Property-Based Automated Repair of DeFi Protocols

Anonymous Author(s)

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 irrecoverable 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, financial or otherwise. 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 nine real-world and benchmark smart contracts reveal that DeFinery efficiently navigates the search space and generates higher-quality patches that cannot be obtained by other smart contract APR tools.

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 [22], it becomes critical to ensure that their implementations are free from bugs and vulnerabilities. Yet, the adoption of DeFi protocols has been accompanied by a number of security exploits leading to billions of dollars being stolen from the underlying smart contracts. According to a recent report [13], 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 [4, 6, 14, 17, 23]. 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 [16, 20, 24, 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, 5, 30].

```
 1   contract iToken ... {
 2       ...
 3       function _internalTransferFrom(
 4           address _from, address _to,
 5           uint256 _value, ...) internal {
 6           ...
 7 +         require(_to != _from);
 8           uint256 balancesFrom = balances[_from];
 9           uint256 balancesTo = balances[_to];
10
11           require(balancesFrom >= _value);
12           uint256 balancesFromNew = balancesFrom - _value;
13           balances[_from] = balancesFromNew;
14
15           uint256 balancesToNew = balancesTo + _value;
16           balances[_to] = balancesToNew;
17       }
18   }
```

Listing 1: Simplified source code of iToken [21]

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 the interactions between numerous economic agents in the DeFi ecosystem—regular blockchain users and other DeFi protocols. Discrepancy between the intended behavior and smart contract implementation, therefore, may enable exploitative and harmful user behaviors, such as buying tokens at an abnormal exchange rate [14] or getting them for free [4], borrowing more tokens than should be allowed [6], and many others. Some of these issues only manifest in a violation of a high-level functional specification defined for a particular DeFi protocol, and cannot be encoded as a low-level source 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 [17] that we use to illustrate our technique (Listing 1). Due to the specifics of users' balance update (lines 12-16), 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 [17]. 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 vulnerable smart contract, it is equally critical that the remaining—valid—behavior satisfying the property is preserved by the patched version. 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 7 in `_internalTransferFrom` implementation (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 interacting smart contracts with one of them known to be vulnerable), (2) a property, and (3) a trace leading to the property 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 it is hard to obtain this level of flexibility using pre-defined mitigation 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 source-level 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, our technique 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 7, or equivalently can be inserted after line 16).

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 [4, 6, 14, 17, 23] 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 [9].

2 RELATED WORK

The majority of the existing tools for smart contract repair are only able to patch a limited number of known types of vulnerabilities. These tools include SmartShield [33], sGuard [20], EVMPatch [24], Elysium [31], Aroc [16], and HCC [12]. Most of these tools 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. In addition, it 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, therefore, 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 [19] or enforcing runtime validity of smart contracts with respect to the user-provided invariant [18]. 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 also 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 an SMT-solver (z3 [7]). 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 formula over a set of input parameters 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 heuristics. (4) For each generated candidate, it invokes the symbolic execution component to check the test cases and analyzes the results. If all test cases pass, we assume the patch might be plausible. In this case, (5) the *SA* module is used to build a symbolic summary once again—this time 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: https://sites.google.com/view/ase2022-definery/. In order 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 the iToken smart contract (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* smart contract's address.

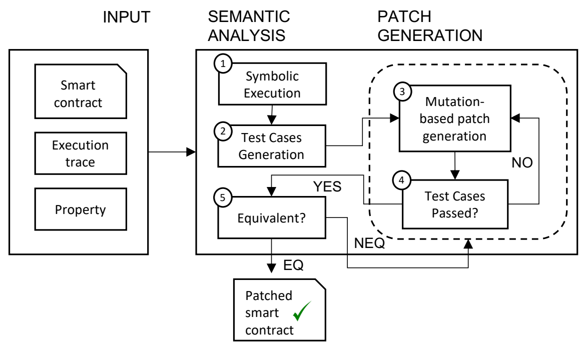

Figure 1: Tool architecture.

For simplicity, we assume that `_to` can be an address of one of three smart contracts: *User*, *Main*, and *iToken*. This assumption is reflected in line 5 of Listing 2 representing constrains on the symbolic address variable `TO`. Parameter `_value` corresponds to a symbolic variable `VAL`, which can be any positive unsigned integer.

