an unhandled exception in the *call* instruction.

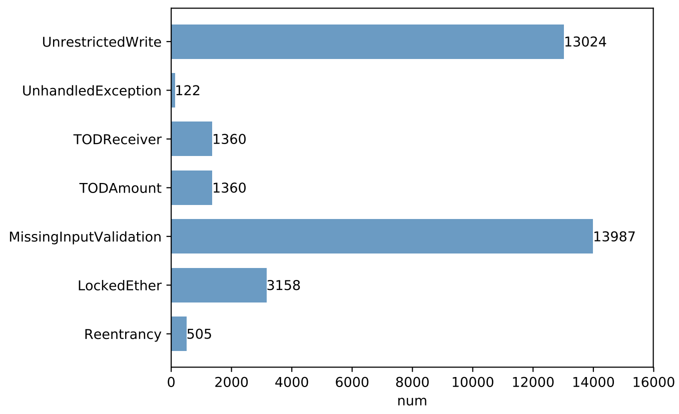


Fig. 3: Vulnerabilities detected by Securify for 4,640 smart contracts collected from Etherscan

Vulnerability Detection: To detect vulnerabilities in a smart contract, existing research has proposed techniques based on testing [47] and symbolic execution [16], [17]. Recently, Securify [18], a verification-based solution, has shown its superiority over previous techniques. The analysis in Securify consists of two steps: (1) Securify symbolically analyzes the contract's dependency graph to derive precise semantic facts; (2) Securify uses the derived facts to check the compliance and violation patterns that capture sufficient conditions for proving whether some security properties hold.

C. Motivation Study

As static-verification-based solutions for vulnerability detection use different security properties to detect different types of vulnerabilities, we need to study the prevalence of vulnerabilities so that VFIX's patch generation strategies can target the most effective properties. While Securify produces false positives, *it is an abstract interpreter that provides soundness guarantees over all possible executions*, and thus it can be used to estimate the trend with affordable manual inspection. Figure 3 shows the vulnerability distribution obtained by applying Securify on a set of $4,640$ smart contracts (with the most transactions) collected from Etherscan. In summary, there are $33,516$ vulnerabilities and each contract contains 7.22 vulnerabilities on average. These results show that vulnerabilities are commonly found in smart contracts and multiple vulnerabilities may often exist in one contract, making manual fixing labor intensive and error prone.

Based on the vulnerability distribution in Figure 3, we select the types of vulnerabilities to include in VFIX's fixing scope. The most common types of vulnerabilities are *MissingInputValidation* and *UnrestrictedWrite* (count $> 13,000$). As *MissingInputValidation* can be fixed via source code transformation, VFIX includes it in its fixing scope. For *UnrestrictedWrite*, the security property used to detect the vulnerabilities

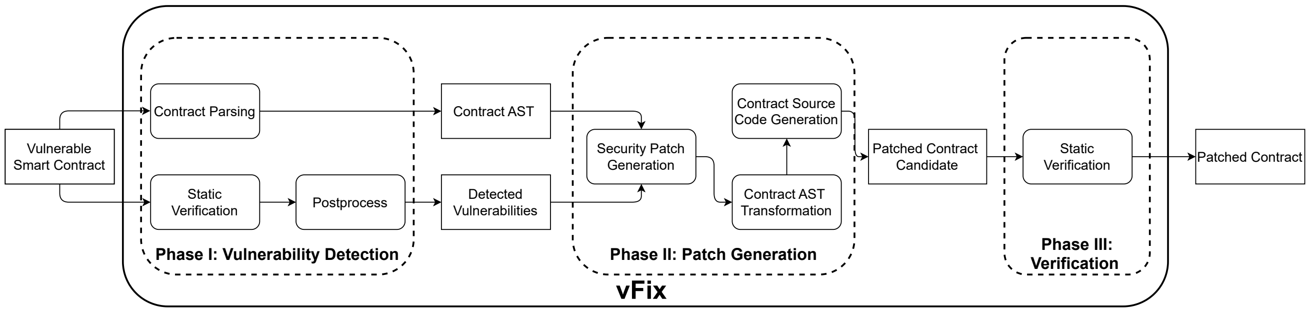

Vulnerable Smart Contract
Contract Parsing
Static Verification
Postprocess
Phase I: Vulnerability Detection
Contract AST
Detected Vulnerabilities
Security Patch Generation
Contract Source Code Generation
Contract AST Transformation
Phase II: Patch Generation
Patched Contract Candidate
Static Verification
Phase III: Verification
Patched Contract
vFix

Fig. 4: Overview of vFix

is too strict, and most of the detected vulnerabilities are false positives. Thus, vFix excludes *UnrestrictedWrite*. *TODAmount* and *TODReceiver* are caused by the indeterminism of the transaction executing order [48], which depends on the miner that mines the block. Such uncertainty is inherent in blockchain execution platforms and cannot be fixed by modifying the smart contract source code. Fixing them needs to change the operational semantics of Ethereum, requiring all the clients in the Ethereum network to upgrade. As doing so is not a practical solution, we exclude these types of vulnerabilities from vFix's fixing scope. Furthermore, we include *Reentrancy*, *LockedEther*, and *UnhandledException* types of vulnerabilities in vFix's fixing scope. These types of vulnerabilities are commonly found in smart contracts and their fixing strategies are different from each other, making them good candidates to demonstrate the effectiveness of vFix in both simple and complex patches. The four types of vulnerabilities that vFix focuses on account for 53.0% of the total vulnerabilities.

