Property-Based Automated Repair of DeFi Protocols

Palina Tolmach
palina001@ntu.edu.sg
Nanyang Technological University
Institute of High Performance Computing, A*STAR
Singapore

Yi Li
yi_li@ntu.edu.sg
Nanyang Technological University
Singapore

Shang-Wei Lin
shang-wei.lin@ntu.edu.sg
Nanyang Technological University
Singapore

ABSTRACT

Programming errors enable security attacks on smart contracts, which are used to manage large sums of financial assets. Automated program repair (APR) techniques aim to reduce developers' burden of manually fixing bugs by automatically generating patches for a given issue. Existing APR tools for smart contracts focus on mitigating typical smart contract vulnerabilities rather than violations of functional specification. However, in decentralized financial (DeFi) smart contracts, the inconsistency between intended behavior and implementation translates into the deviation from the underlying financial model, resulting in monetary losses for the application and its users. In this work, we propose DeFinery—a technique for automated repair of a smart contract that does not satisfy a user-defined correctness property. To explore a larger set of diverse patches while providing formal correctness guarantees w.r.t. the intended behavior, we combine search-based patch generation with semantic analysis of an original program for inferring its specification. Our experiments in repairing 9 real-world and benchmark smart contracts prove that DeFinery efficiently generates high-quality patches that cannot be found by other existing tools.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; **Software verification and validation**.

KEYWORDS

Smart contract, program repair, symbolic execution

ACM Reference Format:
Palina Tolmach, Yi Li, and Shang-Wei Lin. 2022. Property-Based Automated Repair of DeFi Protocols. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559560

1 INTRODUCTION

Smart contracts are computer programs that are executed on top of blockchain. In this work, we focus on smart contracts that are implemented in Solidity—the most popular smart contract programming language. One of the most prominent applications of smart contracts is Decentralized Finance (DeFi). DeFi protocols are blockchain-based applications that enable a wide range of crypto-financial services, allowing users to obtain and manage digital assets, usually tokens [29]. Listing 1 shows an implementation of a *transfer* functionality in a token smart contract. By invoking the `_internalTransferFrom` function, a user can send some of the tokens he/she owns to another blockchain address.

With more than $54 billion locked in DeFi smart contracts [21], it becomes critical to ensure that their implementations are free from bugs and vulnerabilities. Yet, the adoption of DeFi protocols is accompanied by numerous security exploits leading to billions of dollars being stolen from the underlying smart contracts. According to a recent report [12], as much as $1.3 billion were lost in smart contract hacks in 2021. Many of these attacks were enabled by software bugs or security issues left in the smart contract code [3, 5, 13, 16, 22]. With automated program repair (APR), many of such bugs could be fixed automatically. However, the vast majority of existing work on smart contract repair is focused on template-based patching of common security issues, which are identified as patterns in smart contract code through static analysis or symbolic execution [15, 19, 23, 33]. For example, SmartShield [33], deploys pre-defined rectification strategies if a smart contract contains one of the three code patterns: state changes after external calls, missing checks for out-of-bound arithmetic operations, and missing checks for failing external calls. If unaddressed, these issues may cause reentrancy, integer over- and underflow, and "unchecked send" vulnerabilities that have been extensively studied [1, 4, 30].

```
1  contract iToken ... { ...
2      function _internalTransferFrom(
3          address _from, address _to,
4          uint256 _value, ...) internal { ...
5  +       require(_to != _from);
6          uint256 balancesFrom = balances[_from];
7          uint256 balancesTo = balances[_to];
8
9          require(balancesFrom >= _value);
10         uint256 balancesFromNew = balancesFrom - _value;
11         balances[_from] = balancesFromNew;
12         uint256 balancesToNew = balancesTo + _value;
13         balances[_to] = balancesToNew;
14     }
15 }
```

Listing 1: Simplified source code of iToken [20]