The harness ends with a declaration of a property in the form of an assertion, which can be constructed from variables available in the *Main* contract and Solidity operators. To check that the sum of the token balances before and after a transfer is the same (line 10), we record the balances of mentioned three contracts before (lines 3-4) and after (lines 8-9) the execution of `transferTo()` (line 7).

```
1  function _harness_() {
2      ...
3      uint256 init_sum = balances[User] + balances[Main]
4                         + balances[iToken];
5      assume(TO == User || TO == Main || TO == iToken);
6      // Test Case: TO = Main; VAL = 100;
7      User.transferTo(TO, VAL);
8      uint256 res_sum = balances[User] + balances[Main]
9                         + balances[iToken];
10     declare_property(init_sum == res_sum);
11 }
```

Listing 2: Harness Function Example

For the iToken example, symbolic execution generates five valid execution paths that satisfy the property and one invalid path that violates it, which is sufficient to find a correct fix. For each such path, DeFinery generates a test case by assigning concrete values to symbolic variables in the harness function. Line 6 in Listing 2 shows one such instrumentation corresponding to a valid execution 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 analyzed trace, our symbolic engine maintains a symbolic state for each possible path storing a path condition—a first-order 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 6 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 13 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 $(TO == User \land balancesFrom \geq VAL)$, is not included in a summary but is recorded for future use.

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

In addition to the listed functionality, we facilitate fixing typical issues such as reentrancy [30] by labeling traces that exhibit reentrant behavior, i.e., contain an external call and a callback to the same or another contract. This pattern helps efficiently process both same- and cross-contract reentrancy, as shown in Sect. 4.

Patch Generation. The patch generation module of DeFinery is based on a modified version of SCRepair [32]. Our implementation extends a set of statements that are synthesized by SCRepair, integrates our semantic analysis module for patch evaluation, and implements performance-improving heuristics for selecting changes that would likely to 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) or any expression appearing in the same function. Since our fault localization is function-level, 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 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 attempt to 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. The patch generated in this case can be seen in lines 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 have been fixed (became valid) and (2) the number of valid traces that remain valid. We also use the patch simplicity fitness function from SCRepair, which prioritizes patches with smaller number of 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 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).

Finally, we build an equivalence assertion between symbolic summaries of the original and patched smart contracts—S (Eq. (1))

Table 1: A summary of DeFinery evaluation.

#	Smart Contract	Patch	Property	Result		
				DeFinery	sGuard	SmartShield
1	xForce [11]	`+ require(result);`	User didn't receive xForce if he didn't provide any Force	✓	✗	✓

2	Confused_Sign [25]	`- 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 [8]	`+ initialized = true;`	The staked token can't be changed	✓	✗	✗

4	Uranium [15]	`require(balance0 * balance1 >=`
		`- _res0 * _res1 * 10**2);`
		`+ _res0 * _res1 * 100**2);`
			(Constant) product of pool reserves is non-decreasing	✓	✗	✗

5	Refund_NoSub [27]	`+ balances[msg.sender] = 0;`	Sum of balances is constant; the user can receive a refund	✓	✗	✗
6	Unprotected [28]	`+ require(owner == msg.sender);`	Owner can only be changed to a trusted address	✓	✗	✗
7	iToken [21]	`+ require(from != to);`	Constant sum of balances is preserved by a *transfer*	✓	✗	✗

8	cToken [10]	`- amp.transfer(borrower, amount);`
		`borrowBalance[borrower] += amount;`
		`+ amp.transfer(borrower, amount);`
			Protocol balance can't decrease	✓	✗	✗

9	EtherBank [26]	`- msg.sender.call.value(amount);`
		`userBalances[msg.sender] = 0;`
		`+ msg.sender.call.value(amount);`
			User's sum of balances is constant	✓	✓	✗

and S' (Eq. (2)), respectively. An equivalence assertion is a first-order logic formula Φ that helps determine logical and, therefore, functional equivalence between S and S': $\Phi = \neg(S \Leftrightarrow S')$ [3]. We provide this formula to an SMT-solver, which either proves that Φ cannot be satisfied, meaning that the smart contract executions are equivalent, or finds a counterexample, which indicates that smart contracts produce different outputs for at least one input, and, therefore, are not equivalent—in this case, we continue the patch generation process. If a solver cannot find a counterexample, the analyzed smart contract behaviors are considered equivalent, and we conclude that the patch is correct.