III. Overview

Figure 4 illustrates the architecture of vFix. Given a smart contract, vFix generates source code patches for the detected vulnerabilities. vFix consists of three phases, as shown in Figure 4. In Phase I, vFix conducts static verification to detect vulnerability candidates and then performs majority voting and post-processing to identify the four types of vulnerabilities. vFix also generates an abstract syntax tree (AST) from the contract's source code. Both the AST and the identified vulnerabilities are used by Phase II for security patch generation. In Phase II, vFix first locates the vulnerabilities in the source code and obtains their context information via data-flow analysis and type hierarchy analysis. Then, based on the vulnerability types and the context information, vFix transforms the AST to generate patches. vFix repeats this patching process for each detected vulnerability until all vulnerabilities are patched. Finally, vFix converts the transformed AST back to the source code and generates a patched contract. In Phase III, vFix conducts static verification on the patched contract and checks whether the vulnerabilities are eliminated successfully. If the patched contract passes the verification, vFix outputs the patched contract with the details of the patches, such as the changed lines and the types of the targeted vulnerabilities.

IV. Design of vFix

A. Phase I: Vulnerability Detection

In this phase, vFix first conducts static verification to detect vulnerability candidates. Static verification tools [18], [16], [17] adopt over-approximation analysis, which may produce false-positive violations. To address this issue, vFix combines three static verification tools: *Securify* [18], *Slither* [36], and *Smartcheck* [37] to detect vulnerabilities, and leverages the majority voting mechanism to improve the detection accuracy. As some of the detected vulnerabilities are not amenable for automatic fix (*e.g.,* handling external method calls without source code), vFix focuses on the detected vulnerabilities that have severe security impacts based on our motivating study (Section II) and are amenable for automatic fix:

- *Reentrancy*: These vulnerabilities can be detected by *Slither* and *Securify*. However, for some detected vulnerabilities, the return value of external function call is used to control whether to update the state variables. As it is almost impossible to verify the behavior of external function calls, vFix cannot generate a patch properly. Also, some updates of the state variables depend on timestamps, and any patch that moves the updates will cause semantics changes. Thus, vFix ignores the reentrancy vulnerabilities caused by external function calls.
- *MissingInputValidation*: This type of vulnerability can be detected by *Securify*. Except for function arguments of the *address* type, function arguments of other data types (*e.g.,* integers) can have a wide range of values and it requires dynamic analysis to determine the runtime values. Thus, vFix only fixes the *MissingInputValidation* vulnerabilities that concern about the *address* type arguments.
- *LockedEther*: This type of vulnerability can be detected by both *Slither*, *Securify*, and *Smartcheck*. For some contracts, developers use them as the library, which is not assumed to receive ether. Thus, for these contracts, it is not reasonable for them to have a function that can send out Ether. Thus, vFix will not fix the *LockedEther* violations for these contracts.
- *UnhandledException*: This type of vulnerability can be detected by *Slither*, *Securify*, and *Smartcheck*. When developers assign the return value of the Ether transfer function *send()* to a variable, they often provide more code to handle the exceptions. Thus, vFix fixes the violations where developer does not process the return value of *send()*.

TABLE I: Summary of fix patterns

Type	Level	Fix Pattern
Reentrancy	Method	Lock [15], [22], Reorder statements [15]
MissingInputValidation	Method	Require check [49]
LockedEther	Contract	Withdraw function [50]
UnhandledException	Statement	Require check [51]

To identify these types of vulnerabilities that are amenable for automatic fix, vFix performs post-processing on the reported vulnerabilities based on the syntactic analysis on the AST and intra-procedural control and data flow analysis. For example, for a reported reentrancy vulnerability, detecting whether an external function call is used to control the execution of a state variable update will require both control and data flow analysis.

B. Phase II: Patch Generation

vFix performs static program analysis to extract the context information of the detected vulnerabilities, and supports fix patterns in three granularity levels: *statement level*, *method level*, and *contract level*. Table I summarizes the fix patterns supported by vFix. We can see that vFix can support a wide range of fix patterns, while existing work often supports one or two patterns [15], [22]. For example, sGUARD [22] supports only adding locks for fixing reentrancy vulnerabilities, while vFix additionally supports the fix pattern by reordering specific statements. Algorithm 1 shows the patch generation algorithm of vFix.

1) Program Analysis Infrastructure: vFix customized static program analysis techniques to extract the necessary context information for vulnerabilities in smart contracts, including involved variables and their control and data dependencies. We next describe the static program analysis techniques employed by vFix.

Intra-Procedural Data-flow Analysis: vFix performs an intra-procedural data-flow analysis to collect the program points (*i.e.,* statements) where a variable is created, read, modified, and deleted [52], [53]. Our intra-procedural data-flow analysis starts with building the method's control flow graph (CFG), where each statement is considered as a single basic block for the convenience of dependency analysis. It is worth mentioning that modifiers assigned to methods in smart contracts can be executed both before and after the execution of the method body and the parameters of methods can be used in the modifiers. Thus, the control flow of a method follows this sequence: modifier, method body, modifier. Once the CFG is built, existing data-flow analysis is employed to build the data-flow graph (DFG) for the method. Note that fixing *Reentrancy* requires inter-procedural analysis, which is achieved by combining the method summaries with the intra-procedural analysis.