While many attacks are indeed attributed to well-known smart contract vulnerabilities, numerous exploits happened due to semantic (logical) bugs in smart contract code that are unlikely to be captured by a universal vulnerability pattern. Preventing logical issues is especially important for DeFi smart contracts, which encode the financial model of the application, thereby regulating

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559560

the interactions between economic agents in the DeFi ecosystem—regular users and other DeFi protocols. Discrepancy between the intended behavior and smart contract implementation may, therefore, enable harmful user behaviors, such as buying tokens at an abnormal exchange rate [13] or getting them for free [3], borrowing more tokens than should be allowed [5], and many others. Some of these issues only manifest in a violation of a high-level functional specification defined for a specific DeFi protocol, and cannot be encoded as a low-level code pattern. Likewise, these bugs cannot be fixed by existing pattern-based analyses and patching techniques that do not capture behavioral aspects of smart contract operations.

One example of such issue is the iToken hack [16] that we use to illustrate our technique (Listing 1). Due to the specifics of users' balance update (lines 10-13), the implementation (twice audited by top security firms) is prone to *token duplication* if the parameters "`_from`" and "`_to`" are equal. An attacker used this issue to artificially increase his iToken balance, which resulted in $8 million lost from the attack [16]. Although being difficult to detect through the existing vulnerability patterns, this issue can easily be identified as a violation of a basic token transfer invariant: "the sum of sender and recipient's balances should remain constant".

At the same time, as important as it is to remove the problematic behavior from a smart contract, it is equally critical that the remaining valid behavior satisfying the property is preserved by the patch. Code modifications that are too restrictive can remove essential functionality from a smart contract, breaking its core logic and introducing additional issues. For example, adding an unsatisfiable assertion, e.g., `require(false)` as line 5 in Listing 1 would have prevented the exploit of the token duplication issue. But it would also deprive the users and the smart contract itself of the ability to execute the key operation on a token, damaging its usability.

In this work, we propose an approach that enables *property-based* automated repair of a smart contract while providing formal correctness guarantees w.r.t. its original valid behavior. Given (1) a smart contract (or a set of smart contracts with one of them known to be vulnerable), (2) a property, and (3) a trace leading to its violation, our tool DeFinery generates a patched version of a smart contract which satisfies the property at all times but is conditionally equivalent to the original version under valid, i.e., non-bug-triggering, inputs. To find a smart contract modification that satisfies these constraints, we need a high level of flexibility in the patch generation process—as can be seen in Sect. 4, DeFi smart contract issues include missing or incorrect pre- and postconditions, missing variable updates, etc. Since this level of flexibility is hard to obtain using predefined fixing strategies, we perform search-based patch generation that mutates the original smart contract using genetic algorithm search. To maintain the readability of a smart contract and improve the efficiency of our technique, our patch generation prioritizes smaller changes as well as modifications that are more likely to fix a smart contract issue. To assess validity of a patch, we perform equivalence checking between semantic information inferred from valid executions of an original smart contract and executions of a patched smart contract under similar—valid—inputs. We gather these semantic information using symbolic execution—a program analysis technique for evaluating the behavior of a program on all possible inputs by assigning symbolic (instead of concrete) values to input parameters. By striking the balance between scalable exploration of diverse patches and strong correctness guarantees, DeFinery generates one of the correct fixes for the iToken example: adding a check that does not allow parameters "`_from`" and "`_to`" to be equal (line 5, equivalently can be inserted after line 13).

Contributions. We summarize our contributions as follows:

- We introduce a novel automated repair approach for smart contracts that can fix violation of functional specification expressed as a property while providing solid correctness guarantees.
- We propose a set of functional properties that help identify and fix executions violating technical and/or economical security [3, 5, 13, 16, 22] of a smart contract.
- We implement the approach as a tool and evaluate it on a dataset of 9 vulnerable smart contracts constructed from previously exploited DeFi protocols and a SmartBugs benchmark dataset [8].

