

# SMARTFIX: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models

Sunbeom So\*  
Korea University  
Republic of Korea  
sunbeom\_so@korea.ac.kr

Hakjoo Oh†  
Korea University  
Republic of Korea  
hakjoo\_oh@korea.ac.kr

## ABSTRACT

We present SMARTFIX, a new technique for repairing vulnerable smart contracts. There is an urgent need to develop automatic bug-repair techniques for smart contracts, as smart contracts are safety-critical software and manual debugging is burdensome and error-prone. While several repair approaches have been proposed recently, they are unsatisfactory since no existing techniques can achieve high repairability, full automation, and safety guarantee at the same time, posing significant problems for practical use. SMARTFIX aims to address these shortcomings by using a “generate-and-verify” approach that iteratively enumerates candidate patches while validating their correctness by invoking a safety verifier. However, in this approach, a technical challenge arises as the search space is huge and the verification-based patch validation is expensive. To address this challenge, we present a novel technique for accelerating the generate-and-verify repair procedure using statistical models derived from the verifier’s feedback. Experimental results on real-world Ethereum smart contracts show that SMARTFIX is able to achieve a fix success rate of 94.8% for critical classes of vulnerabilities, far outperforming sGUARD, the existing state-of-the-art technique whose success rate is 65.4%.

## CCS CONCEPTS

• Security and privacy → Software security engineering; • Software and its engineering → Software testing and debugging.

## KEYWORDS

smart contract, generate-and-verify repair, statistical model

### ACM Reference Format:

Sunbeom So and Hakjoo Oh. 2023. SMARTFIX: Fixing Vulnerable Smart Contracts by Accelerating Generate-and-Verify Repair using Statistical Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’23)*, December 3–9, 2023, San Francisco, CA, USA. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3611643.3616341>

\*Current affiliation: Gwangju Institute of Science and Technology (GIST)

†Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0327-0/23/12...\$15.00

<https://doi.org/10.1145/3611643.3616341>

## 1 INTRODUCTION

Ensuring the safety of smart contracts is vital for trustworthy blockchain ecosystems. Blockchain is a decentralized database in which stored data is transparent and unchangeable. Smart contracts are programs running on blockchain whose primary means is to automatically fulfill obligations between untrusted parties. Inheriting advantages of blockchain such as transparency, smart contracts have received much attention from various domains such as digital assets and supply chain [26]. Unfortunately, however, considerable safety concerns remain for smart contracts; because smart contracts often manage valuable assets, they easily become targets of malicious users and, once exploited, even a single flaw can cause huge financial damage (e.g., [2, 8]). To make matters worse, smart contracts are immutable and hence their flaws cannot be fixed. It is therefore crucially important to develop techniques for improving the safety of smart contracts before deployment.

In this paper, we present SMARTFIX, a new technique for automatically fixing vulnerable smart contracts. The last few years have witnessed a large number of techniques for finding bugs in smart contracts (e.g., [4, 10, 11, 13, 16, 20, 22, 24, 28, 31, 33, 40, 42, 44, 45, 47, 49, 50, 52–54]). Although they are useful for identifying safety issues, manually fixing bugs detected by those safety analyzers remains a time-consuming and error-prone task. Our aim is to reduce this burden with an automated technique that can safely and accurately fix vulnerable smart contracts before they get deployed on blockchain.

**Existing Approaches.** Recently, a few techniques have been proposed to repair vulnerable smart contracts [21, 43, 56, 57] but they are insufficient for practical use. This is because most of the existing approaches [21, 43, 57] rely on a single repair strategy for each bug type and hence fails to fix diverse patterns of bugs (e.g., Section 2). An exception is SCREPAIR [56], a test-based repair technique that supports multiple repair strategies for each bug type. However, SCREPAIR fails to achieve full automation; it requires users to provide test cases (i.e., transaction sequences with concrete arguments for each transaction [24, 49]) for validating candidate patches, but manually constructing robust test suites is nontrivial. Furthermore, SCREPAIR does not guarantee the safety of generated patches because test cases are typically incomplete as repair specification in practice [35, 48].

**Our Approach.** Unlike existing techniques, SMARTFIX aims to achieve high repairability, full automation, and safety guarantee all at once. To achieve this goal, we present a new bug-repair technique that combines a “generate-and-verify” repair procedure with statistical models. Figure 1 illustrates our approach. SMARTFIX starts

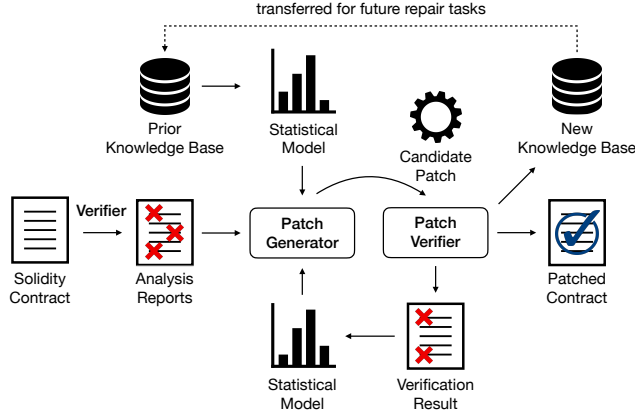


Figure 1: Overview of SMARTFix.

by automatically obtaining analysis reports for a given smart contract using a verifier (Section 3). Given the smart contract to repair and its analysis reports, SMARTFix basically alternates between the “generate” and “verify” phases to fix diverse patterns of security bugs automatically and safely; it iteratively performs enumerative search for exploring various candidate patches and verification-based patch validation for ensuring the safety of output patches. This approach, however, poses a technical challenge to repair efficiency, as the search space for candidate patches is huge and invoking a verifier is expensive. To address this challenge, we propose a machine learning-based technique that intelligently guides the repair procedure using statistical models. We build statistical models both online and offline; we construct a model online from the verifier’s feedback (verification results such as the number of alarms) on candidate patches for the contract under repair, and we also derive a model offline from a prior knowledge base (abstract forms of verification results obtained during the repair of other contracts). By using learned statistical models to prioritize candidate patches that are helpful in finding desired safe contracts, we can greatly improve the performance of the repair procedure.

**Results.** Experimental results show that our technique is highly effective at fixing security bugs in smart contracts. We implemented our approach in a tool SMARTFix, which generates patches at source code level. SMARTFix currently targets contracts written in Solidity [6], the most popular programming language for writing Ethereum smart contracts. We extensively evaluated SMARTFix on four datasets. These datasets contain five types (Section 3) of known security bugs that are arguably the most common yet critical ones in Ethereum smart contracts: integer over/underflow, ether-leak, suicidal, reentrancy, and improper use of `tx.origin`. In comparison with sGUARD [43], the state-of-the-art bug-fixing tool for smart contracts, the fix success rate of SMARTFix was 94.8% on commonly supported classes of bugs, whereas the fix success rate of sGUARD was 65.4%. We also show that our learning-based technique is critical for high performance. Compared to the baseline algorithm that explores candidate patches in order of increasing size without using learned statistical models, our techniques collectively improved the repair effectiveness by 56.7% in terms of generating bug-free contracts.

```

1  function transferFrom (address from, address to, uint value)
2      public returns (bool success) {
3      if (value==0) return false;
4      uint fromBalance = balance[from];
5      uint allowance = allowed[from][msg.sender];
6      (-) bool sufficientFunds = fromBalance <= value;
7      (-) bool sufficientAllowance = allowance <= value;
8      (-) bool overflowed = balance[to]+value > balance[to];
9      (+) bool sufficientFunds = fromBalance >= value;
10     (+) bool sufficientAllowance = allowance >= value;
11     (+) bool overflowed = balance[to]+value < balance[to];
12     if (sufficientFunds && sufficientAllowance
13         && !overflowed) {
14         balance[to] += value;
15         balance[from] -= value;
16         allowed[from][msg.sender] -= value;
17         return true;
18     }
19     else {return false;}
20 }

```

Figure 2: A vulnerable function simplified from DimonCoin (CVE-2018-11411). (-) indicates buggy lines in original source code. (+) indicates patches produced by SMARTFix.

**Contributions.** We make the following contributions.

- We present an effective approach for automatically fixing vulnerable smart contracts. The key enabling technology is a learning-based method for accelerating the generate-and-verify procedure with statistical models.
- We conduct comprehensive evaluation of SMARTFix in comparison with sGUARD [43], the state-of-the-art repair tool for smart contracts, using four datasets that contain five types of annotated security bugs.
- For open science, we make the implementation of SMARTFix open-sourced and datasets publicly available (Section 8).

## 2 MOTIVATING EXAMPLE

We demonstrate usefulness of SMARTFix with an example.

Figure 2 shows a buggy implementation of the `transferFrom` function in the DimonCoin contract (CVE-2018-11411). The job of `transferFrom` is to allow an agent (`msg.sender`) to transfer tokens on behalf of an original token holder (`from`). The core functionality is implemented at lines 10–12; when the agent (the transaction sender, `msg.sender`) invokes the function, the agent’s allowance (`allowed[from][msg.sender]`) and the original token holder’s balance (`balance[from]`) decrease by value tokens (lines 11, 12), and the recipient’s balance (`balance[to]`) increases by the same amount of tokens (line 10). Observe that, due to flaws in conditional expressions at lines 5–7, integer over/underflow bugs can happen at lines 10–12, which can result in improper state changes. For example, contrary to what the variable name `sufficientFunds` indicates, the flawed condition allows `from` to send tokens when `from`’s balance (`balance[from]`) is insufficient.