Pointer Analysis for Solidity: Pointer analysis is known to be expensive and is required for precise analysis (*e.g.,* flow-sensitivity analysis and context-sensitivity analysis). As smart contract languages such as Solidity restrain the usage of references, existing pointer analysis can be easily adapted for obtaining accurate point-to information for the contracts. In

Algorithm 1: Security Patch Generation of vFix

```
Input: $ast$ as the AST of original source code, $violations$
    as the detected vulnerabilities
Output: $p\_ast$ as the patched AST
1: // Patching UnhandledException
2: for each $vulnerability$ in
   $violations.UnhandledException$ do
3:    $call \leftarrow vulnerability.line$
4:    $validations \leftarrow$ addCheck($call$)
5:    $patch \leftarrow$ patchGenerator($validations$)
6:    $p\_ast \leftarrow$ ASTTransformer($patch, ast$)
7: // Patching Reentrancy
8: for each $vulnerability$ in $violations.Reentrancy$ do
9:    $call \leftarrow vulnerability.line$
10:   $func \leftarrow$ locateFunc($ast, call$)
11:   $func\_sums \leftarrow$ scan($ast$)
12:   $pointer\_info \leftarrow$ pointerAnalysis($ast$)
13:   $DFG \leftarrow$ dataflowAnalysis($ast, func,$
              $pointer\_info, func\_sums$)
14:   $writes \leftarrow$ findWritestoStorage($func, call$)
15:   $dependences \leftarrow$ dependencyAnalysis($DFG, writes,$
      $call, func$)
16:   $patch \leftarrow$ patchGenerator($dependences, func$)
17:   $p\_ast \leftarrow$ ASTTransformer($patch, ast$)
18: // Patching MissingInputValidation
19: for each $vulnerability$ in
    $violations.MissingInputValidation$ do
20:   $func \leftarrow$ locateFunc($vulnerability.line$)
21:   $unchecked\_parameters \leftarrow$
      checkValidation($func.arguments$)
22:   $validations \leftarrow$
      addValidation($unchecked\_parameters$)
23:   $patch \leftarrow$ patchGenerator($validations$)
24:   $p\_ast \leftarrow$ ASTTransformer($patch, ast$)
25: // Patching LockedEther
26: for each $vulnerability$ in $violations.LockedEther$ do
27:   $contract \leftarrow$ locateContract($vulnerability.line$)
28:   if mustPatch($contract$) == $True$ then
29:      $owner \leftarrow$ findOwner($contract$)
30:      $withdraw \leftarrow$ createWithdraw($owner$)
31:      $patch \leftarrow$ patchGenerator($withdraw$)
32:      $p\_ast \leftarrow$ ASTTransformer($patch, ast$)
```

Solidity ($\geq$v0.6.1), there are three locations where a variable can be stored:

- *memory*: the variable in memory is not persistent and its lifetime is limited to an external method call.
- *storage*: the variable in storage is persistent and its lifetime is the same as the contract's.
- *calldata*: this location is only available for external method call parameters.

Solidity's reference types include *struct*, *array*, and *mapping*. Figure 5 shows an example contract to illustrate reference creations. There are only two situations where a variable of these types can be a reference: (1) assignments

```
1 contract C {
2   uint[] x;
3   uint[] z;
4   function f(uint[] memory memoryArray) public {
5     x = memoryArray; //copies the whole array to storage
6     uint[] storage y = x; // assigns a pointer
7     uint[] memory w = memoryArray; // assigns a pointer
8     z = x; //copies x to z, does not create a reference
9   }
10 }
```

Fig. 5: Reference creations in Solidity

from a variable in storage to a local variable in storage create a reference (Line 6); (2) assignments from a variable in memory to another variable in memory create a reference (Line 7).

To determine which pointer analysis algorithm to use, we analyzed $6,420$ real-world smart contracts to find how often the reference type is used in solidity programs. We scanned all assignments among these contracts and checked if there are variables that meet the definitions of reference types mentioned above. In summary, we found $3,210$ reference variables among $199,724$ assignments in $6,420$ contracts. That is, on average, only 1.6% of the assignments use reference types. Thus, vFix adapts a flow-insensitive and context-insensitive pointer analysis by extending its point-to model to incorporate data locations and reference creations as described above, and computes the point-to information for each variable in a contract [52].

Inter-Procedural Analysis via Method Summary: To enable inter-procedural analysis, for each method, vFix builds a method summary that computes the side effects of the state variables, *i.e.,* whether the state variables are modified in each method. Method summaries (or called function summaries) have been used to build inter-procedural program analysis in a modular way [54], [55], [56], which can be easily combined with other intra-procedural program analysis to enable inter-procedural analysis.

Data Dependency Classification: Based on the inter-procedural analysis, vFix further classifies data dependency into the following types:

- *Flow Dependence* or *Read After Write (RAW):* a statement s_2 is flow dependent on s_1 if and only if s_1 modifies a resource that s_2 reads and s_1 precedes s_2 in execution.
- *Anti-Dependence* or *Write After Read (WAR):* a statement s_2 is antidependent on s_1 if and only if s_2 modifies a resource that s_1 reads and s_1 precedes s_2 in execution.
- *Output Dependence* or *Write After Write (WAW):* a statement s_2 is output dependent on s_1 if and only if s_1 and s_2 modify the same resource and s_1 precedes s_2 in execution.
- *Input Dependence* or *Read After Read (RAR):* a statement s_2 is input dependent on s_1 if and only if s_1 and s_2 read the same resource and s_1 precedes s_2 in execution.

The classification of the data dependency will later be used by the fix patterns (*e.g.,* fixing *Reentrancy* in Section IV-B3) to guide the patch generation.

2) Fix Patterns in Statement Level: Vulnerabilities in this level are usually caused by misuses of individual statements, such as *UnhandledException* that forgets to check method return values.