2 RELATED WORK

The majority of the existing tools for smart contract repair are only able to patch a number of well-known vulnerabilities. These tools include SmartShield [33], sGuard [19], EVMPatch [23], Elysium [31], Aroc [15], and HCC [11]. Most of them rely on static analysis or symbolic execution tools to identify whether a smart contract contains a specific vulnerability and choose a fixing pattern accordingly. SCRepair [32]—a genetic mutation-based APR tool, also relies on a static vulnerability detector for fault localization. It also utilizes a set of test cases as a weak correctness criteria, while they may not be available for smart contracts and may cause test overfitting of the generated patches. Different from all these tools, our approach enables automated repair of semantic smart contract issues that result in violation of functional specification and, thus, financial losses. DeFinery provides strong correctness guarantees for the generated patches, while not requiring access to test cases or historical transaction data, which may not always be available.

Adjacent lines of work address the problem of updating a vulnerable smart contract that has already been deployed [18] or enforcing runtime validity of smart contracts with respect to the user-provided invariant [17]. However, these techniques do not perform automated repair of a smart contract.

3 METHODOLOGY

Figure 1 shows a high-level overview of DeFinery architecture. It comprises two main components: a *semantic analysis* module and a *patch generation* module. (1) First, *semantic analysis (SA)* module symbolically executes an input smart contract w.r.t. a property and a sequence of functions leading to its violation, which we refer to as a *trace*. We assume the property is provided by the user, while the trace can be generated by a smart contract verification tool. (2) For each execution path, we generate a "test case" by setting symbolic variables to concrete values generated by Z3 SMT-solver [6]. Concrete values restrict the execution of the contract towards a specific execution path, which allows faster checking of whether the modified code behaves similarly to the original for given concrete inputs. To enable more thorough assessment of patches, DeFinery also summarizes the observed *valid* behaviors of an input contract in a symbolic summary—a first-order logic (FOL) formula over a set of input and output variables. (3) Given a set of function names appearing in the trace, test cases, and a symbolic summary—all generated by the *SA* module, the *patch generation (PG)* module mutates these functions' code using a genetic algorithm and a set of

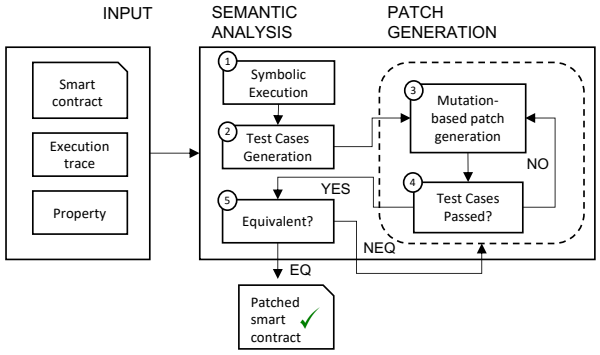

Figure 1: Tool architecture.

heuristics. (4) For each generated candidate, it invokes the symbolic execution component to check the test cases. If all test cases pass, the patch might be plausible. In this case, (5) the *SA* module is used to build a symbolic summary for a patched smart contract. Z3 is then used to perform conditional equivalence checking between the original and the patched symbolic summaries under valid inputs.

