Demystifying Exploitable Bugs in Smart Contracts (Supplementary Material)

***Abstract*—Exploitable bugs in smart contracts have caused significant monentary loss. Despite the substantial advances in smart contract bug finding, exploitable bugs and real-world attacks are still trending. In this paper we systematically investigate 516 unique real-world smart contract vulnerabilities in years 2021-2022, and study how many can be exploited by malicious users and cannot be detected by existing analysis tools. We further categorize the bugs that cannot be detected by existing tools into seven types and study their root causes, distributions, difficulties to audit, consequences, and repair strategies. For each type, we abstract them to a bug model (if possible), facilitating finding similar bugs in other contracts and future automation. We leverage the findings in auditing real world smart contracts, and so far we have been rewarded with $102,660 bug bounties for identifying 15 critical zero-day exploitable bugs, which could have caused up to $22.52 millions monetary loss if exploited.**

I. Description of Each DeFi Category

In this section, we provide the detailed explanation of each DeFi category.

Lending. Lending projects allow users to borrow or lend assets. Lenders deposit their assets into the project and earn interests, while borrowers borrow assets by providing collaterals. Borrowers need to pay for interests as well. Collateral and loan are different assets, e.g., WETH and USDC.

Dexes. *Dex* is short for *Decentralized Exchanges*. Such projects allow users to exchange assets. Uniswap, which we have intensively discussed in the main text, is one of the most well-known Dex projects.

Yield. Yield projects reward users for their staking. Specifically, users deposit their funds into yield projects, and the projects leverage the funds to invest in other places. All users (of a yield project) share the profits of investment, according to their staking.

Services. Service projects provide basic functionalities that can be used by other projects. For example, governance voting is popular in DeFi and can be provided as a service. Another common service is to lock funds of seed investors and founders for a pre-defined period (i.e., tokenization).

Derivatives. Derivative projects are similar to their counterparts in the traditional finance, which derive their value from the performance of an underlying entity. Futures [1] and Options [2] are two typical derivative products.

Yield Aggregator. Such projects aggregate profits from other yield projects. Specifically, given a large number of yield projects using diverse investment strategies, yield aggregator projects manage user funds (to deposit into a portfolio of yield projects) for optimal profit.

Real World Assets. Such projects bridge real-world assets into blockchains. Specifically, these projects provide products whose value are decided by real-world assets, e.g., stocks. Users can invest on these products like in the physical world.

Stablecoins. Stablecoins are a special kind of crypto-currency. Their market value is pegged to some real-world currency (e.g., US Dollar), and hence considered *stable* (e.g., not as volatile as ETH and BTC). There are two mainstream types of stablecoins. One is called asset-backed stablecoins, minting which requires to reserve the same amount of reference assets in a central issuers or a smart contract. Another is called algorithmic stablecoins, whose price stability results from the use of specialized algorithms.

Indexes. Similar to their counterparts in the physical world, such projects track the performance of a group of assets in a pre-defined (and usually standardized) way. For example, stock market index [3] is commonly used to measure a specific stock market and helps investors understand the current stock price level.

Insurance. It is another typical financial product in both the traditional finance and DeFi. Users pay the insurance fee for the protection of their assets.

NFT Marketpalce. Non-fungible tokens (NFT) can be sold and bought like houses and paintings in the physical world. Such projects provide platforms for users to trade their NFTs.

NFT Lending. NFT lending projects allow users to deposit their NFTs as collateral for loans. Different from lending projects mentioned before, NFT lending projects additionally need to properly quote for NFTs. Recall that the value of NFTs varies from token to token.

Cross Chain. Cross-chain projects provide interoperability among different blockchains. Note that smart contracts on different blockchains (e.g., Ethereum and Binance Smart Chain) cannot communicate with each other by design. Cross-chain projects connect these smart contracts (on different blockchains) by a secured and trusted off-chain service.

II. Description of Machine Auditable Bugs

In this section, we discuss different types of machine auditable bugs.

Assertion Failure. Like in traditional software, an assertion failure denotes a condition violation in a user-provided assertion statement.