Existing techniques for repairing smart contracts [21, 43, 56, 57] do not work well on this example. For example, approaches [21, 43, 57] that resort to a single repair strategy (i.e., inserting runtime bound checks) could generate safe but functionally incorrect patches, failing to eliminate root causes of bugs. Indeed, sGUARD [43]

produces the following incorrect patch,<sup>1</sup> which replaces all arithmetic operators by bound checks (i.e., + by `safeAdd` and - by `safeSub`) that raise exceptions when over/underflows occur at runtime:

```

2  if (value==0) return false; // value!=0 holds afterwards
...
7  (-) bool overflowed = balance[to] + value > balance[to];
7' (+) bool overflowed = safeAdd(balance[to], value) > balance[to];
8  if (sufficientFunds && sufficientAllowance
9      && !overflowed) { // deadcode: if-branch
10 (-) balance[to] += value;
11 (-) balance[from] -= value;
12 (-) allowed[from][msg.sender] -= value;
10' (+) balance[to] = safeAdd(balance[to], value);
11' (+) balance[from] = safeSub(balance[from], value);
12' (+) allowed[from][msg.sender] = safeSub(allowed[from][msg.sender], value);
13  return true;
14 } ...
    
```

where line 7 is changed to line 7' and lines 10–12 are changed to lines 10'–12', respectively. While this patch ensures that the fixed parts are free of integer over/underflow bugs, it does not provide proper functionalities, making the function unusable. For example, the patch introduces deadcode in the if-branch due to the two conflicting conditions: `value != 0` and `value = 0`. The former is imposed by line 2. The latter implicitly holds by the combinations of the two: `balance[to] + value >= balance[to]` (imposed by `safeAdd` at line 7' in the above) and `balance[to] + value <= balance[to]` (!overflowed at line 9 in the above). SCREPAIR [56] is also unsatisfactory here; while SCREPAIR would be able to generate candidates that change comparison operators as it supports multiple repair templates, it imposes the hard burden on users of building rigorous test suites for validating the safety of candidate patches. Moreover, it could generate unsafe patches when tests are incomplete.

SMARTFIX aims to address the shortcomings of existing techniques, by a new generate-and-verify approach guided by statistical models. In this example, SMARTFIX accurately fixes the bugs within reasonable time; while iterating the repair-loop, it tries changing comparison operators in lines 5–7 to the ones in lines 5'–7' (Figure 2), and outputs the patch after verifying its safety. Moreover, SMARTFIX can reject the incorrect patch made by sGUARD through verification-based patch validation (Section 3.1.2).

### 3 REPAIR ALGORITHM

In this section, we present the algorithm of SMARTFIX. Section 3.1 describes the basic “generate-and-verify” approach, which iteratively generates candidate patches and verifies their safety. Section 3.2 explains how we accelerate this basic approach using statistical models learned during the iterative process.

**Vulnerability- and Regression Reports.** Given a smart contract  $s$ , we assume we have two types of reports for it: vulnerability report ( $VR$ ) and regression report ( $RR$ ). These reports can be obtained by our patch verifier (Section 3.1.2) during the algorithm.

- $VR$  is a set of 4-tuples  $(v, x, l, es)$ , where  $v \in V$  is a vulnerability type,  $x$  is the signature of the function that contains the

<sup>1</sup>This patch was obtained after we modified the original contract, as sGUARD generated an execution error (heap out of memory) for the original contract. To make sGUARD produce patches somehow, we applied two changes to the contract: (1) removing a loop in the contract (to prevent the runtime error from happening), and (2) inserting an external function call that has dependencies on arithmetic operations (sGUARD does not generate any patches even after applying (1)—see Section 5.2).

**Algorithm 1** Basic algorithm of SMARTFIX, and its enhanced version based on machine learning (lines 6 and 13)

**Input:** A smart contract  $s_0$

**Output:** A patched contract  $s$ , a knowledge base  $K_{on}$

```

1: ( $VR_0, RR_0$ )  $\leftarrow$  VERIFY( $s_0$ )
2: ( $D_{on}, K_{on}$ )  $\leftarrow$  ( $\emptyset, \emptyset$ )
3: ( $s^*, VR^*, p^*$ )  $\leftarrow$  ( $\perp, VR_0, \epsilon$ )
4:  $C \leftarrow$  EXTRACT( $s_0, VR_0$ ) ▷ §3.1.1
5:  $W \leftarrow$  GENERATE( $C, s_0, \epsilon$ ) ▷ §3.1.1
6: (+) Derive  $M_{off}$  from prior knowledge  $K_{off}$  ▷ §3.2.2
7: repeat
8:   ( $s, p$ )  $\leftarrow$  argmin $(s,p) \in W$  cost( $p$ )
9:    $W \leftarrow W \setminus \{(s, p)\}$ 
10:  ( $VR, RR$ )  $\leftarrow$  VERIFY( $s$ ) ▷ §3.1.2
11:  if BetterCandidate( $(RR, RR_0), (VR, VR^*), (p, p^*)$ ) then
12:    ( $s^*, VR^*, p^*$ )  $\leftarrow$  ( $s, VR, p$ )
13:  (+) ( $D_{on}, K_{on}, M_{on}, M_{off}$ )  $\leftarrow$  Run Algorithm 2
14:   $W \leftarrow W \cup$  GENERATE( $C, s_0, p$ ) ▷ §3.1.1
15:  if  $|VR^*| = 0$  then ▷  $s^*$  is a solution contract
16:     $W \leftarrow \{(s, p) \mid (s, p) \in W, |p| < |p^*|\}$ 
17: until  $W = \emptyset$  or timeout
18: return ( $s^*, K_{on}$ )
    
```

vulnerability,  $l$  is the source code line where the vulnerability is detected, and  $es$  is the expression (or the statement) related to the vulnerability detected at  $l$ . We write  $|VR|$  to denote the number of alarms reported for  $s$ .

In this paper,  $v$  is one of the five bug-types that are most common and critical in Solidity smart contracts: IO (integer over/underflow), RE (reentrancy; vulnerabilities due to unexpected reentrancy from external contracts), EL (ether-leak; hijacking of ethers in contracts), SU (suicidal; deactivation of contracts by unauthorized users), and TX (improper use of `tx.origin` in user authentication).

- $RR$  is a set of pairs  $(r, x)$ , where  $r \in R$  is a functional regression type (Section 3.1.2) and  $x$  is the signature of the function where the potential regression is detected.

**Goal.** Given a contract  $s_0$  to repair, our goal is to transform  $s_0$  into a new contract  $s$  proven to be free of bugs, which we call a *solution contract*. Specifically, it should satisfy  $|VR| = 0$  and  $RR = RR_0$ .

- $|VR| = 0$ : means that the absence of bugs in  $s$  is proven by the patch verifier (Section 3.1.2) based on formal verification.
- $RR = RR_0$ : means  $s$  is non-regressive, in that existing potentially abnormal behaviors of  $s_0$  are preserved and no new abnormal behaviors are detected in  $s$ .

#### 3.1 Basic Generate-and-Verify Repair

Algorithm 1 shows the basic “generate-and-verify” architecture of SMARTFIX, which iteratively searches for solution contracts. The input is a potentially vulnerable smart contract  $s_0$ . The outputs are a (partially) patched contract  $s$ , and a new knowledge base  $K_{on}$  constructed during the repair of  $s_0$ . For now, we assume that the lines highlighted by (+) are ignored in Algorithm 1, which are

procedures for accelerating the overall repair process using learned statistical models (Section 3.2).

In the preparation phase (lines 1–6), we first run the verifier (VERIFY) to obtain the vulnerability- and regression reports for  $s_0$  (line 1). At line 2, we initialize  $D_{on}$  and  $K_{on}$  with the empty sets;  $D_{on}$  is a training dataset for deriving a statistical model online (during the repair-loop), and  $K_{on}$  is the new knowledge base consisting of the abstract form of the verifier’s feedback (Section 3.2.2). At line 3, we initialize  $s^*$  with  $\perp$  (null),  $VR^*$  with  $VR_0$ , and  $p^*$  with  $\epsilon$  (the empty sequence that indicates the null value of a patch), respectively. Here,  $s^*$  will be a patched contract that is best so far in terms of safety and patch size,  $VR^*$  is a vulnerability report for the best contract  $s^*$  (or  $VR_0$  if  $s^*$  does not exist), and  $p^*$  is a patch applied to  $s^*$ . At line 4, we extract patch components  $C$ , which are the basic elements of patches. At line 5, we generate initial candidate patches using  $C$ , and initialize the workset  $W$  with them. During the repair process,  $W$  stores candidate patches that will be examined. More precisely,  $W$  maintains a set of pairs  $(s, p)$ , where  $p$  is a candidate patch and  $s$  is a candidate contract obtained by applying  $p$  to the original contract  $s_0$ . Given a candidate  $(s, p)$ , if  $s$  is a contract that has been added to  $W$ , we assume that  $(s, p)$  is not accumulated to  $W$  to avoid redundant attempts.