Symbolic Analysis. We perform symbolic analysis using our own source-level symbolic execution engine for Solidity smart contracts developed in C++. Binary is available in supplementary material [28]. To perform symbolic execution, we construct a *harness function* that orchestrates functions of the contracts as call sequences, based on the trace provided as input. This function is defined in a separate *Main* smart contract. The harness for iToken (Listing 1) is shown in Listing 2. We define a smart contract named *User* with a function `transferTo()`, which, in turn, calls `_internalTransferFrom()` of iToken. The parameter `_from` in the latter function, then, becomes *User*'s address. For simplicity, we assume that `_to` can be an address of one of three smart contracts: *User*, *Main*, and *iToken*. This assumption is represented by a constraint on the symbolic variable `TO` in line 4 of Listing 2. Parameter `_value` corresponds to a symbolic variable `VAL`, which is any positive unsigned integer. The harness ends with a declaration of a property in the form of an assertion, which can be built from variables visible in the *Main* contract and Solidity operators. To check that the sum of the token balances is constant (line 9), we record the balances of mentioned three contracts before (lines 2-3) and after (lines 7-8) the execution of `transferTo()` (line 6).

```
1  function _harness_() { ...
2      uint256 init_sum = balances[User] + balances[Main]
3                       + balances[iToken];
4      assume(TO == User || TO == Main || TO == iToken);
5      // Test Case: TO = Main; VAL = 100;
6      User.transferTo(TO, VAL);
7      uint256 res_sum = balances[User] + balances[Main]
8                       + balances[iToken];
9      declare_property(init_sum == res_sum);
10 }
```

Listing 2: Harness function example

For the iToken example, symbolic execution generates 5 valid execution paths that satisfy the property and 1 invalid path that violates it. For each path, DeFinery generates a test case by assigning concrete values to symbolic variables in the harness . Line 5 in Listing 2 shows one such instrumentation corresponding to a valid path. While this test suite is insufficient as a correctness criteria, it helps to quickly discard some incorrect patches before moving on to a more rigorous correctness check relying on *symbolic summaries*.

To build a symbolic summary S of the trace, our symbolic engine maintains a symbolic state for each possible path storing a path condition—a FOL formula describing the conditions satisfied by the branches taken along that path, and effect—a mapping of variables to symbolic values or expressions. A symbolic summary of a path is a conjunction of its path condition and symbolic state. For example, a summary of the path captured by the values shown in line 5 of Listing 2 is shown in line 1 (path condition) and line 2 (effect) in Eq. (1). Line 3 in Eq. (1) summarizes a path that does not satisfy a check in line 9 of Listing 1—the execution reverts, and no effects are recorded. A summary S of the trace is a disjunction of its path summaries, as partially shown in Eq. (1). The information about invalid paths, e.g., for $(TO == User \wedge balancesFrom \geq VAL)$, is not included in a summary but is recorded for future use.

$$\begin{aligned} S = &(TO == Main \wedge balancesFrom \geq VAL \wedge \\ &balances[\_from] \mathrel{-}= VAL \wedge balances[\_to] \mathrel{+}= VAL) \vee \\ &(TO == Main \wedge balancesFrom < VAL) \vee \ldots \end{aligned} \tag{1}$$

To facilitate fixing reentrancy [30], we also label traces that exhibit reentrant behavior, i.e., contain an external call and a callback to the same or another contract. This pattern helps efficiently handle both same- and cross-contract reentrancy, as shown in Sect. 4.

Patch Generation. The patch generation module of DeFinery is based on SCRepair [32]. DeFinery extends a set of statements that are synthesized by SCRepair, integrates our semantic analysis module for patch evaluation, and adds heuristics for selecting changes that would likely fix the issue. We use three mutation operators:

- **`Insert(St)`** generates a statement `St` of one of three types in the following order: (1) a `require()` statement, (2) an assignment, (3) any expression appearing in the same function. For optimization purposes, we only insert statements at the beginning or the end of the block: a body of a function or of an if-else statement.
- For the same reason, we only select expressions within a `require()` statement as a target for **`Replace(Exp)`**, which replaces an expression `Exp` with another expression of the same type;
- **`Move(St)`** moves the statement `St` to the beginning or the end of the block. For most of our experiments, using only `Insert`, `Replace` operators and their combination has been proven most efficient, unless an invalid trace falls into the reentrancy pattern. In this case, we fix the contract by enforcing the Check-Effect-Interaction pattern, i.e., by *moving* the function call to the end of the block following the state update (see rows 8–9 of Table 1).

These heuristics were proved efficient for our experimental dataset, but we leave extending the patch search space for future work. To guide the search, we use two fitness functions: (1) the number of invalid traces that were fixed and (2) the number of valid traces that remain valid. We also use the patch simplicity fitness function from SCRepair prioritizing patches with less mutations.