Fixing *UnhandledException*: The fix pattern (Lines 1-6 in Algorithm 1) for this vulnerability is to check the return value of each coin transfer function: *send()* and *value()*. The type of their return values is boolean because they indicate whether the transfers succeed. Transactions in which these transfers fail must be reverted to notice the caller to ensure the coherence between the contract states and the transactions. Thus, to fix this vulnerability, vFix adds a *require()* function call to validate the return values of *send()* and *value()* (Line 4 in Algorithm 1) and ensures that their executions are successful before completing the whole transaction.

3) Fix Patterns in Method Level: Vulnerabilities in this level are usually caused by missing parameter checks or miuse some method calls in a method. The two types of vulnerabilities in this level are *Reentrancy* and *MissingInputValidation*.

Fixing *Reentrancy*: The preferred fix pattern for *Reentrancy* (Lines 8-17 in Algorithm 1) is to move all writes to storage ahead so that there is no write to storage after an external method call or a coin transfer call, such as the patch shown in Figure 1. vFix first identifies the method that has the vulnerability (Lines 9-10), and computes the method summary, the pointer information, and the DFG of the method (Lines 11-13). vFix then identifies the writes that result in the vulnerability (Line 14), and further computes the data dependencies that are used to move the writes (Line 15). In particular, if any of these writes (represented as w) has data dependencies to the variables used by the external calls (represented as c), depending on the type of the dependencies, vFix may eliminate such dependencies without changing the semantics before moving the writes ahead:

- For *flow dependence* from w to a statement s, vFix creates a temporary variable to store the value of the variables in w before they are written and replace the same variables in c with these temporary variables, so that w does not impact c if w is moved ahead.
- For *anti-dependence* and *output dependence* from w to a statement s, vFix moves both w and s ahead if s is not the external call.
- For *input dependence* from w to a statement s, vFix simply moves w ahead since there are no side effects in this type of dependency.

However, when there is a *anti-dependence* or *output dependence* from w to c, the data dependencies cannot be eliminated. Because in this case the updates in w must wait for the execution results of c so w cannot be moved ahead of c. In this case, vFix adopts another more expensive fix pattern, which declares a new global *bool* value as a lock to limit the method invocations [22]. This lock will not allow the unexpected recall, if the previous call does not finish the execution. As the modification of a global variable is much more expensive than the declarations of local variables in smart contracts, we

```
1 function transferFrom(address src, address dst, uint wad) public returns (bool) {
2       //validation that checks wad
3       require(balanceOf[src] >= wad);
4       //validation that checks src and dst
5       require(dst != address(0x0));
6       require(src != address(0x0));
7       ...
8       return true;
9 }
```

Fig. 6: Patch for *MissingInputValidation* vulnerability

```
function withdraw(uint val) public {
      require(msg.sender == owner && val <= address(this).balance);
      msg.sender.transfer(val);
}
```

Fig. 7: *Withdraw()* function inserted to patch *LockedEther* vulnerability

find out that for each transaction, the global lock increases the gas cost by $\sim$ 25000 for the function, while declaring temporary variables and moving statements increase the gas cost by only 5.

Fixing *MissingInputValidation*: The fix pattern for *MissingInputValidation* is to add conditional checks to validate the method parameters at the beginning of method body (Lines 18-25 in Algorithm 1). To patch this vulnerability, vFix first identifies the method parameters that are not validated (Line 21 in Algorithm 1). To do so, vFix checks whether the parameters appear in any *require()* method, which is often used in Solidity to perform validation. This checking can be easily done by using the DFG of the method. vFix inserts validations for the other unchecked parameters:

- For those parameters whose type is address, vFix adds a common validation to check whether this address is $0x0$ because an address with value *0x0* is invalid.
- For parameters whose type are integer, vFix adds a safe math library to prevent integer overflow and underflow when doing calculations.
- For parameters whose type are bytes or self-defined, vFix may not add proper validations because it lacks contextual information for vFix to acquire sufficient information about their valid ranges.

Consider the contract in Figure 6. The parameter *wad* is validated by the highlighted statement (Line 6) and thus there is no need for further validation. For the parameters *src* and *dst*, vFix adds a *require()* method to validate their values (Lines 8-9).

4) Fix Patterns in Contract Level: Vulnerabilities in this level are usually related to the properties of contracts instead of a specific method or statement.

Fixing *LockedEther*: Our fix pattern for *LockedEther* is to add a *withdraw()* function to enable the transferring of Ether (Lines 25-32 in Algorithm 1). While the major part of the method is quite straightforward, vFix needs to protect the contract balances. Thus, vFix adds two validations to constrain the Ether transfer: (1) the transaction initiator who calls this method must be the *owner* of the contract, (2) the amount of balance transferred out cannot be larger than the total balance of contract. An example of this *withdraw()* function is shown in Figure 7. The validation of the balance is to insert *require(amount* $<=$ *this.balance)*. However, the validation of the owner requires further analysis of the type hierarchy for the vulnerable contract. To identify the owner of a contract (Line 29 in Algorithm 1), vFix builds an inheritance graph of the vulnerable contract and checks the contract constructors in all of its parent contracts to see whether there is an *owner* field with the initial value *msg.sender*. If it is found, vFix uses it to implement the owner validation. Otherwise, vFix defines a new *owner* field and assigns *msg.sender* as its initial value in the constructor of the vulnerable contract. Then, based on the type hierarchy for the vulnerable contract, vFix inserts the *withdraw()* method in the base contract for the vulnerable contract (Line 30). This fix pattern can largely reduce the lines of code inserted into the contracts and save gases required when executing the patched contracts.

When all the identified vulnerabilities are patched, vFix converts its transformed AST back to source code and outputs the patched contract to Phase III.