Arbitrary Write. Smart contracts store data in *storage* (similar to *memory* in traditional software). Contracts are responsible for ensuring only authorized users (or contracts) can access sensitive data. The problem arises when an adversary can arbitrarily write some storage locations and thus modify sensitive data (e.g., the owner of the contract).

Block-state Dependency. *Ether* (also known as ETH) is the native crypto-currency of Ethereum. Users and smart contracts can transfer Ether to each other. For a subject contract, if how much Ether it transfers depends on some block states, this contract is likely vulnerable to miners' manipulation. Note that most block states are controlled by minters. For example, miners can decide when to execute a specific transaction (i.e., `block.timestamp`), regardless of the gas price proposed by that transaction.

Compiler Error. There are known bugs in some outdated Solidity compilers [4] which will induce incorrect compilation results, i.e., misbehaved contracts. The problem arises when contracts are compiled by these compilers.

Control-flow Hijack. Solidity supports *callback functions*. A callback function is a function passed into another function as an argument and invoked inside later. The problem arises once a user can control a callback function so that he can execute arbitrary code.

Ether Leak. As aforementioned, Ether is the native crypto-currency of Ethereum. The problem arises once a user can freely retrieve Ether from a subject contract, i.e., stealing Ether deposited by other users.

Freezing Ether. The problem arises when a large portion of Ether is locked in the contract. In other words, user funds (in Ether) are permanently frozen.

Gas-realted Issue. Ethereum has a maximum gas limit for each transaction. The problem arises when a function requires more computation units that exceed the gas limit. In other words, the function will always revert due to insufficient gas.

Integer Bug. Integer bugs denote integer overflows or underflows.

Mishandled Exception. When calling an external function, the smart contract fails to validate the return status.

Precision Loss. The problem arises when a precision loss is significantly amplified during calculation, rendering the final result largely deviated from the expected one.

Reentrancy. The problem arises when a function in a victim contract is re-entered, leading to an inconsistent state of that contract. Reentrancy bugs are well-studied by existing works. According to our study, 29 out of 38 existing techniques are able to detect reentrancy bugs.

Suicidal Contract. Smart contracts can be destructed by the `selfdestruct` operation. If such a privileged operation can be invoked by users, an adversary can intentionally destroy the contract and leave all other users' funds permanently locked.

Transaction-ordering Dependency. The problem arises when a contract's outcomes are dependent on the order of transactions. Note that miners can decide the order of transactions and are hence able to manipulate the outcomes.

Transaction Origin Use. The problem arises when a contract uses `tx.origin` (i.e., the address of the user who proposes the current transaction) for authorization. Consider a vulnerable contract V who uses `tx.origin` for authorization, an authorized user A, and a malicious contract M. If A calls into contract M and M further calls into V, A's privilege will be granted to `M` since `tx.origin` returns the address of A.

```
1  contract UniswapV2Pair {
2
3    IERC20 token0; IERC20 token1;
4    uint reserve0; uint reserve1;
5
6    function swapToken0ForToken1(
7      uint amount1Out, address to
8    ) external {
9      token1.transfer(to, amount1Out);
10
11     IUniswapV2Callee(to).uniswapV2Call();
12
13     uint balance0 = token0.balanceOf(address(this));
14     uint balance1 = token1.balanceOf(address(this));
15
16     uint amount0In = balance0 - (reserve0 - amount0Out);
17     uint balance0Adj = balance0 * 1000 - amount0In * 3;
18
19     require(
20       balance0Adj * balance1 >=
21         reserve0 * reserve1 * 1000,
22       "insufficient funds transferred back"
23     );
24
25     reserve0 = balance0; reserve1 = balance1;
26   }
27 }
```

Fig. 1: The swap function of Uniswap

Uninitialized Variable. Solidity assigns a default value for each uninitialized variable, according to the variable type (e.g., 0 for `uint`). The problem arises when developers use an uninitialized variable (intentionally or unintentionally) but assume a wrong default value.