Conditional Equivalence Checking. To ensure that a patched version behaves similarly to the original smart contract, we compute a symbolic summary S' of its executions under the inputs, for which the original contract shows valid behavior. To determine if a patched contract's path corresponds to such valid inputs, we conjunct its

Table 1: A summary of DeFinery evaluation.

#	Smart Contract	Patch	Property	Result DeFinery	Result sGuard	Result SmartShield
1	xForce [10]	`+ require(result);`	User didn't receive `xForce` if he didn't provide any `Force`	✓	✗	✓
2	Confused_Sign [24]	`- require(amt >= bal[msg.sender]);` `+ require(amt <= bal[msg.sender]);`	User can't withdraw more than he deposited; he can receive a refund	✓	✗	✗
3	Value [7]	`+ initialized = true;`	The staked token can't be changed	✓	✗	✗

4	Uranium [14]	`require(balance0 * balance1 >=`
		`- _res0 * _res1 * 10**2);`
		`+ _res0 * _res1 * 100**2);`
			(Constant) product of pool reserves is non-decreasing	✓	✗	✗

5	Refund_NoSub [26]	`+ balances[msg.sender] = 0;`	Sum of balances is constant; the user can receive a refund	✓	✗	✗
6	Unprotected [27]	`+ require(owner == msg.sender);`	Owner can only be changed to a trusted address	✓	✗	✗
7	iToken [20]	`+ require(_from != _to);`	Constant sum of balances is preserved by a *transfer*	✓	✗	✗

8	cToken [9]	`- amp.transfer(borrower, amount);`
		`borrowBalance[borrower] += amount;`
		`+ amp.transfer(borrower, amount);`
			Protocol balance can't decrease	✓	✗	✗

9	EtherBank [25]	`- msg.sender.call.value(amount);`
		`userBalances[msg.sender] = 0;`
		`+ msg.sender.call.value(amount);`
			User's sum of balances is constant	✓	✓	✗

path condition with negated path conditions of invalid trace(s). If the resulting clause is satisfiable, this path does not correspond to bug-triggering inputs and the updated set of path conditions is added to a path summary. For example, the (partial) summary S' for the patched version shown in Listing 1 is demonstrated in Eq. (2).

Then, we build an equivalence assertion between symbolic summaries of the original and patched versions—S (Eq. (1)) and S' (Eq. (2)), respectively. An equivalence assertion is a FOL formula Φ that helps determine logical and, thus, functional equivalence between S and S': $\Phi = \neg(S \Leftrightarrow S')$ [2]. We provide this formula to Z3, which either proves that Φ cannot be satisfied, meaning that the executions are equivalent and the patch is correct, or finds a counterexample indicating that smart contracts produce different outputs for at least one input—in this case, we continue with the patch generation.

$$\begin{aligned} S' = (&TO == Main \wedge \mathbf{TO} \neq \mathbf{User} \wedge balancesFrom \geq VAL \wedge \\ &\neg(\mathbf{TO} == \mathbf{User} \wedge \mathbf{balancesFrom} \geq \mathbf{VAL}) \wedge \\ &balances[\_from] \mathrel{-}= VAL \wedge balances[\_to] \mathrel{+}= VAL) \vee \\ (&TO == Main \wedge \mathbf{TO} \neq \mathbf{User} \wedge balancesFrom < VAL \wedge \\ &\neg(\mathbf{TO} == \mathbf{User} \wedge \mathbf{balancesFrom} \geq \mathbf{VAL})) \vee \ldots \end{aligned} \quad (2)$$

4 PRELIMINARY EVALUATION

In this section, we report the results of our evaluation on 9 smart contracts that include 5 previously exploited DeFi smart contracts and 4 smart contracts from the SmartBugs dataset [8]. The choice of experimental subjects aims to prove that DeFinery is applicable to both DeFi and regular smart contracts. It also shows that issues caused by typical vulnerabilities can be fixed by DeFinery too. We simplified some smart contracts to allow processing them with our symbolic engine. Their source code and results are available on our website [28]. We ran the experiments on MacOS Monterey v.12.3.1, 32GB RAM and 2 GHz quad-core Intel Core i5 processor.