TABLE II: Vulnerabilities detected by each detector

Type	Securify	Slither	Smartcheck	Majority
Reentrancy	107	461	0	23
MissingInputValidation	979	0	0	131
LockedEther	184	100	43	137
UnhandledException	83	129	36	60
Total	1353	690	79	351

C. Phase III: Patch Verification

In this phase, vFix reapplies the static verification tools to ensure that the detected vulnerabilities are eliminated while the expected behaviors are preserved in the patched contract. Once a patched contract candidate is generated, vFix applies static verification again on the patched contract and checks whether the vulnerabilities reported in Phase I are eliminated. If the static verification tool no longer reports the same vulnerabilities, the patched contract is considered to pass the static verification.

V. Evaluation

We implemented vFix in JavaScript ($\sim$5000 lines of code). We adopted three state-of-the-art vulnerability tools, Securify [18], *Slither* [36], and *Smartcheck* [37], as the static verification tools for vFix. We built the parsing and transformation modules upon an open source Solidity parser [57] built on top of ANTLR4 [58]. We evaluated the effectiveness of vFix in fixing vulnerabilities in real-world smart contracts. Specifically, our evaluations aim to answer the following research questions.

- **RQ1:** How effective is vFix in generating successful patches for vulnerable contracts?
- **RQ2:** How effective is the synergy of static verification and post-processing in detecting vulnerabilities, compared to static verification only?

TABLE III: Vulnerable contracts used in evaluation

Type	Contract Count	Lines of Code
Reentrancy	17	8,522
MissingInputValidation	21	26,102
LockedEther	20	17,277
UnhandledException	19	30,349
Mixed	17	38,644
Injected	50	20,510
Total	144	141,404

- **RQ3:** How efficient is VFIX in generating patches?

A. Evaluation Setup

Our evaluation datasets have 144 contracts ($141,404$ lines of Solidity code), and the evaluations are conducted on a server with Intel(R) Xeon(R) CPU E5-2637 v4 (3.50GHz) and 256GB RAM.

Injected Vulnerabilities: We use a widely adopted dataset [40] that contains smart contracts with various types of injected known vulnerabilities (*e.g.,* integer overflow/underflow, reentrancy, and timestamp-dependency). We choose 50 contracts with injected reentrancy vulnerabilities as other types of vulnerabilities in the dataset are not the focus of VFIX. These contracts provide the detailed information of the injected vulnerabilities and thus we can directly evaluate our patch generation without applying vulnerability detection.

Real Vulnerabilities: We collect real smart contracts based on the addresses obtained from BigQuery [59], and select the top $10,000$ contracts sorted by the number of transactions. We further download the source code from Etherscan [41] based on the addresses, and exclude the contracts whose source code is not available. We then apply static verification on these downloaded contracts and exclude the ones that cannot be analyzed by the Solidity compiler due to version incompatibility. In total, we obtain $4,640$ contracts. For each contract to be used in our evaluation, we inspect the source code to confirm vulnerabilities, verify the patches, and prepare test cases that exercise the vulnerable behaviors of the original contracts, which requires non-trivial manual efforts. Within our affordable efforts, we select 94 vulnerable contracts that have the four types of vulnerabilities described in Section II-C, as shown in Table III. The test cases are shown in Table IV. If a contract has several types of vulnerabilities, we classify it into the *Mixed* category.

Semantic Validation: To ensure that the patched contracts preserve the expected behaviors, semantic validation is conducted using a smart contract testing platform named Truffle [60]. Truffle allows us to deploy a contract on a local blockchain powered by Ganache and make a method call or issue a transaction. To obtain the test cases for checking contracts' expected behaviors, we made use of the publicity of all transactions on blockchain: we downloaded the existing transactions of a contract and extracted the input values to create test cases because these transactions should reflect the functionality of a contract. But these constructed test cases lack test oracles to assert the expected results. To address this

TABLE IV: Effectiveness of VFIX in patch generation

Type	Total	Success	Fail	Suc. Rate	Test Case
Reentrancy	23	21	2	0.91	73
MissingInputValidation	131	128	3	0.98	68
LockedEther	137	136	1	0.99	60
UnhandledException	60	60	0	1.00	245
Injected	250	220	*30	0.88	-
Total	601	565	36	0.94	446

problem, we consider the original contract and the patched contract as different implementations for the same requirements [61], [62] and assert that the states of both contracts (i.e., the values of the on-chain persistent state variables) should be the same after the testing. Specifically, VFIX first deploys both the patched contract and the original contract on the Truffle with the same initial states. Then, VFIX runs the test cases and compares the states of both contracts. If the states of both contracts remain the same after the testing, VFIX considers that the expected behaviors are preserved in the patched contract.

B. RQ1: Effectiveness in Patch Generation

We run VFIX on each of the contracts in our testing set to generate a patched contract. We then manually examine the patched contract and verify its correctness. We also examine why VFIX fails to generate patches for certain contracts.

Overall Results: Table IV shows the path generation results. On average, each contract in the testing set has 4.17 vulnerabilities. We consider a patch is successful if the patched contracts pass the test cases written to exercise the vulnerable behaviors (Table IV) and the test cases built from the public-transaction records (Table V). We also perform manual verification to ensure the detected vulnerabilities are eliminated. Overall, VFIX successfully generates 565 patches for 601 vulnerabilities, achieving a very high success rate (94%). These results indicate that the combination of template-based fix patterns and static program analysis is very effective in generating successful patches.