In the generate-and-verify repair-loop (lines 7–17), we first pick the candidate  $(s, p)$  with the least cost (line 8) and remove it from  $W$  (line 9). In the baseline algorithm, the function cost prefers small-sized patches:  $\text{cost}(p) = |p|$ . In Section 3.2, we will enhance this size-based cost function, as it is insufficient for making SMART-Fix practical. At line 10, we validate  $s$ , where the candidate patch  $p$  is applied, by invoking the patch verifier. If  $p$  is a “better” candidate patch than  $p^*$  (line 11) in terms of safety or patch size, we replace  $p^*$  by  $p$  and update other related data similarly (line 12). The predicate BetterCandidate at line 11 is defined as follows:

$$\begin{aligned} \text{BetterCandidate}((RR, RR_0), (VR, VR^*), (p, p^*)) &\iff \\ (RR = RR_0 \wedge |VR| < |VR^*|) & \\ \vee (RR = RR_0 \wedge |VR| = |VR^*| \wedge |p| < |p^*|) & \end{aligned}$$

That is, a patch  $p$  is better than  $p^*$  if:  $p$  is non-regressive and safer (the first case), or  $p$  is non-regressive, equally safe, and simpler (the second case). At line 14, we add new candidates to  $W$ . If  $s^*$  is a solution contract (line 15), we eliminate patches larger than  $p^*$  to find simpler solution contracts (line 16). The repair-loop repeats until the workset becomes empty or a timeout occurs (line 17).

**3.1.1 Patch Generation.** We explain how to extract patch components  $C$  (EXTRACT – line 4 in Algorithm 1) and generate patches  $P$  (GENERATE – lines 5, 14).

**Repair Template.** An atomic repair template  $a \in A$  is the basic unit of a patch component and a patch. We use the following atomic repair templates to fix the five types of bugs.

- **Insert( $l, s$ ):** inserts a statement  $s$  in front of line  $l$ .
- **Rep( $l, e1, e2$ )** replaces expressions  $e1$  at  $l$  with  $e2$ .
- **AddM( $x, m$ )** adds a modifier whose name is  $m$  to a function whose signature is  $x$ .
- **Move( $l, \{l_1, \dots, l_n\}$ )** moves statements at  $l_1, \dots, l_n$  ahead of the statement at  $l$ .
- **ToCnstr( $x$ )** replaces a function, whose signature is  $x$ , by a constructor.

- **ElseRevert( $l$ )** inserts `else {revert ();}` at  $l$ , where  $l$  is the line at which an if-statement without else-branches ends.

We devised these templates by carefully studying typical causes of security bugs in smart contracts. Insert, Rep, AddM, Move are templates that are commonly used to fix vulnerabilities in smart contracts [21, 43, 56, 57]. In addition, we included the ToCnstr template to fix EL and SU vulnerabilities whose root cause is a faulty access control due to mistakenly named constructors (e.g., [33]). We also included ElseRevert as an auxiliary template that is used in combination with Move to fix RE bugs, as we observed cases where putting the two templates together is essential for fixing RE bugs safely and correctly.<sup>2</sup>

**Component Extraction.** A patch component  $c \in C$  is a sequence of atomic repair templates (i.e.,  $c = a_1 \dots a_n$ ). While most components are single repair templates, to accelerate the repair process, we also use components with multiple templates for several special cases (e.g., fixing RE bugs often requires applying multiple nonReentrant modifiers [21, 43]).

Given an original contract  $s_0$  and its vulnerability report  $VR_0$ , EXTRACT( $s_0, VR_0$ ) outputs  $C = C_{IO} \cup C_{RE} \cup C_{EL} \cup C_{SU} \cup C_{TX}$ , where each  $C_v$  is a set of patch components for fixing bugs whose type is  $v$ . EXTRACT collects components from  $s_0$ , according to vulnerability types and locations reported in  $VR_0$ . For example, given an alarm  $(IO, -, l, a+b) \in VR_0$  (i.e., an IO bug is detected at  $a+b$  in line  $l$ ),  $C_{IO}$  will include a component that inserts a bound check in front of line  $l$  (i.e., Insert( $l, \text{require}(a+b \geq a)$ )). The component extraction rules for each bug type are explained in the supplementary material.

**Patch Enumeration.** A patch  $p$  is a sequence of patch components (i.e.,  $p = c_1 \dots c_n \in C^*$ ). Given a set of patch components  $C$  and a candidate patch  $p$ , GENERATE returns new candidates as follows: GENERATE( $C, s_0, p$ ) =  $\{(s, p \cdot c) \mid c \in C\}$ . Here,  $p \cdot c$  is a new patch obtained by appending a patch component  $c$  to the old patch  $p$ , and  $s$  is a new contract obtained by applying the new patch to the original contract  $s_0$ .

**3.1.2 Patch Verification.** Given a contract  $s$ , the job of the patch verifier (VERIFY – lines 1 and 10 in Algorithm 1) is to produce  $VR$  and  $RR$  for  $s$ , by performing safety verification and regression detection on  $s$ . We developed the patch verifier on top of VERISmart [9, 50], an open-sourced verification tool for Solidity contracts.

**Safety Verification.** We obtain the vulnerability report  $VR$  from vulnerability-detection results of VERISmart; its latest version [9] supports the verification of the five types of bugs that we target.

SMARTFix guarantees patch safety by design, as we use the patch verifier based on the formal verification method [50] that can prove the absence of bugs; if there are bugs reported from  $s_0$  but not from a patched contract, it is guaranteed that the original bugs from  $s_0$  were safely fixed w.r.t. safety conditions defined in the verifier. For example, SMARTFix outputs the patches in Figure 2, only after the patch verifier has formally proven that the safety condition  $\text{balance}[to] + \text{value} \geq \text{balance}[to]$  is always *true* at line 10 and therefore line 10 is free of IO bugs.

<sup>2</sup>E.g., <https://github.com/smartbugs/smartbugs-curated/blob/main/dataset/reentrancy/0xbaf51e761510c1a11bf48dd87c0307ac8a8c8a4f.sol>



**Regression Detection.** In automatic program repair (APR) research, ensuring patch correctness is an open problem [23], as complete formal specifications on functional correctness are not readily available.

We aim to mitigate this challenge as much as possible by eliminating likely incorrect patches. Specifically, we detect incorrect patches by validating them against predefined regression conditions. For example, suppose a patched function consists of the following statements:

```
require(x <= a); require(x >= a); y := x - a
```

where the underlined statement is a patch to fix the underflow at  $x - a$  in the last statement. Observe that this patch introduces a regression that is absent in the original function; the final statement always assigns the zero value to  $y$ , as the condition  $x = a$  holds due to the improper combinations of the two conditions in the first two statements:  $x \leq a$  and  $x \geq a$ . To detect this regression, we insert the assertion statement `assert (y==0)` after the last assignment during the preprocessing step:

```
require(x <= a); require(x >= a); y := x - a; assert(y == 0)
```

Here,  $y==0$  is a regression condition to see whether  $y$  always gets assigned 0, which would be typically an abnormal behavior. Next, in the regression detection step, we generate the following verification condition and check its validity [12]:

$$x \leq a \wedge x \geq a \wedge y = x - a \rightarrow y = 0.$$

Here, since the condition is valid (i.e., the likely abnormal property always holds), we include this result in  $RR$ , i.e.,  $(f, ZA) \in RR$ , where  $f$  is a signature of the patched function and  $ZA$  denotes the regression type for zero-value assignments. Finally, we decide that the patch is regressive because this behavior exists in the patched function only (i.e.,  $RR \neq RR_0$  where  $RR_0 = \emptyset$ ).

We detect three kinds of incorrect patches, which can appear in our search space due to improper overlaps of conditionals: deadcode (e.g., Section 2), zero-value assignment (e.g., the example above), and effectless assignment (the l-value's state of an assignment never changes). We implemented a regression detector, as a post-analysis procedure invoked after the safety verification. For efficiency, we perform regression detection at program locations likely to be beneficial only. For example, we insert an assertion `assert(false)` after guard-statements to detect deadcode by checking the feasibility of paths, but do not insert it after assignments.

### 3.2 Guiding Repair using Statistical Models

We present the main technical contributions of this paper. We enhance the basic algorithm using statistical models learned at lines 6 and 13 in Algorithm 1. At line 6, we construct a unified prior knowledge base  $K_{off}$  by collecting knowledge bases obtained from past repair tasks, and derive a model  $M_{off}$  offline (before entering the repair-loop) from  $K_{off}$ . The process for line 13 is described in Algorithm 2.

At line 1 in Algorithm 2, we compute *score*, which we call *safety score*, is a real number that quantifies how close  $p$  is to solution patches (patches necessary for generating solution contracts); the definition of the function  $Q$  will be explained soon. At line 2, we generate a training data  $d$  based on the verifier's feedback (*score*),

---

#### Algorithm 2 Learning phase of SMARTFix

---

**Input:** a set of patch components ( $C$ ), verification results ( $VR_0, VR, RR_0, RR$ ), a candidate patch  $p$ , learning-related data ( $D_{on}, K_{on}, M_{on}, M_{off}$ )

**Output:** ( $D_{on}, K_{on}, M_{on}, M_{off}$ )

- 1:  $score \leftarrow Q(p, VR_0, VR, RR_0, RR)$
  - 2:  $D_{on} \leftarrow D_{on} \cup \{d\}$   $\triangleright d = (F_C(p), score), \text{\S 3.2.1}$
  - 3: Derive  $M_{on}$  from  $D_{on}$
  - 4:  $K_{on} \leftarrow K_{on} \cup \{k\}$   $\triangleright k = (\alpha(p), score), \text{\S 3.2.2}$
  - 5: Set  $M_{off}$  to  $\perp$  if  $score < 0$  or a solution contract has not been generated within a predetermined loop bound.
  - 6: **return** ( $D_{on}, K_{on}, M_{on}, M_{off}$ )
- 

and add  $d$  to  $D_{on}$ . At line 3, we derive a model  $M_{on}$  online (during the iteration of the repair-loop) from  $D_{on}$ . At line 4, we create a new knowledge data  $k$  based on *score*, and accumulate  $k$  to  $K_{on}$ . A model  $M_{off}$  derived offline could misguide the repair procedure, because, for example, a contract under repair could have characteristics different from prior contracts. Thus, at line 5, if at least one of the two conditions holds, we consider that the model  $M_{off}$  learned offline is unlikely to be useful in the next iterations, and thus we invalidate  $M_{off}$ ; in the implementation, the pre-set loop bound is 5.