Weak PRNG. The program arises when a subject contract uses a pseudo-random number generator (PRGN) that relies on predictable variables, which breaks the randomness. Note that many variables are counter-intuitively predictable on Ethereum blockchain. For example, `block.timestamp` in a victim function can be predicted by an adversary if he wraps a function and the victim function into a single transaction (so that the two functions share the same `block.timestamp`).

III. Price Oracle Manipulation

A. Uniswap

Fig 1 presents a code snippet of Uniswap's swap function, which instantiates the aforementioned exchange rule. It is critical to understand this function as most existing price oracle exploits entail manipulating this contract *in a legal way*. The code is simplified for the illustrative purpose, and hence slightly differs from the real implementation. Starting from line 1, the code declares a contract `UniswapV2Pair`. Lines 3 and 4 define several state variables, including `token0` and `token1` denoting the two assets for exchange, and `reserve0` and `reserve1` standing for the reserve balances of `token0` and `token1`, respectively. A user invokes function `swapToken0ForToken1()` starting from line 6 to exchange `token0` for `token1`, by specifying the amount of `token1` she demands, namely `amount1Out`, and her address `to` to receive `token1` and pay with `token0`. The transaction is between the user and Uniswp which owns both

tokens for trading. The main body of the function is divided into three phases, transferring `token1` (line 9), receiving `token0` (line 11), and verifying the constant-product invariant (lines 13-23), respectively. To transfer `token1` to the user's contract, at line 9, a standard `transfer` function of *ERC20* is invoked, which essentially transfers a specified amount of the underlying asset from the UniswapV2Pair contract to the user. At line 11, an external function call, i.e., `uniswapV2Call`, happens upon the user's contract, within which the user transfers a certain amount of `token0` back to Uniswap. The use of external call enables *flash-loan*, a powerful and unique feature of DeFi. We will elaborate more on flash-loan later in the section. Starting from line 13, the contract verifies whether the constant-product invariant is guaranteed after receiving the user's fund, i.e., whether the user sends back a sufficient amount of `token0`. Variables `balance0` and `balance1` denote the current balances of assets (lines 13 and 14), based on which the amount of received `token0` can be calculated (`amount0In` at line 16). Uniswap charges a contract fee of 0.3%, reflected as `balace0Adj` at line 17. Note that `balance0Adj` denotes the amount of the current balance after charging the contract fee with a multiplier of 1000. In lines 19-23, the contract compares the product of reserve balances before and after the exchange. If the check fails, i.e., the user does not pay a sufficient amount of `token0`, the exchange fails with the whole transaction reverted. Given the atomicity of block-chain transactions, the token transfers at lines 9 and 11 get reverted as well, without affecting the user's and Uniswap's funds. Also note that the user is allowed to transfer more funds back, which is profitable for the contract. The reserve balances `reserve0` and `reserve1` are updated accordingly at line 25. There is another function `swapToken1ForToken0` for the exchange in the opposite direction.

Example. Consider a Uniswap pair of WETH and USDC with reserve balances 100 and $400,000$, respectively. The current price of WETH in Uniswap is hence $\$4,000 = 400,000/100$. Assume due to the high volatility of cryptocurrency, the price of WETH (in the rest of the world) drops drastically to $\$1,000$. Assume Alice plans to swap out $100,000$ USDC from Uniswap by invoking the contract's exchange function. According to the constant-product invariant, the contract needs to hold $100 \times 400,000/(400,000 - 100,000) \approx 133$ WETH after the external call. That is, Alice is required to send $133 - 100 = 33$ WETH back to the contract. Taking the 0.3% contract fee into consideration, the total amount that Alice needs to pay is only $33/(1-0.003) \approx 33.1$ WETH. It becomes extremely profitable for Alice, since she pays 33.1 WETH (worth $\$33100$ since the current real-world price of WETH is $\$1,000$) but gets $100,000$ USDC (worth $\$100,000$) back. The profit incentive continually attracts arbitragers in the wild and pushes the Uniswap pair towards the balanced status of an WETH price of $\$1,000$, i.e., with $200,000$ USDC and 200 WETH reserves, explaining Uniswap's business model. □

```
1  function swap(uint amount1Out, address to) external {
2    token1.transfer(to, amount1Out);
3    IUniswapV2Callee(to).uniswapV2Call();
4
5    uint balance0 = token0.balanceOf(address(this));
6    uint balance1 = token1.balanceOf(address(this));
7    uint amount0In = balance0 - (reserve0 - amount0Out);
8    uint balance0Adj = balance0 * 10000 - amount0In * 22;
9    require(
10     balance0Adj*balance1 >= reserve0* reserve1* 1000,
11     "insufficient funds transferred back");
12   reserve0 = balance0; reserve1 = balance1;
13 }
```