Patch Statistics: For the real contracts, on average, VFIX needs to change 3.7, 17.47, 3.71, 2.26, and 12.71 lines of code to patch a contract with *Reentrancy* vulnerabilities, *LockedEther* vulnerabilities, *MissingInputValidation* vulnerabilities, *UnhandledException* vulnerabilities, and multiple types of vulnerabilities (*i.e., Mixed*). Without considering the type of vulnerabilities, VFIX needs to averagely change 7.97 lines of code to patch a contract. For the contracts with injected vulnerabilities, VFIX needs to change 4 lines of code to patch the *Reentrancy* vulnerabilities.

Transaction Replay: To show that the patch preserves the original functionality of the smart contracts, we chose 25 contracts (top 5 contracts that have the most transactions in each vulnerable type) and crawled 5000 transactions for each of them ($125,000$ transactions in total). We then deployed their original version and the patched version in the local testing environment (*e.g.,* Ganache), and issued transactions to execute the contracts using the crawled transaction data. Table V shows the results of the transaction replay. Column "Status diff" shows the number of different result states.

Column "Gas Diff" shows the average gas usage differences. We can see the number of contracts that have different result states are all 0. This result shows that *the patches generated by* vFix *preserves the contracts' original functionality*. The results also show that gas usage has only a slight increase (maximum being 188.84). As each unit of gas equals to $10E-9$ Ethers [63], which is $\sim \$0.000027$, *the cost of the extra gas is negligible*.

Failed Patches: For real contacts, vFix fails to generate patches for 2 *Reentrancy* vulnerabilities, which is mainly due to the stack depth limit of EVM. When fixing *Reentrancy* vulnerability, vFix usually needs to create temporary variables. While not often, such behavior may trigger the EVM exceptions about stack depth, causing the patches not to pass the validation. This case may be improved by requesting the developers to limit their call stack, which will also help defend stack depth attack [64]. The main reason why fixing *MissingInputValidation* and *LockedEther* may fail lies in the limitation of our source code generator, which adopts the pre-order traversal of the patched AST to generate patches. However, when a method call is used as the argument for a function modifier, the generated source code will have syntax errors. This case can be fixed by improving the code generation mechanism.

For contracts with injected vulnerabilities, vFix fails to generate patches for 30 contracts as our dataflow analysis considers the injected vulnerabilities cannot be fixed by moving statements. But *all these* 30 *contracts can be fixed by* vFix *by adopting the global bool value as a lock to limit the method invocations*.

Example Patched Contract: We select one real contract with a *Reentrancy* vulnerability to describe our patch:

```
+ var totalUnreleasedTokens_temp = totalUnreleasedTokens.
+ vestingSchedule.principleLockAmount =
     _principleLockAmount;
+ vestingSchedule.bonusLockAmount = _bonusLockAmount;
+ vestingSchedule.isPrincipleReleased = false;
+ vestingSchedule.isBonusReleased = false;
+ totalUnreleasedTokens = safeAdd(totalUnreleasedTokens,
          _totalAmount);
+ vestingSchedule.amountReleased = 0;
+ require(token.balanceOf(this) >= safeAdd(
     totalUnreleasedTokens_temp, _totalAmount));

- require(token.balanceOf(this) >=
          safeAdd(totalUnreleasedTokens, _totalAmount));
- vestingSchedule.principleLockAmount =
     _principleLockAmount;
- vestingSchedule.bonusLockAmount = _bonusLockAmount;
- vestingSchedule.isPrincipleReleased = false;
- vestingSchedule.isBonusReleased = false;
- totalUnreleasedTokens = safeAdd(totalUnreleasedTokens,
          _totalAmount);
- vestingSchedule.amountReleased = 0;
```

Based on the data-flow analysis, vFix finds that there is a WAR dependence between *safeAdd()* and the writes to *totalUnreleasedTokens*. In this case, vFix generates a patch that saves *totalUnreleasedTokens* before *safeAdd()* by creating a temporary variable, replaces the parameters of *safeAdd()* with the temporary variable, and moves all the writes ahead.

TABLE V: Execution results about transaction replay

Type	Status Diff.	Gas Diff.
Reentrancy	0	53.38
MissinputValidation	0	46.99
LockedEther	0	7.36
UnhandledException	0	27.68
Mixed	0	188.84

Comparison to Existing Works:. As there are no existing works that combine detection, patching, and verification as vFix does, we cannot directly compare vFix with existing works. Thus, we compare only the patching step in our evaluations. As shown in Table I, vFix provides fix patterns (global lock for *Reentrancy* and owner check for *LockedEther*) that EVMPatch [65] and sGuard [22] can support. The results show that vFix can effectively generate these patches (91% for *Reentrancy* and 99% for *LockedEther*) as the existing works without compromising expected behaviors. Beyond that, vFix supports moving statement to fix *Reentrancy* vulnerabilities with a much lower gas cost than sGuard (5 v.s. 25000). EVMPatch is not released publicly and cannot be directly compared. Nonetheless, it cannot support this fix pattern as binary code loses the source code semantics and requires extra data analysis to move the statements.

C. RQ2: Ensemble of Static Verification Tools

In this RQ, we evaluate the effectiveness of vFix's ensemble of multiple static verification tools. Table II shows the vulnerability detected by different static verification tools by vFix. Column *Majority* shows the number of vulnerability confirmed by at least two static verification tools except for *MissInputValidation*, because only *Securify* supports the detection of this vulnerability. The results show that our post-processing filters out the candidates that cannot be fixed (Section IV-A). For example, *MissingInputValidation* reports 795 vulnerabilities and only 131 of them are related to address types. We further manually examine the detected vulnerabilities by the majority voting, and confirm all of them are true positives, indicating the effectiveness of majority voting in improving the detection performance. For example, *Slither* reports 129 *UnhandledException* vulnerabilities, while the majority voting confirms 60 out of them. Similarly, *Slither* misses 37 *LockedEther* vulnerabilities, but the combination of *Securify* and *SmartCheck* finds these 37 vulnerabilities.