**Quantifying Verifier's Feedback.** Given a candidate patch  $p$ , its verification results ( $VR, RR$ ), and verification results for the original contract ( $VR_0, RR_0$ ), the function  $Q$  computes a safety score (*score*) for  $p$  as follows:

$$Q(p, VR_0, VR, RR_0, RR) = \begin{cases} \frac{|VR_0| - |VR| - \gamma * |p|}{|VR_0|} & \text{if } RR = RR_0 \text{ and } |VR| \leq |VR_0| \\ -(1 + |p|) & \text{if } RR \neq RR_0 \text{ or } |VR| > |VR_0|. \end{cases}$$

In short, the main metric of  $Q$  is the alarm reduction rate against the number of alarms from the original contract. In the first case where  $p$  is a non-regressive ( $RR = RR_0$ ) and equally or more safe ( $|VR| \leq |VR_0|$ ) candidate, we compute a safety score by subtracting the scaled patch size ( $\gamma * |p|$ ) from the reduced number of alarms ( $|VR_0| - |VR|$ ) and normalizing the result with  $|VR_0|$ . Here, subtracting  $\gamma * |p|$  means that we prefer smaller patches. This is because, as in previous studies (e.g., [41, 56]) on automatic program repair, we observed that, when compared to smaller patches that are equally safe, larger patches are likely to contain redundant or incorrect patches. In the current implementation,  $\gamma = 0.5$ . In the second case where  $p$  is a regressive or more unsafe patch, we compute a negative score  $-(1 + |p|)$  for  $p$ .

In the remaining of this section, we explain the following: how to learn models online and offline (Section 3.2.1 and 3.2.2), and how to use learned models (Section 3.2.3).

**3.2.1 Learning a Model Online.** To build a statistical regression model  $M_{on}$  online, we produce a training data  $d \in D_{on}$ :

$$d = (\langle v_1, \dots, v_n \rangle, score).$$

*score* is a safety score for  $p$  (computed at line 1 in Algorithm 2).  $\langle v_1, \dots, v_n \rangle \in \{0, 1\}^n$  is a feature vector representation for  $p$ . Suppose a set of patch components  $C = \{c_1, \dots, c_n\}$  is given, where each component  $c_i$  has a unique identifier  $i \in [1, n]$ . The feature

vector for  $p$  is obtained by converting  $p$  using an encoding function  $F_C$  parameterized by  $C$ :

$$F_C(p) = \langle v_1, \dots, v_n \rangle$$

where  $v_i$  is 1 (resp., 0) if  $p$  contains (resp., does not contain) an  $i$ -th patch component  $c_i$ . Our feature representation method is inspired from [25], which used a similar representation in the context of learning-based program debloating.

Given a training dataset  $D_{on}$  iteratively collected this way,  $M_{on}$  can be derived using an off-the-shelf supervised learning algorithm. In the current implementation, we used the one for learning a linear regression model (Section 4).

**3.2.2 Learning a Model Offline.** We also try to accelerate the repair procedure using prior knowledge. More precisely, given repair templates that were used to fix some vulnerabilities in past repair attempts, we aim to quickly fix similar vulnerabilities using similar templates. We first explain how we build a new knowledge base (line 4 in Algorithm 2), and then describe how we derive a statistical model from an existing knowledge base (line 6 in Algorithm 1).

**Constructing Knowledge.** Given a safety score (*score*) for a candidate patch  $p$  (line 1 in Algorithm 2), we generate a knowledge data  $k = (\hat{p}, \text{score})$ , where  $\hat{p}$  is an abstract form of  $p$  for abstracting away its code-specific information (e.g., line numbers, variable names) that would not be directly applicable to new contracts. We convert a patch  $p = c_{x_1} \dots c_{x_m}$  ( $x_1, \dots, x_m \in [1, n]$ ,  $n = |C|$ ) into an abstract patch  $\hat{p}$  using a function  $\alpha$  (i.e.,  $\alpha(p) = \hat{p}$ ), which is defined as follows:  $\alpha(p) = \alpha'(c_{x_1}) \dots \alpha'(c_{x_m})$ . Here,  $\alpha'$  is a function that returns an abstract patch component for a given patch component  $c = a_1 \dots a_n$  as follows:

$$\alpha'(c) = \begin{cases} \text{Insert}(H(f_l), T_s(s')) & \text{if } c = \text{Insert}(l, s') \\ \text{Rep}(H(f_l), T_e(e_1), \text{Op}(e_2)) & \text{if } c = \text{Rep}(l, e_1, e_2) \\ \text{AddM}(H(f_x), H(m)) & \text{if } c = \text{AddM}(x, m), m \neq \text{nr} \\ \text{NR}(H(f_x)) & \text{if } c = \text{AddM}(x, m), m = \text{nr} \\ \text{Move}(H(f_l)) & \text{if } c = \text{Move}(l, \{l_1, \dots, l_n\}) \\ \text{ToCnstr}(H(f_x)) & \text{if } c = \text{ToCnstr}(x) \\ \text{ElseRevert}(H(f_l)) & \text{if } c = \text{ElseRevert}(l) \\ \alpha'(a_1) \dots \alpha'(a_n) & \text{if } c = a_1 \dots a_n \end{cases}$$

Each abstract component is built by an abstract component constructor. For example,  $\text{NR}$  is a constructor for abstracting components that apply nonReentrant (denoted *nr* in the above) modifiers for fixing RE bugs (e.g., [43]).

In short,  $\alpha'$  abstracts information on patch locations (line numbers, function signatures) into function-level features, and patch expressions/statements into types of variables in them.  $f_l$  is a function that contains line  $l$ , and  $f_x$  is a function whose signature is  $x$ .  $T_s(s')$  (resp.,  $T_e(e')$ ) returns a set of types of global variables that appear in a statement  $s'$  (resp., an expression  $e'$ ).  $\text{Op}(e)$  computes the set of binary operators in  $e$ . Given a function  $f$  (or a modifier),  $H(f)$  returns its feature:

$$H(f) = (D(f), U(f), P(f), L(f), E(f), X(f))$$

where  $D(f)$  and  $U(f)$  return the set of types of global variables defined via assignments and used via guards in  $f$ , and the rest are predicates (return 1 if *true*, and 0 if *false*) that check whether a function contains the payable modifier ( $P$ ), loops ( $L$ ), ether-sending

```
1 function mintToken (address to, uint value) public
2 returns (bool success) {
3   require (msg.sender == owner);
4'  (+) require (totalSupply + value >= totalSupply); // fix
4   totalSupply += value; // overflow
5   balances[to] += value; // overflow
6 }
```

(a) A vulnerable code snippet and fix for it.

```
1 function mintToken (address to, uint value) public
2 returns (bool success) {
3   require (msg.sender == owner);
4   balances[to] += value; // overflow
5   totalSupply += value; // overflow
6 }
```

(b) A similar (statements reversed) code snippet to repair.

**Figure 3: Code snippets for explaining offline-learning (Example 1 and 2).**

statements ( $E$ ), and contract-deactivating statements ( $X$ ) such as selfdestruct in Solidity. The design of the function feature is mostly inspired from [49], where similar abstractions were used to learn probability distributions on vulnerable transaction sequences.

**EXAMPLE 1.** Let  $p = \text{Insert}(4, \text{require}(\text{totalSupply} + \text{value} >= \text{totalSupply}))$  be a candidate patch in Figure 3(a). Suppose the types of the variables are the following: *address* (*owner*), *uint* (*totalSupply*), and *mapping*(*address*=>*uint*) (*balances*). Then, we have  $\hat{p} = \alpha(p)$  that is an abstract patch for  $p$ :

$$\hat{p} = \text{Insert}(\underbrace{(\{\text{uint}, \text{mapping}(\text{address} \Rightarrow \text{uint})\})}_{D(f)}, \underbrace{(\{\text{address}\}, 0, 0, 0, 0)}_{U(f)}, \underbrace{\{\text{uint}\}}_{T_s(s')})$$

where  $f$  denotes the function *mintToken* in Figure 3(a) and  $s' = \text{require}(\text{totalSupply} + \text{value} >= \text{totalSupply})$ . Finally, assuming a safety score for  $p$  is 0.45, we obtain a knowledge data  $k = (\hat{p}, 0.45)$ .