Table 1 summarizes the evaluation of our tool on 9 smart contracts, showing the correct patch found by DeFinery and the property that was used. The results show various patches that fix both common smart contract issues ("unchecked send" (1), reentrancy (8,9)) as well as missing or wrong pre- and post-conditions (2,4,6,7) and variable updates (3,5). On average, it took DeFinery 53 seconds to find a correct patch. We repeated each experiment 5 times.

We compare DeFinery to sGuard [19] and SmartShield [33]. Two other tools, EVMPatch [23] and Elysium [31], are not available at the time of the evaluation. While we reused part of SCRepair [32], we had to modify its implementation for it to compile, thus, we could not use it for comparison. sGuard and SmartShield can only fix common vulnerabilities and cannot repair most smart contracts in our dataset. sGuard can only repair reentrancy in the EtherBank smart contract (9) after minor modification of the code. SmartShield fixes only the "unchecked send" issue in xForce (1).

The $50M bug in Uranium (4) caused by using a wrong constant [13] is one of the issues that can be fixed by DeFinery but not existing tools. It cannot be found or fixed by a pattern, but even if the faulty statement is localized, SCRepair will try to replace it with a completely new one, which is inefficient due to the complexity of the correct statement. In closing, our evaluation shows that DeFinery can efficiently repair smart contracts that cannot be fixed by other tools, while preserving correctness of the remaining behavior.

5 CONCLUSION AND FUTURE WORK

In this work, we formulate the problem of property-based automated repair of smart contracts. We propose an approach that makes a first attempt at fixing violations of functional specification in smart contracts. We also demonstrate that combining semantic inference with search-based patch generation is a promising direction for smart contract repair. Our future work includes extending the considered patch search space and improving fault localization to enable more effective patch generation. We plan to expand the experimental dataset and evaluate the impact of our patches on gas consumption. We also consider integrating a verification tool that can automatically find the trace leading to the property violation.

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag, Berlin, Heidelberg, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[2] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: Scaling Program Equivalence Checking via Iterative Abstraction and Refinement of Common Code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3368089.3409757

[3] Alberto Cevallos. 2021. xFORCE Exploit Post Mortem. https://blog.forcedao.com/xforce-exploit-post-mortem-7fa9dcba2ac3/. Accessed: May 26, 2022.

[4] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.* 53, 3, Article 67 (jun 2020), 43 pages. https://doi.org/10.1145/3391195

[5] C.R.E.A.M. 2021. C.R.E.A.M. Finance Post Mortem: AMP Exploit. https://medium.com/cream-finance/c-r-e-a-m-finance-post-mortem-amp-exploit-6ceb20a630c5/. Accessed: May 26, 2022.

[6] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[7] Value DeFi. 2021. Value DeFi: ProfitSharingRewardPool Source Code – BSCScan. https://bscscan.com/address/0x7a8ac384d3a9086afcc13eb58e90916f17affc89#code. Accessed: May 26, 2022.

[8] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. SmartBugs: A Framework to Analyze Solidity Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1349–1352. https://doi.org/10.1145/3324884.3415298

[9] Cream Finance. 2021. CREAM Finance cToken smart contract. https://github.com/CreamFi/compound-protocol/blob/0e079dd9e1d6fdf974ab429a17b955dedf677315/contracts/CToken.sol#L442. Accessed: May 26, 2022.

[10] ForceDAO. 2021. xForce: ForceProfitSharing Source Code – EtherScan. https://etherscan.io/address/0xe7f445b93eb9cdabfe76541cc43ff8de930a58e6#code. Accessed: May 26, 2022.