We can also see the combination of post-processing and majority voting addresses the limitations of using only one static verification tool. For example, some security properties used by *Securify* are too general: for *Reentrancy* vulnerabilities, *Securify*'s property detects all the writes to storage after an external method call; however, if another external method call is used to determine the execution of the writes to storage, a false positive is reported. Based on the results, *Securify* reports 26 false positives, which is first reduced by the post-processing to 3 and then by the majority voting to 0. These results demonstrate that static-verification and post-processing greatly improve the precision of the vulnerability detection, making it feasible and practical to support the patch generation.

D. RQ3: Runtime Performance

To understand the performance of VFIX, we measure the execution time of VFIX's three phases. We exclude the validation using Truffle since it requires manual interactions (*e.g.,* sending transactions). The results show that VFIX takes $1159.58s$ to finish the whole process. Without considering the time needed by static verification (*i.e.,* detection and validation), VFIX only takes $3.75s$ to fix a contract on average, indicating that VFIX's light-weight program analysis and patch generation based on template-based fix patterns are very efficient.

VI. Discussion

Static Verification: Static verification techniques employ sound analysis that produces no false positives but can produce false positives, such as Securify [18]. VFIX addresses this issue by leveraging majority voting to ensemble multiple static verification tools and employs post-processing to filter out vulnerabilities not amenable for automatic fix. Alternatively, more precise static verification with more flexible security properties can be used, but this direction requires further research efforts and is out of the scope of this paper.

Generalization of Fix Patterns: The infrastructure of VFIX can easily enable other fix patterns as long as data/control-flow analysis and type hierarchy analysis are sufficient to produce the required repair actions. For example, developers may revise our fix patterns for *MissingInputValidation* to include more data types with valid range checking.

Threat To Validity: We mitigate the *major external threat* of dataset representativeness by using datasets of both real vulnerabilities and injected vulnerabilities. In future work, we plan to collect more types of vulnerabilities that can be supported with more provided fix patterns. The *major internal threat* is the human errors in constructing ground truths and verifying the patches. We mitigate this threat through multi-user inspection and rigorous testing on fixed contracts.

Limitations: VFIX cannot fix vulnerabilities whose exploits rely on the mechanisms of the underlying blockchain platform, such as transaction-executing order (*e.g., TODAmount* and *TODReceiver* vulnerabilities) and timestamp. Existing research on blockchain security [66], [67], [68], [69] can be adopted to address these vulnerabilities, being beyond the scope of this paper. VFIX can easily integrate other fix solutions as patch templates, such as the mutex for reentrancy. Currently, we do not use mutex because this solution will cost more gas (*i.e.,* execution fee) than our employed template (*i.e.,* reordering statements).

VII. Related Work

Automatic Program Repair: Some previous research [70], [31], [35], [29], [32] focuses on the code that is executed for negative test cases, and then produces modifications to a program, including deleting a statement and inserting a statement found in the program. Treating all the modifications of a program as a search space, the research adapts search algorithms such as genetic algorithms to generate patches, facing significant challenges in search space explosion [34], [35]. Some other research [26], [27] uses constraint solving to find correct expressions to replace incorrect or vulnerable expressions in a program. Compared with the preceding previous research, VFIX does *not* rely on the test suite that triggers a failure of the target program to generate a patch. Also, VFIX conducts static verification and rule-based checking to detect vulnerabilities and includes program analysis techniques to enable complex fix patterns.

Patch Generation for Smart Contracts: Recently, researchers developed multiple tools to fix vulnerable smart contracts automatically. Rodler *et al.* and Torres *et al.* [65], [71] propose techniques to patch vulnerable smart contracts at the binary code level. Unlike VFIX, which can support a diverse set of fix patterns by leveraging the source code semantics, binary-level templates lack source code semantics and can support only a subset of VFIX's fix patterns such as fix patterns for uncaught exceptions. Nguyen *et al.* [22] propose sGUARD to detect vulnerable smart contracts and fix them (*e.g.,* adding global locks to fix reentrancy vulnerabilities). Compared with sGUARD, VFIX avoids the heavy dependency on a single detector and can be easily extended to support more types of vulnerabilities. Yu *et al.* [72] propose a search-based approach among the mutations of a buggy contract to find a patched contract. As the approach searches only the mutations of the original contract, it cannot support fix patterns that require additional code as VFIX does.

Template-Based Code Generation: Template-based code generation [38] is a widely used approach in program repair and other software engineering tasks. Kyle *et al.* [73] propose a template-based approach to help developers understand program refactoring. Sakamoto *et al.* [74] propose a template-based, test-case generator for web applications. VFIX is inspired by these approaches and provides template-based fix patterns whose parameter values are filled using the context information inferred by our program analysis techniques tailored to smart contracts.

VIII. Conclusion

In this paper, we have presented a novel framework named VFIX, which facilitates software maintenance of smart contracts by automatically generates source code patches to fix detected vulnerabilities. VFIX innovatively combines template-based fix patterns with static verification, providing a program analysis framework that can support multiple fix patterns and can prove the elimination of the vulnerabilities in the patched contract. Our evaluations on vulnerable contracts from a widely used dataset and real-world vulnerable contracts have shown that VFIX can successfully fix 94% of the detected vulnerabilities.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system."

[2] V. Buterin, "Ethereum: a next generation smart contract and decentral- ized application platform," 2013, https://github.com/ethereum/wiki/wiki/White-Paper.