**Deriving a Model from Knowledge.** To derive  $M_{off}$  (line 6 in Algorithm 1), we first construct a unified prior knowledge base  $K_{off}$  by collecting all knowledge bases from past repair attempts, and transform it into a training dataset  $D_{off}$ :

$$D_{off} = \{(\langle v_1, \dots, v_n \rangle, \text{score}) \mid (\hat{p}, \text{score}) \in K_{off}\}$$

where  $\langle v_1, \dots, v_n \rangle \in \{0, 1\}^n$  is a feature vector for  $\hat{p}$ . We obtain  $\langle v_1, \dots, v_n \rangle$  using a feature encoding function  $G_C$  (i.e.,  $\langle v_1, \dots, v_n \rangle = G_C(\hat{p})$ ).  $G_C$  takes an abstract patch  $\hat{p}$  and converts it into a binary feature vector that is valid in the context of a contract under repair (in that parameterized by  $C = \{c_1, \dots, c_n\}$ ):

$$G_C(\hat{p}) = \langle v_1, \dots, v_n \rangle$$

where  $v_i$  is 1 (resp., 0) if  $\hat{p}$  contains (resp., does not contain) an abstract version of an  $i$ -th patch component  $c_i$  ( $i \in [1, n]$ ). Once  $D_{off}$  is generated, we can derive  $M_{off}$  by invoking an off-the-shelf supervised learning algorithm on  $D_{off}$  as in Section 3.2.1.

**EXAMPLE 2.** Suppose our goal is to fix the code in Figure 3(b). Suppose we have a knowledge base  $K_{off} = \{k\}$ , where  $k = (\hat{p}, 0.45)$  is the knowledge data made in Example 1. Suppose we have a set of

patch components  $C = \{c_1, c_2\}$  where

$$\begin{aligned} c_1 &= \text{Insert}(4, \text{require}(\text{balances}[to] + \text{value} \geq \text{balances}[to])), \\ c_2 &= \text{Insert}(5, \text{require}(\text{totalSupply} + \text{value} \geq \text{totalSupply})). \end{aligned}$$

Note that  $\hat{p}$  contains  $\alpha'(c_2)$  but does not contain  $\alpha'(c_1)$ :

$$\begin{aligned} \alpha'(c_1) &= \widehat{\text{Insert}}(H(f'), \underbrace{\{\text{uint}, \text{mapping}(\text{address} \Rightarrow \text{uint})\}}_{T_S(s_1)}), \\ \alpha'(c_2) &= \widehat{\text{Insert}}(H(f'), \underbrace{\{\text{uint}\}}_{T_S(s_2)}) \end{aligned}$$

where  $f'$  denotes the function `mintToken` in Figure 3(b), and  $H(f')$  equals  $H(f)$  in Example 1, and  $s_1$  (resp.,  $s_2$ ) denotes the statement in  $c_1$  (resp.,  $c_2$ ). Thus,  $G_C(\hat{p})$ , the feature vector of  $\hat{p}$ , is  $\langle 0, 1 \rangle$ . As a result, we obtain  $D_{\text{off}} = \{((0, 1), 0.45)\}$  from  $K_{\text{off}}$ .

**3.2.3 Using Learned Models.** We redefine the function cost (line 8 in Algorithm 1) using learned statistical regression models ( $M_{\text{on}}$ ,  $M_{\text{off}}$ ) to effectively guide the selection of likely candidate patches:  $\text{cost}(p) = -\text{score}(p)$  where  $\text{score}$  computes an expected safety score for a patch  $p$ :

$$\text{score}(p) = \begin{cases} M_{\text{off}}(F_C(p)) & \text{if } M_{\text{off}} \neq \perp \\ -|p| & \text{if } M_{\text{off}} = \perp \text{ and } \exists e \in E. e \notin \{v \mid (v, -) \in D_{\text{on}}\} \\ M_{\text{on}}(F_C(p)) & \text{if } M_{\text{off}} = \perp \text{ and } \forall e \in E. e \in \{v \mid (v, -) \in D_{\text{on}}\}. \end{cases}$$

In the above,  $E$  denotes the set of all possible one-hot vectors in  $n$  ( $= |C|$ ) dimensions:  $E = \{i \mid i \in \{0, 1\}^n, i = \langle i_1, \dots, i_n \rangle, \sum_{j=1}^n i_j = 1\}$ . In the first case where  $M_{\text{off}}$  is learned ( $M_{\text{off}} \neq \perp$ ), we prioritize candidates using  $M_{\text{off}}$ . In the second case where  $M_{\text{off}}$  is not available and impacts of some patch components are unknown yet, we perform size-based estimations. In the third case where  $M_{\text{off}} = \perp$  and impacts of all patch components are known, we search by prioritizing likely patches according to  $M_{\text{on}}$ . Note that the function cost is updated at each iteration in accordance with changes of  $M_{\text{on}}$  and  $M_{\text{off}}$ . Further note that, in the definition of cost, we negate the score's output, as our algorithm selects a candidate with the least cost at each iteration (line 8 in Algorithm 1).

### 3.3 Applicability to Other Types of Bugs

Although we formalized and implemented our approach for the five critical classes of bugs in Solidity smart contracts, the core principle of our technique (machine learning-based patch prioritization – Section 3.2) can be extended to support fixing other types of bugs as well. As one aspect that supports this claim, we note that the function  $Q$  (line 1 in Algorithm 2), which quantifies the verifier's feedback, does not use any information on bug types. To fix other kinds of bugs, two major extensions are necessary: devising patch search space (Section 3.1.1) for other bugs, and extending the analysis scope of the verifier (Section 3.1.2).

## 4 IMPLEMENTATION AND OPTIMIZATION

**Implementation.** We implemented SMARTFix in OCaml on top of VERISmart [50], an open-sourced [9] verification tool for smart contracts written in Solidity [6]. To learn and use statistical regression models (Section 3.2), we wrote a Python script that uses the scikit-learn library [14]. To invoke learning-related functions in the Python script from OCaml code, we used `pympl` [5].

SMARTFix provides patch safety (Section 3.1.2) by design, but it might produce unsafe patches for two reasons in practice. First, to produce simple patches as much as possible, we used several heuristics for filtering benign IO and RE alarms; they can be found in the supplementary material. Such intentionally ignored alarms do not appear in VR. Second, the underlying verifier may be unsound for some tricky features (e.g., inline assembly [50]). Nonetheless, we note that these two potential sources do not significantly harm the practicality of SMARTFix; in our experiments (Section 5), we have not observed unsafe patches generated due to these reasons.

To accelerate the repair process, similar to [51], when enumerating patches (Section 3.1.1), we prune away candidates likely to perform nonsensical actions (e.g., inserting the same bound checks at the same lines).

While the basic approach (Section 3.1) almost immediately terminates once a solution contract is found (as it enumerates patches in increasing order and  $W$  becomes empty by lines 15–16 in Algorithm 1), the learning-based approach (Section 3.2) may not. To balance the efficient termination and the patch simplicity, once the first solution contract is found, the repair-loop repeats  $Y$  (10 in the implementation) times at most.

**Acceleration via Differential Verification.** The verification-based patch validation (line 10 in Algorithm 1) ensures the patch safety, but poses a significant performance problem to SMARTFix. We devised a method based on differential verification to further mitigate this issue. In particular, our technique aims to reduce the patch validation cost in the learning-based enhanced algorithms (Section 3.2), which arises due to additional  $Y$  ( $=10$ ) iterations for finding simpler patches when solution contracts are already found.

Suppose a solution contract has been generated and its invariant is  $I$  (can be obtained as a part of outputs of VERISmart [9, 50]). Then, given a candidate contract  $s$ , we first check whether  $I$  is an inductive invariant of  $s$  or not. If not, we immediately break out of the current iteration of the repair-loop without performing safety verification and regression detection on  $s$  (typically more expensive than inductiveness checking), and start the new iteration (i.e., pick another candidate contract  $s'$  at line 8 in Algorithm 1). That is, we expect the invariant of a solution contract to hold in other solution contracts too. By precisely detecting unpromising candidates and early stopping verification for them this way, we can effectively improve the repair performance.

## 5 EVALUATION

We evaluate SMARTFix to answer the following research questions.

- How effectively does SMARTFix repair vulnerable smart contracts? How does SMARTFix compare to sGUARD [43], the state-of-the-art repair tool for smart contracts? (Section 5.2)
- Is using statistical models important for improving the performance of SMARTFix? (Section 5.3)

### 5.1 Experimental Setup

**Benchmark.** We collected four datasets with annotated bugs.

- IO Bench: 200 contracts that contain IO bugs, which are randomly selected out of 487 CVE-reported contracts [1].



- EL-SU Bench: 104 contracts from [49], which contain EL and SU vulnerabilities.
- RE Bench: 47 contracts with RE vulnerabilities from three sources: 28 contracts from [15, 19], and 19 contracts collected by us (2 contracts from the wild, and 17 contracts with injected likely bugs).
- TX Bench: 10 contracts with TX vulnerabilities: one test contract from [19] and 9 contracts with injected likely bugs.

In total, the source lines of the benchmark contracts range from 16 to 1,225, including 36 contracts that are relatively large ( $> 500$  lines). On average, the benchmarks consist of 259 lines.

To conduct a more meaningful experiment, we applied modifications on the RE Bench contracts from prior works [15, 19]. As several contracts from [19] were “buggy” and indeed safe from RE although they contain vulnerable code patterns, we modified the code so that the intended vulnerabilities can be triggered. We duplicated contracts from [15, 19] (e.g., duplicated addresses, minor syntactic differences). We excluded contracts where the root causes of bugs are not related to reentrancy.

For more extensive evaluation on datasets that contain trustful ground-truth bugs, we constructed our own benchmarks from real-world contracts deployed on the Ethereum blockchain [3]. For RE Bench, we collected 2 contracts that contain RE bugs without modifications and constructed 17 contracts by manually injecting likely bugs in deployed contracts. For TX Bench, we constructed 9 contracts by injecting likely bugs. We tried hard to inject realistic bugs as much as possible. In the supplementary material, we provide concrete bug-injection patterns that we used. Our benchmarks, including the refined benchmark set, are publicly available (Section 8).