[11] Jens-Rene Giesen, Sebastien Andreina, Michael Rodler, Ghassan O. Karame, and Lucas Davi. 2022. Practical Mitigation of Smart Contract Bugs. arXiv:2203.00364 [cs.CR]

[12] Eliza Gkritsi. 2022. Funds Lost to DeFi Hacks More Than Doubled to $1.3B in 2021: Certik. https://www.coindesk.com/business/2022/01/13/funds-lost-to-defi-hacks-more-than-doubled-to-13b-in-2021-certik/. Accessed: May 26, 2022.

[13] Colin Harper. 2021. Binance Chain DeFi Exchange Uranium Finance Loses $50M in Exploit. https://www.coindesk.com/markets/2021/04/28/binance-chain-defi-exchange-uranium-finance-loses-50m-in-exploit/. Accessed: May 26, 2022.

[14] Igor Igamberdiev. 2021. Binance Chain DeFi Exchange Uranium Finance Loses $50M in Exploit. https://twitter.com/FrankResearcher/status/1387347036916260869. Accessed: May 26, 2022.

[15] 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

[16] K. J. Kistner. 2020. iToken Duplication Incident Report. https://bzx.network/blog/incident/. Accessed: May 26, 2022.

[17] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing Smart Contract with Runtime Validation. 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, 438–453. https://doi.org/10.1145/3385412.3385982

[18] 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

[19] Tai D. Nguyen, Long H. Pham, and Jun Sun. 2021. sGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. arXiv:2101.01917 [cs.CR]

[20] OokiTrade. 2020. iToken LoanTokenLogicStandard smart contract. https://github.com/OokiTrade/contractsV2/blob/bf95cbe373d4e972da5e93daf8ddb0f3886e78a1/contracts/connectors/loantoken/LoanTokenLogicStandard.sol#L279. Accessed: May 26, 2022.

[21] DeFi Pulse. 2022. DeFi - The Decentralized Finance Leaderboard at DeFi Pulse. https://defipulse.com/. Accessed: May 26, 2022.

[22] REKT. 2021. VALUE DEFI - REKT 2. https://rekt.news/value-rekt2/. Accessed: May 26, 2022.

[23] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2020. EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts. arXiv:2010.00341 [cs.CR]

[24] SmartBugs. 2021. SmartBugs Confused_Sign Wallet smart contract. https://github.com/smartbugs/smartbugs/blob/master/dataset/access_control/wallet_04_confused_sign.sol. Accessed: May 26, 2022.

[25] SmartBugs. 2021. SmartBugs EtherBank smart contract. https://github.com/smartbugs/smartbugs/blob/master/dataset/reentrancy/etherbank.sol. Accessed: May 26, 2022.

[26] SmartBugs. 2021. SmartBugs Refund_NoSub smart contract. https://github.com/smartbugs/smartbugs/blob/master/dataset/access_control/wallet_02_refund_nosub.sol. Accessed: May 26, 2022.

[27] SmartBugs. 2021. SmartBugs Unprotected smart contract. https://github.com/smartbugs/smartbugs/blob/master/dataset/access_control/unprotected0.sol. Accessed: May 26, 2022.

[28] Palina Tolmach, Yi Li, and Shang-Wei Lin. 2022. DeFinery: Online Supplementary Material. https://sites.google.com/view/ase2022-definery/. Accessed: May 26, 2022.

[29] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. 2021. Formal Analysis of Composable DeFi Protocols. In *Financial Cryptography and Data Security. FC 2021 International Workshops*. Springer Berlin Heidelberg, Berlin, Heidelberg, 149–161.

[30] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (jul 2021), 38 pages. https://doi.org/10.1145/3464421

[31] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2021. Elysium: Automagically Healing Vulnerable Smart Contracts Using Context-Aware Patching. arXiv:2108.10071 [cs.CR]

[32] 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

[33] 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