[3] "Blockchain in financial services," 2021, https://consensys.net/blockchain-use-cases/finance/.

[4] "Blockchain casino games," 2021, https://blockchain-casino-games.com/.

[5] "Blockchain in digital identity," 2021, https://consensys.net/blockchain-use-cases/digital-identity/.

[6] "Solidity," 2021, https://github.com/ethereum/solidity.

[7] "Ethereum platform," 2021, https://ethereum.org/.

[8] "King of ether," 2021, https://github.com/kieranelby/KingOfTheEtherThrone\/blob/master/contracts/KingOfTheEtherThrone.sol.

[9] "Accidental' bug may have frozen $280 million worth of digital coin ether in a cryptocurrency wallet," 2017, https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[10] "How to find $10m just by reading the blockchain," 2021, https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95.

[11] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *Proceedings of the International Conference on Principles of Security and Trust (POST)*, 2017, pp. 164–186.

[12] "The dao attack," 2021, https://www2.deloitte.com/ie/en/pages/technology/articles/DAO-Attack-Analysis.html.

[13] "Security alert," 2017, https://www.parity.io/security-alert/.

[14] "TheDAO," 2021, https://etherscan.io/token/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[15] "Reentrancy," 2021, https://swcregistry.io/docs/SWC-107.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 254–269.

[17] "Mythril," 2021, https://github.com/ConsenSys/mythril.

[18] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 67–82.

[19] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018, pp. 653–663.

[20] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2018, pp. 1–12.

[21] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2019, pp. 1–15.

[22] T. D. Nguyen, L. H. Pham, and J. Sun, "sguard: Towards fixing vulnerable smart contracts automatically," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2021, pp. 1215–1229.

[23] V. C. Bui, S. Wen, J. Yu, X. Xia, M. S. Haghighi, and Y. Xiang, "Evaluating upgradable smart contract," in *2021 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2021, pp. 252–256.

[24] "Ethereum smart contract best practices," 2021, https://consensys.github.io/smart-contract-best-practices/.

[25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 54–72, 2011.

[26] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[27] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[28] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the Annual ACM SIGPLAN-SIGACT symposium on principles of programming languages (POPL)*, 2016, pp. 298–312.

[29] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, p. 789–799.

[30] R. van Tonder and C. Le Goues, "Static automated program repair for heap properties," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, pp. 151–162.

[31] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012, pp. 3–13.

[32] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPOLOS)*, 2013, p. 317–328.

[33] S. So and H. Oh, "Smartfix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models," in *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023, p. 185–197.

[34] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software Quality Journal*, vol. 21, no. 3, pp. 421–443, 2013.

[35] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2018, p. 1–11.

[36] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019, pp. 8–15.

[37] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[38] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, D. Zhang and A. Møller, Eds., 2019, pp. 31–42.

[39] "vfix source code," 2024, https://github.com/vfixresearch/vFix.

[40] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2020, pp. 415–427.

[41] "Etherscan," 2021, https://cn.etherscan.com/.

[42] "Bitcoin script," 2021, https://en.bitcoin.it/wiki/Script.

[43] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 443–458.

[44] "Hyperledger," 2021, https://www.hyperledger.org/.

[45] "Corda," 2021, https://www.corda.net/.

[46] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts." *IACR Cryptology ePrint Archive*, vol. 2016, p. 1007, 2016.

[47] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: fuzzing smart contracts for vulnerability detection," in *Proceedings of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 259–269.

[48] "Transaction ordering dependency," 2021, https://consensys.github.io/smart-contract-best-practices/known_-attacks/.

[49] "Missing zero check," 2021, https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation.

[50] "Lock ether," 2021, https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether.

[51] "Swc 104," 2021, https://swcregistry.io/docs/SWC-104.

[52] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, *Compilers principles, techniques & tools*. pearson Education, 2007.

[53] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. springer, 2015.

[54] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: hybrid program analysis for determinism," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 463–474.

[55] X. Xiao, N. Tillmann, M. Fahndrich, J. De Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 2012, pp. 80–89.

[56] B.-C. Cheng and W.-M. W. Hwu, "Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2000, p. 57–69.

[57] F. Bond, "A solidity parser for js built on top of a robust antlr4 grammar," 2019, https://github.com/federicobond/solidity-parser-antlr.

[58] T. Parr, "ANTLR," 2014, http://www.antlr.org/.

[59] "Google bigquery," 2021, https://cloud.google.com/bigquery/.

[60] "Tuffle," 2021, https://www.trufflesuite.com/.

[61] K. Taneja, N. Li, M. R. Marri, T. Xie, and N. Tillmann, "Mitv: multiple-implementation testing of user-input validators for web applications," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010, pp. 131–134.

[62] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[63] "Gas and fees," 2021, https://ethereum.org/en/developers/docs/gas/.

[64] Ethereum, "Solidity security considerations," 2019, https://solidity.readthedocs.io/en/v0.6.1/security-considerations.html.

[65] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Evmpatch: timely and automated patching of ethereum smart contracts," in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021, pp. 1289–1306.

[66] G. O. Karame and E. Androulaki, *Bitcoin and blockchain security*. Artech House, 2016.

[67] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 185–200.

[68] E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via pvorm," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 701–717.

[69] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2016, pp. 839–858.

[70] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[71] C. Ferreira Torres, H. Jonker, and R. State, "Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2022, pp. 115–128.

[72] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–32, 2020.

[73] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[74] K. Sakamoto, K. Tomohiro, D. Hamura, H. Washizaki, and Y. Fukazawa, "Pogen: a test code generator based on template variable coverage in gray-box integration testing for web applications," in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2013, pp. 343–358.