**Comparison Tool.** We evaluate the effectiveness of SMARTFIX in comparison with sGUARD [43], the state-of-the-art bug-repair tool for Solidity contracts. We could not consider SCREPAIR [56] as we failed to run it in our environments despite our best efforts. We did not consider EVM bytecode-level patching tools (ELYSIUM [21], SMARTSHIELD [57]), since comprehending bytecode-level changes is difficult [46] and thus evaluating their patch correctness objectively is nontrivial.

Given a potentially vulnerable contract, both SMARTFIX (Section 3) and sGUARD aim to make it vulnerability-free. The overall workflow of sGUARD is the following. It first detects all potential bugs by using relatively light-weight static analyses (control- and data-dependency analyses to identify certain vulnerable instructions that have dependencies to external function calls [43]). Next, it applies bug type-specific runtime checks to fix the detected bugs.

For the experiment, we used the latest version [7] of sGUARD as of February 2023. sGUARD currently supports fixing arithmetic vulnerabilities (IO and division-by-zero), RE, and TX. Since its current implementation does not have options for specifying Solidity compiler versions and the names of main contracts to be patched, we modified its source code to take those information as inputs; the modified code is publicly available (Section 8).

**Evaluating Correctness.** Assuring patch correctness remains an open problem in automatic program repair techniques [23].

SMARTFIX and sGUARD are no exceptions. Thus, we manually validated the correctness of patches produced by each tool. In particular, we inspected whether patches safely fix vulnerabilities and do not damage any functionalities of original implementations.

**Hardware and Execution Options.** All experiments were conducted on Ubuntu machine with AMD Ryzen Threadripper 3970X CPU (3.7 GHz) (32 cores and 64 threads in total) and 62GB of memory. We ran SMARTFIX using 40 threads at most, after putting all 361 contracts together in random order. We ran sGUARD using 40 threads at most on 257 contracts, excluding 104 contracts in EL-SU Bench as it does not support fixing EL and SU. For SMARTFIX, we set timeout to 90 minutes for the total repair time (Algorithm 1). We allocated 150 seconds for each invocation of the verifier and the external timeout to 20 minutes; technically, the former is a time budget for the safety verification [50] and regression checking is done during a separate time budget (Section 3.1.2). We set the timeout to 20 seconds per Z3 [18] (v. 4.11.2) invocation from the verifier. For both tools, we set the external timeout to 100 minutes.

## 5.2 Effectiveness of SMARTFIX

**Overall Results.** Table 1 shows repair results on fixing annotated bugs, which are likely to be more impactful bugs in contracts. The column “Fix Rate” ( $\frac{\#C}{\#B}$ ) shows end-to-end fix rates on annotated bugs in entire contracts. In our experiment, sGUARD [43] generated execution errors (uncompilable patched contracts, no outputs are generated, and timeout) on non-negligible number of contracts: 50 for IO Bench, 17 for RE Bench, and 8 for TX Bench. Since these failures vacuously lowered the fix rate of sGUARD, in an effort to favorably evaluate sGUARD, we also report “Success Rate” ( $\frac{\#C}{\#B^R}$ ), a rate of successful fixes on annotated bugs in contracts without execution errors. When tools correctly fixed annotated bugs but introduced functional regressions in other parts of a contract, we considered generated patches to be incorrect.

The results show SMARTFIX is highly competitive in fixing bugs compared to the state-of-the-art. Specifically, SMARTFIX was much more effective on commonly supported classes of bugs (IO, RE, TX), achieving a success rate of 94.8% (vs. 65.4%) and an accuracy of 100.0% (vs. 97.1%). Moreover, including EL and SU, which are hardly supported by existing approaches (e.g., [21, 43, 57]) that rely on a single repair template for each bug type (Section 2), SMARTFIX obtained still noticeable results; the success rate is 81.1% and the accuracy is 96.7%.

In particular, we found SMARTFIX is far more effective than sGUARD in fixing critical arithmetic overflow bugs (IO) that are assigned CVE IDs. Concretely, SMARTFIX fixed 218 CVE-reported arithmetic bugs safely and correctly, achieving the success rate of 95.6%. By contrast, the success rate of sGUARD was only 60.2%, as the total number of repair attempts was only 103 (the column #G) out of 170 valid repair targets (the column #B<sup>R</sup>). This is because sGUARD relies on a rather restricted fix strategy to reduce false positives [43] (i.e., to reduce the number of patches that are unnecessarily applied to already-safe arithmetic operations). Specifically, sGUARD fixes only IO bugs that have dependencies to external function calls. Unfortunately, such a restricted strategy can result in missing opportunities to fix critical arithmetic bugs, as shown in Table 1. Further note that security disasters can happen due to IO



**Table 1: Results on fixing annotated bugs in each dataset. #B: the number of annotated bugs in contracts. #B<sup>R</sup>: the number of annotated bugs in contracts that are successfully run by each tool. #G: the number of annotated bugs fixed by each tool. #C: the number of annotated bugs correctly fixed by each tool. Fix Rate:  $\frac{\#C}{\#B}$  (end-to-end fix rate on annotated bugs in entire contracts). Success Rate:  $\frac{\#C}{\#B^R}$  (fix rate on annotated bugs in contracts without execution errors). Accuracy:  $\frac{\#C}{\#G}$ .**

Bug Type	#B	SMARTFix						sGUARD [43]					
		#B <sup>R</sup>	#G	#C	Fix Rate	Success Rate	Accuracy	#B <sup>R</sup>	#G	#C	Fix Rate	Success Rate	Accuracy
IO	229	228	218	218	95.2%	95.6%	100.0%	170	103	103	45.0%	60.6%	100.0%
RE	52	51	46	46	88.5%	90.2%	100.0%	33	33	29	55.8%	87.9%	87.9%
TX	12	12	12	12	100.0%	100.0%	100.0%	2	2	2	16.7%	100.0%	100.0%
EL	137	134	83	76	55.5%	56.7%	91.6%	n/a	n/a	n/a	n/a	n/a	n/a
SU	53	51	40	34	64.2%	66.7%	85.0%	n/a	n/a	n/a	n/a	n/a	n/a
<b>IO+RE+TX</b>	<b>293</b>	<b>291</b>	<b>276</b>	<b>276</b>	<b>94.2%</b>	<b>94.8%</b>	<b>100.0%</b>	<b>205</b>	<b>138</b>	<b>134</b>	<b>45.7%</b>	<b>65.4%</b>	<b>97.1%</b>
<b>Total</b>	<b>483</b>	<b>476</b>	<b>399</b>	<b>386</b>	<b>79.9%</b>	<b>81.1%</b>	<b>96.7%</b>	<b>-</b>	<b>-</b>	<b>-</b>	<b>-</b>	<b>-</b>	<b>-</b>

\* For SMARTFix, #G is counted if no alarms are generated by the patch verifier at annotated lines. For sGUARD, #G is counted if runtime checks are inserted at annotated lines (IO), nonReentrant modifiers are inserted in contracts (RE), or annotated lines have been changed by the patches (TX).

bugs irrelevant to external function calls (e.g., CVE-2018-10299). Unlike sGUARD, SMARTFix does not rely on such a rather restricted strategy to reduce false positives, as it can identify where to apply patches much more accurately (will be discussed shortly with Table 2) thanks to the verification-based patch validation.

For RE, sGUARD incorrectly fixed 4 bugs, where 3 of them were incorrect as the patches introduced deadcode due to improper overlaps of nonReentrant modifiers. By contrast, SMARTFix did not generate such incorrect patches thanks to its verification-based regression detection (Section 3.1.2), achieving 100.0% accuracy.

For EL and SU, SMARTFix achieved relatively low fix- and success rates and accuracy, compared to the numbers for the other types of bugs; we discuss the reasons in Section 5.5. Nevertheless, we believe the results are overall encouraging, in that SMARTFix is the first repair tool that can safely fix diverse patterns of access-control related bugs (EL, SU).

**Patch Simplicity.** We evaluated the patch simplicity in terms of the number of inserted runtime checks (Table 2): #BC (the number of bound checks applied to addition, subtraction, and multiplication for fixing IO bugs), #NR (the number of nonReentrant modifiers for fixing RE bugs). We assess the simplicity in terms of #BC and #NR, because BC and NR are the two types of additive repair operators that increase code sizes. Evaluating the patch simplicity is important, because redundant patches would be undesirable for developers and they would result in unnecessary payments of gas fees [43, 56].

Table 2 shows that SMARTFix can generate solution contracts more economically, by producing simpler yet safe patches thanks to the verification-based patch validation. To make 100 contracts bug-free, SMARTFix used 244 (= 243+1) runtime checks, which correspond to 26.9% of sGUARD. In particular, considering that sGUARD typically fixes fewer IO bugs (i.e., fixes only overflows that have dependencies to external calls), our result is noteworthy.