Fig. 2: The LFW ecosystem exploit

B. Example of Flash Loan

Uniswap inherently supports flash loans. Specifically, in Figure 1, Alice specifies the debt amount as `amount1Out` and gets the funds at line 9. Within the external call at line 11, Alice not only trades for arbitrage (or launches the aforementioned exploit) with the borrowed `token1`, but also pays the debts after the trading (or exploit). After line 11, both `balance0` and `balance1` remain unchanged, satisfying the repayment check in lines 19-24.

IV. Erroneous Accounting

This type of bugs are due to incorrect implementation of the underlying financial model formulas. They are difficult to find due to the substantial domain knowledge needed.

Example. Figure 2 presents a code snippet of the LFW ecosystem, which has been exploited and lost $\$0.21$ millions. LFW is an AMM contract that allows exchange of two types of assets. A user invokes function `swap()` to exchange `token0` (the first asset) for `token1` (the second) with LFW. Variables `balance0` and `balance1` denote the instant balances of the respective tokens, and `reserve0` and `reserve1` their reserve balances (i.e., committed balances). At line 1, the user specifies the amount of `token1` she demands, namely `amount1Out`, and her address `to` to receive `token1` and send `token0`. The main body of the function is divided into three phases, transferring `token1` (line 2), receiving `token0` (line 3), and verifying the constant-product invariant (lines 5-11), respectively. At line 9, the contract verifies whether the user sends back a sufficient amount of `token0` such that the constant-product invariant is respected. The amount of received `token0` can be calculated from the instant and the reserve balances (`amount0In` at line 7). LFW charges a contract fee of 0.22%, reflected as `balace0Adj` at line 8. The developers use a multiplier of $10,000$ at line 8 (to avoid expensive floating point computation). Lines 9-11 are supposed to check the invariant. However, the developers use a wrong multiplier $1,000$. Since the asset prices are determined by the ratio of their reserve balances, this bug leads to substantial pricing errors. Consider the actual invariant checked by the contract, i.e., $(\texttt{balance0} \times 10 - \texttt{amount0In} \times 0.022) \times \texttt{balance1} \geq \texttt{reserve0} \times \texttt{reserve1}$ (reduced from $(\texttt{balance0} \times 10000 - \texttt{amount0In} \times 22) \times \texttt{balance1} \geq \texttt{reserve0} \times \texttt{reserve1} \times 1000$). The adversary pays only

```
1  contract NFTMarketReserveAuction{
2    mapping(address => mapping(uint => uint)) auctionIds;
3    mapping(uint => ReserveAuction) idAuction;
4    uint auctionId;
5
6    function createReserveAuction(
7      address nftContract, uint tokenId) external ...{
8       auctionId++;
9       _transferToEscrow(nftContract, tokenId);
10      auctionIds[nftContract][tokenId] =
11        auctionId;
12      idAuction[auctionId] = NewAuction(
13         msg.sender, ..., tokenId, ...);
14      ...
15   }
16   function _transferToEscrow(
17     address nftContract, uint tokenId) internal ...{
18      uint auctionId =
19        auctionIds[nftContract][tokenId];
20      if (auctionId == 0) { // NFT is not in auction
21       super._transferToEscrow(nftContract, tokenId);
22       return;
23     } ...
24   }
25 }
```

Fig. 3: The NFTMarketReserveAuction exploit

one tenth of the expected `token0` to get `token1` he demands. □

Abstract Bug Model and Remedy. It is hard to derive general abstract models for erroneous accounting bugs, since such bugs are project/implementation specific. This makes them difficult to find. However, it is encouraging to see that audit contests provide an effective way to expose them (these bugs are the most popular kind among the Code4rena bugs, due to the very broad domain expertise brought by the participants. The fixes are usually simple, including changing coefficients and rephrasing arithmetic expressions.