$$\begin{aligned}S' = (&\mathit{TO} == \mathit{Main} \land \mathbf{TO} \neq \mathbf{User} \land \mathit{balancesFrom} \geq \mathit{VAL} \land \\ &\neg(\mathbf{TO} == \mathbf{User} \land \mathbf{balancesFrom} \geq \mathbf{VAL}) \land \\ &\mathit{balances}[\_\mathit{from}] \mathrel{-}= \mathit{VAL} \land \mathit{balances}[\_\mathit{to}] \mathrel{+}= \mathit{VAL}) \lor \\ (&\mathit{TO} == \mathit{Main} \land \mathbf{TO} \neq \mathbf{User} \land \mathit{balancesFrom} < \mathit{VAL} \land \\ &\neg(\mathbf{TO} == \mathbf{User} \land \mathbf{balancesFrom} \geq \mathbf{VAL})) \lor \ldots\end{aligned} \tag{2}$$

4 PRELIMINARY EVALUATION

In this section, we report the results of our evaluation on 9 smart contracts that include 5 DeFi smart contracts that were previously exploited and 4 smart contracts taken from the SmartBugs [9] dataset. The choice of experimental subjects aims to demonstrate that our technique is applicable to both DeFi and regular smart contracts. It also shows that issues caused by typical vulnerabilities can be fixed by DeFinery too. Some of the smart contracts are simplified to allow processing them with our symbolic engine. Their source code and results are available on our website [2]. 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 result of running our tool on 9 experimental smart contracts, showing the correct patch found by DeFinery and the property that was used to find and assess the fix. The results demonstrate a variety of patches that fix both common smart contract issues ("unchecked send" in (1), reentrancy in (8,9)) as well as missing or wrong pre- and post-conditions (2,4,6,7) and missing variable updates (3,5). On average, it took DeFinery 53 seconds to find a correct patch. We repeated every experiment five times.

We compare DeFinery to sGuard [20] and SmartShield [33]. Other smart contract repair tools, such as EVMPatch [24] and Elysium [31], are not available at the time of the evaluation. Although we have reused part of the SCRepair [32] implementation, it required substantial modification of its implementation to be compiled, therefore, we could not use it for comparison. Both sGuard and SmartShield can only fix typical smart contract vulnerabilities and cannot process most of the 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 the xForce smart contract (1).

The $50M bug in Uranium Finance (4) caused by using an incorrect constant [14] is a good example of the issue that can be fixed by our technique but not existing smart contract repair tools. It cannot be detected 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 an inefficient search strategy due to the complexity of the correct statement. In conclusion, the evaluation results show that our technique is capable of efficiently repairing the smart contract that otherwise cannot be handled by other existing tools, while also preserving correctness of the remaining behavior.

5 CONCLUSION AND FUTURE WORK

In this work, we formulate the problem of property-based automated program repair of smart contracts. We propose an efficient approach that makes a first attempt at fixing violations of functional specification in DeFi or regular smart contracts. In addition, we demonstrate that combining semantic inference with search-based patch generation is a promising direction for smart contract repair.

Our future work includes improving fault localization to enable more effective patch generation as well as extending the considered search space for patches. We also plan to expand the experimental dataset and evaluate the impact of our patches on gas consumption. Finally, we consider integrating a verification component that would find the trace leading to the property violation automatically.

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] Anonymous Author(s). 2022. DeFinery: Online Supplementary Material. https://sites.google.com/view/ase2022-definery/. Accessed: May 26, 2022.

[3] 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

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

[5] 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

[6] 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.

[7] 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.

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

[9] 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

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

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

[12] 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]

[13] 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.

[14] 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.

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

[16] 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

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

[18] 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

[19] 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

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

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

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

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

[24] 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]

[25] 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.

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

[27] 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.

[28] SmartBugs. 2021. SmartBugs Unprotected smart contract. https://github.com/smartbugs/smartbugs/blob/master/dataset/access_control/unprotected0.sol. 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