**Scalability.** To obtain the results in Table 1, the average termination time of SMARTFix is the following (excluding runtime

**Table 2: Comparison on the patch simplicity. Sol: the number of solution contracts generated by both tools. #BC: the number of inserted bound checks. #NR: the number of added nonReentrant modifiers.**

Dataset	Sol	SMARTFix		sGUARD [43]		SMARTFix sGUARD
		#BC	#NR	#BC	#NR	#BC+#NR
IO Bench	73	213	0	783	0	27.2%
RE Bench	25	27	1	53	61	24.6%
TX Bench	2	3	0	10	0	30.0%
<b>Total</b>	<b>100</b>	<b>243</b>	<b>1</b>	<b>846</b>	<b>61</b>	<b>26.9%</b>

\* For sGUARD, the contracts in Sol are patched contracts that are proven to be free from IO, RE, and TX, according to its safety criteria [43].

exception cases): 43m (for all four datasets), 43m (IO Bench), 32m (RE Bench), 17m (TX Bench), and 51m (EL-SU Bench). sGUARD took much smaller time, as it uses relatively light-weight static analyses: 34s (for three datasets), 39s (IO Bench), 8s (RE Bench), and 5s (TX Bench). Considering the importance of safe smart contracts and the benefits of SMARTFix (much higher fix- and success rates, supporting more classes of bugs, patch simplicity), we believe its cost is reasonable.

We also found that SMARTFix can be useful for relatively large contracts too. In particular, SMARTFix could fix RE bugs in contracts consisting of 829 lines and 963 lines. For 65 annotated bugs from 36 contracts with more than 500 lines, the fix- and the success rate of SMARTFix was 46.2% ( $\frac{30}{65}$ ) and 47.6% ( $\frac{30}{63}$ ). The average termination time for 34 contracts without execution errors was 1h 15 m.

**Summary of Comparison.** The experimental results show that SMARTFix has two main advantages over sGUARD by adopting the generate-and-verify approach. First, SMARTFix can fix wider ranges of security bugs safely. For example, SMARTFix can fix more diverse patterns of arithmetic bugs and it can support access-control related bugs (e.g., EL, SU), which could be hardly supported by tools such

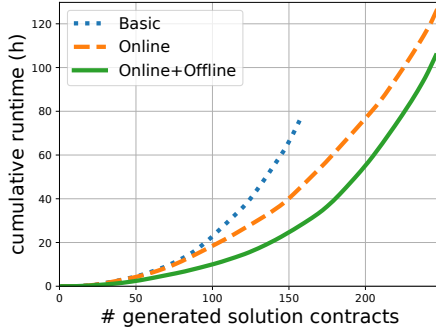


Figure 4: Comparison between the variants of SMARTFIX.

as sGUARD that rely on a single repair strategy for each bug type. Second, SMARTFIX can generate much simpler patches.

Our evaluation also identifies a downside of SMARTFIX. It could be much more inefficient than sGUARD when generating patches that are safe by construction. For example, patches that insert runtime checks for IO bugs are always safe. sGUARD is specialized for producing such patches and therefore fast. On the other hand, SMARTFIX attempts to formally prove safety even in this case.

**Contract-Level Fixing Results.** SMARTFIX was also effective at generating solution contracts too; SMARTFIX achieved a fix rate of 67.6% and an accuracy of 99.2% on the 361 benchmark contracts.

### 5.3 Impact of Using Statistical Models

To assess the utility of using statistical models (Section 3.2), we made three variants of SMARTFIX. Basic indicates SMARTFIX that performs the basic generate-and-verify repair; it uses the size-based cost function in Section 3.1. Online indicates SMARTFIX without offline learning; it uses the final cost function in Section 3.2.3 where  $M_{off} = \perp$  always. Online+Offline indicates the final version of SMARTFIX, which combines all of our techniques. The cactus plot in Figure 4 compares the performance of the three variants, by visualizing the number of generated solution contracts (x-axis) according to the cumulative termination time (y-axis).

The results show that using statistical models is critical for greatly enhancing the performance of SMARTFIX. Compared to our baseline approach (Basic), our techniques based on online and offline learning collectively improved the contract-level repair effectiveness by 56.7% ( $= \frac{246-157}{157}$ ) in terms of generating solution contracts. We also found that both online and offline learning is crucial, as depicted in Figure 4. Although the two enhanced variants (Online, Online+Offline) were similar in terms of generating solution contracts (249, 246) within 100 minutes of the termination budget (Section 5.1), when we counted the number of solution contracts generated within 30 minutes of termination time, the performance difference between the variants became more evident: Basic (89), Online (143), Online+Offline (168).

### 5.4 Limitations

To inspect room for future improvements, we manually analyzed why SMARTFIX failed to fix bugs in our experiments.

**Incorrect Patches.** SMARTFIX incorrectly fixed 13 known annotated bugs (#G – #C in Table 1), as the patches failed to satisfy contract-specific properties that are unspecified at runtime of SMARTFIX. For example, consider the simplified code snippet:

```

1  modifier onlySettler { require (msg.sender == settler); _; }
2  modifier onlyOwner { require (msg.sender == owner); _; }
3  function settleBet (string randomSeed) onlySettler { ... }
4  function setSettler (address newSettler) onlyOwner { settler = newSettler; }
5
6  (-) function withdrawFunds(address rcv, uint amt) { // "onlyOwner" removed
7  (+) function withdrawFunds(address rcv, uint amt) onlySettler {
8      rcv.send(amt); } // ether-leak

```

In the above contract for gambling, settler is responsible for determining the winner of bets made by players (line 3), and owner is responsible for managing the contract such as designating a new settler (line 4). In the original contract, the critical statement (send) at line 8, which sends ethers (amt) of the contract to rcv, can be executed by any unauthorized user and therefore has the EL bug. SMARTFIX produced the above patch using the modifier onlySettler (line 7), which can be considered vulnerability-free because settler is an authorized user (hence proven to be safe by the patch verifier [9, 50]). However, this patch is incorrect, considering the original contract (the benchmark was made by removing the modifier onlyOwner in withdrawFunds). To correctly fix these bugs, SMARTFIX should be able to distinguish the roles of settler and owner, and infer that withdrawFunds should be only invoked by owner not by another trusted user (settler).

**Unfixed Bugs.** We identified three main reasons on why SMARTFIX failed to generate patches. First, SMARTFIX failed due to the limited performance of the verifier. For example, SMARTFIX even could not finish the verification within a verification time budget for a complex original contract (line 1 in Algorithm 1), causing the abnormal early termination of the repair procedure (in the implementation, if we fail to obtain  $VR_0$  and  $RR_0$ , we just stop the execution). Second, SMARTFIX could not effectively handle rather a large search space for candidate patches. For example, SMARTFIX was able to fix only one bug out of the three annotated bugs for a contract in which 21 patch components were generated to fix 19 potential bugs. Third, SMARTFIX failed due to the imprecision of the verifier; even though desired patches have been searched, the verifier failed to validate their safety.

### 5.5 Threats to Validity

We discuss potential threats to validity of our experimental results. First, the four datasets used in our experiments might not be representative despite our significant effort for the benchmark construction (Section 5.1); thus, the effectiveness of SMARTFIX on other new datasets remains to be seen. Second, we did not conduct a comprehensive study on the choice of hyperparameter values (e.g.,  $\gamma$  in Section 3.2, Z3 solving timeout) used in our algorithm; the efficacy of our techniques might vary if using different values.

## 6 RELATED WORK

**Repairing Smart Contracts.** Compared to recent approaches for fixing vulnerable smart contracts before they get deployed [21, 43, 56, 57], SMARTFIX is differentiated in that it achieves high repairability, full automation, and safety guarantee all at once.

sGUARD [43] is a tool that aims to make contracts vulnerability-free. ELYSIUM [21] and SMARTSHIELD [57] are tools that generate patches at EVM bytecode level, in combination with bug-finders such as MYTHRIL [4]. A major weakness of them [21, 43, 57] is that they cannot fix diverse patterns of bugs, as they rely on a single repair template for each bug type. SCREPAIR [56], a test-based repair tool, would be an exception from this limitation as it supports multiple repair strategies by using a genetic search algorithm. However, unlike SMARTFix, SCREPAIR does not achieve full automation and patch safety, as it relies on test suites for patch validation.

There are also techniques for patching already-deployed contracts [30, 37, 46]. SMARTFix is complementary to these approaches. For example, for patch validation, EVMPATCH [46] uses past transactions as test suites, which would not be available before deployment. By contrast, SMARTFix can safely fix bugs without them.

**Repairing Traditional Programs.** There are a bunch of prior works on automatically fixing traditional and typically larger programs such as C or Java. In particular, our work is closely related to generate-and-validate approaches [29, 32, 38, 39, 55], which iteratively generate candidate patches using search algorithms until a solution program is found. Our work substantially differs from these works in two aspects. First, while most existing works perform patch validation using test cases, we perform verification-based patch validation to guarantee safety of generated patches. Second, and more importantly, we propose a new learning-based technique to speed up the repair procedure using statistical models.

Our work is also related to approaches that use sound verification techniques [27, 34, 36] for patch validation. MEMFix [36] and SAVER [27] use abstract interpretation [17] specially designed to safely fix memory errors (e.g., memory leak) in C/C++. By contrast, we use hoare-style verification techniques [50] to safely fix security vulnerabilities in smart contracts. Similar to ours, the work in [34] uses a hoare-style verification tool to guarantee patch correctness of C programs (assuming functional specifications of programs are available). The difference is, it uses the verifier’s feedback (the number of alarms [34]) in the context of genetic programming, while we leverage the verifier’s feedback for learning statistical models.

**Analyzing Smart Contracts.** A number of analyzers for smart contracts have been developed (e.g., [4, 10, 13, 16, 20, 22, 24, 28, 31, 33, 40, 42, 44, 45, 47, 49, 50, 52, 53]). We believe using other tools in the patch validation could enhance the practicality of SMARTFix. For example, fuzzers (e.g., [16, 24, 28]) would help to filter false alarms of the patch verifier and thus produce more economical patches.

## 7 CONCLUSION

In this paper, we showed that the “generate-and-verify” approach can be made practical for repairing vulnerable smart contracts. Previous techniques for automatically repairing smart contracts failed to achieve high repairability, full automation, or safety guarantee. The generate-and-verify approach has the potential to overcome these shortcomings but poses a significant performance challenge. We presented a learning-based technique to accelerate the approach using statistical models derived from the verifier’s feedback, and

demonstrated that our enhanced method is highly effective at fixing five important classes of vulnerabilities in smart contracts.

## 8 DATA AVAILABILITY

The replication package for this paper, including the source code of SMARTFix and the benchmarks, is publicly available at: <https://github.com/kupl/SmartFix-Artifact>.

## ACKNOWLEDGMENT

We thank Doseop Lee for his contribution to the construction of the RE benchmark. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair). This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-00177,High Assurance of Smart Contract for Secure Software Development Life Cycle). This work was also supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains). This research was also supported by the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2023-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). This work was also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No.2021R1A5A1021944). This research was also supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(2020R1A6A3A13068761).

## REFERENCES

- [1] [n. d.]. 487 smart contracts that contain arithmetic vulnerabilities with assigned CVE IDs. <https://github.com/kupl/VeriSmart-benchmarks/tree/master/benchmarks/cve>. Accessed: February 2023.
- [2] [n. d.]. A \$50 Million Hack Just Showed That the DAO Was All Too Human. <https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/>. Accessed: February 2023.
- [3] [n. d.]. Etherscan: The Ethereum Blockchain Explorer. <https://etherscan.io/>. Accessed: February 2023.
- [4] [n. d.]. Mythril: a security analysis tool for EVM bytecode. <https://github.com/ConsensSys/mythril>. Accessed: February 2023.
- [5] [n. d.]. pyml: OCaml bindings for Python. <https://opam.ocaml.org/packages/pyml>. Accessed: February 2023.
- [6] [n. d.]. Solidity Documentation. <https://docs.soliditylang.org>. Accessed: February 2023.
- [7] [n. d.]. The Github repository of for the latest version of sGUARD. <https://github.com/duytai/sGuard/tree/643c5f67f21d5a433965218a84ce407d93ccdc23>. Accessed: February 2023.
- [8] [n. d.]. The Parity Wallet Hack Explained. <https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/>. Accessed: February 2023.
- [9] [n. d.]. VeriSmart: a formal verification tool for Solidity smart contracts. <https://github.com/kupl/VeriSmart-public>. Accessed: February 2023.
- [10] Leonardo Alt and Christian Reitwiesner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 376–388.
- [11] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1235–1252. <https://doi.org/10.1109/SP46214.2022.00072>



- [12] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag, Berlin, Heidelberg.
- [13] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 454–469. <https://doi.org/10.1145/3385412.3385990>
- [14] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.
- [15] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, Yuxing Tang, Xiaodong Lin, and Xiaosong Zhang. 2020. SODA: A Generic Online Detection Framework for Smart Contracts. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. <https://www.ndss-symposium.org/ndss-paper/soda-a-generic-online-detection-framework-for-smart-contracts/>
- [16] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 227–239. <https://doi.org/10.1109/ASE51524.2021.9678888>
- [17] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) (POPL '77). Association for Computing Machinery, New York, NY, USA, 238–252. <https://doi.org/10.1145/512950.512973>
- [18] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.
- [19] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 530–541. <https://doi.org/10.1145/3377811.3380364>
- [20] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. <https://doi.org/10.1109/WETSEB.2019.00008>
- [21] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2022. Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses* (Limassol, Cyprus) (RAID '22). Association for Computing Machinery, New York, NY, USA, 115–128. <https://doi.org/10.1145/3545948.3545975>
- [22] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2757–2774. <https://www.usenix.org/conference/usenixsecurity20/presentation/frank>
- [23] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. <https://doi.org/10.1145/3318162>
- [24] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) (CCS '19). Association for Computing Machinery, New York, NY, USA, 531–548. <https://doi.org/10.1145/3319535.3363230>
- [25] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Deobfuscation via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) (CCS '18). Association for Computing Machinery, New York, NY, USA, 380–394. <https://doi.org/10.1145/3243734.3243838>
- [26] Tharaka Hewa, Mika Ylianttila, and Madhusanka Liyanage. 2021. Survey on blockchain based smart contracts: Applications, opportunities and challenges. *Journal of Network and Computer Applications* 177 (2021), 102857. <https://doi.org/10.1016/j.jnca.2020.102857>
- [27] Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. 2020. SAVER: Scalable, Precise, and Safe Memory-Error Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) (ICSE '20). Association for Computing Machinery, New York, NY, USA, 271–283. <https://doi.org/10.1145/3377811.3380323>
- [28] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) (ASE 2018). Association for Computing Machinery, New York, NY, USA, 259–269. <https://doi.org/10.1145/3238147.3238177>
- [29] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) (ISSTA 2018). Association for Computing Machinery, New York, NY, USA, 298–309. <https://doi.org/10.1145/3213846.3213871>
- [30] Hai Jin, Zeli Wang, Ming Wen, Weiqi Dai, Yu Zhu, and Deqing Zou. 2021. Aroc: An Automatic Repair Framework for On-chain Smart Contracts. *IEEE Transactions on Software Engineering* (2021), 1–1. <https://doi.org/10.1109/TSE.2021.3123170>
- [31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society. [http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018\\_09-1\\_Kalra\\_paper.pdf](http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf)
- [32] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) (ICSE '13). IEEE Press, 802–811.
- [33] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1317–1333. <https://www.usenix.org/conference/usenixsecurity18/presentation/krupp>
- [34] Xuan-Bach D. Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing Automated Program Repair with Deductive Verification. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 428–432. <https://doi.org/10.1109/ICSME.2016.66>
- [35] Xuan-Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Over-fitting in Semantics-Based Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) (ICSE '18). Association for Computing Machinery, New York, NY, USA, 163. <https://doi.org/10.1145/3180155.3182536>
- [36] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: Static Analysis-Based Repair of Memory Deallocation Errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) (ESEC/FSE 2018). Association for Computing Machinery, New York, NY, USA, 95–106. <https://doi.org/10.1145/3236024.3236079>
- [37] Zecheng Li, Yu Zhou, Songtao Guo, and Bin Xiao. 2021. SolSaviour: A Defending Framework for Deployed Defective Smart Contracts. In *Annual Computer Security Applications Conference (Virtual Event, USA) (ACSAC)*. Association for Computing Machinery, New York, NY, USA, 748–760. <https://doi.org/10.1145/3485832.3488015>
- [38] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) (ESEC/FSE 2015). Association for Computing Machinery, New York, NY, USA, 166–178. <https://doi.org/10.1145/2786805.2786811>
- [39] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) (POPL '16). Association for Computing Machinery, New York, NY, USA, 298–312. <https://doi.org/10.1145/2837614.2837617>
- [40] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) (CCS '16). Association for Computing Machinery, New York, NY, USA, 254–269. <https://doi.org/10.1145/2976749.2978309>
- [41] Sergey Mehtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. <https://doi.org/10.1109/ICSE.2015.63>
- [42] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. <https://doi.org/10.1109/ASE.2019.00133>
- [43] T. Duy Nguyen, L. Hong Pham, and J. Sun. 2021. sGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 982–996. <https://doi.org/10.1109/SP40001.2021.00057>
- [44] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) (ACSAC '18). Association for Computing Machinery, New York, NY, USA, 653–663. <https://doi.org/10.1145/3274694.3274743>



- [45] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachler-Cohen, and M. Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 414–430. <https://doi.org/10.1109/SP.2020.00024>
- [46] Michael Rodler, Wenting Li, Ghassan O. Karamé, and Lucas Davi. 2021. EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1289–1306. <https://www.usenix.org/conference/usenixsecurity21/presentation/rodler>
- [47] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (Virtual Event, USA) (CCS '20)*. 621–640. <https://doi.org/10.1145/3372297.3417250>
- [48] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (Bergamo, Italy) (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. <https://doi.org/10.1145/2786805.2786825>
- [49] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. <https://www.usenix.org/conference/usenixsecurity21/presentation/so>
- [50] S. So, M. Lee, J. Park, H. Lee, and H. Oh. 2020. VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 718–734. <https://doi.org/10.1109/SP.2020.00032>
- [51] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Seattle, WA, USA) (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 727–738. <https://doi.org/10.1145/2950290.2950295>
- [52] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference (San Juan, PR, USA) (ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 664–676. <https://doi.org/10.1145/3274694.3274737>
- [53] Petar Tsankov, Andrei Dan, Dana Drachler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (Toronto, Canada) (CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. <https://doi.org/10.1145/3243734.3243780>
- [54] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 189 (oct 2019), 29 pages. <https://doi.org/10.1145/3360615>
- [55] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. Association for Computing Machinery, New York, NY, USA, 364–374. <https://doi.org/10.1109/ICSE.2009.5070536>
- [56] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart Contract Repair. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 27 (sep 2020), 32 pages. <https://doi.org/10.1145/3402450>
- [57] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic Smart Contract Protection Made Easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 23–34. <https://doi.org/10.1109/SANER48275.2020.9054825>

Received 2023-02-02; accepted 2023-07-27