V. ID Uniqueness Violations

These bugs are caused by violations of the uniqueness property of ID fields. They are the 3rd most popular bugs in the auditing stage.

Example. This is a real case from an auction contract enlisted for audit in Code4rena [5]. A user acting as a seller can put their NFT up for auction. The bug was caught by only one auditor. If exploited, it could make a winning bidder's funds locked within the smart contract with no direct way of recovery. As shown in Figure 3, to initiate an auction, the seller calls the function `createReserveAuction` at line 6 with the parameters of the address of their NFT contract (`nftContract`) and the ID of the token they are selling (`tokenId`). At line 9, the token is transferred to escrow via function `_transferToEscrow` (defined at line 16). Inside the function, it looks up an auction using the seller's address and the token ID (line 19). It then checks if this is a new auction by checking if `auctionID == 0` on line 20. If so, the NFT is transferred to the escrow via the super class function at line 21. Then the token is marked as in storage at line 10 via `auctionIds` and a new auction is created at line 12 via `NewAuction` with all the necessary parameters. The bug lies in that the developers are essentially using the seller address and the NFT token ID to denote an auction (their intention can be inferred from line 19). However, they do not ensure the uniqueness of these data fields.

```
1  contract SushiTrident{
2    uint128 internal reserve0;
3    uint128 internal reserve1;
4
5    function burn(bytes calldata data)
6      public override lock returns (...) {
7      (... , uint128 amount, address recipient...) =
8        abi.decode(data,(int24, int24, ..));
9      // calculates amounts of each reserve to be returned
10     (uint amount0, uint amount1) =
11       _getAmountsForLiquidity(..., amount);
12     //calculate fees to burn from amounts to burn
13     (uint amount0fees, uint amount1fees) =
14       _updatePosition(msg.sender, ..., -amount);
15     ...
16     reserve0 -= uint128(amount0fees);
17     reserve1 -= uint128(amount1fees);
18     //returns the reserve tokens
19     _transferBothTokens(recipient, amount0, amount1,...)
20     ...
21   }
22 }
```

Fig. 4: The SushiTrident exploit

It can hence be exploited as follows. A (malicious) seller invokes `createReserveAuction` (line 6) twice using the same `nftContract` and `tokenId`. The first invocation correctly transfers the NFT and creates the auction. In the second invocation, the lookup at line 19 simply yields the previously created auction and the check at line 20 falls though (not reverting). A new (duplicate) auction is created at line 12. Now, there are two auctions for the same NFT. Then the adversary cancels the first auction to get the NFT back. However, bidders are still bidding in the duplicate auction. Eventually, someone wins the auction and the contract is supposed to transfer the NFT to the winner. However, since the NFT is already gone, the transfer reverts all the time. Essentially, all the highest bidders' funds are locked in the contract forever. The developers fixed the bug by adding a check: before a `NewAuction` is created, the storage of `toTokenIdToAucionId` (line 10) is checked to make sure that the `tokenID` of the NFT on auction has not been placed in the auction storage before. □

Abstract Bug Model and Remedy. Variables or data structure fields are used as the ID for some entity/asset and the access to some critical operation (e.g., creating/cancelling an auction) is granted based on the ID. However, developers do not check the uniqueness of ID fields. Although this type of bugs is not difficult to find in general, it may require nontrivial efforts to infer if some variables are intended to be an ID, especially when the variable names are not informative. Sometimes, the guarded operation is implicit and distant from the ID check. Developers usually fix these bugs by checking for duplication.

VI. Inconsistent State Updates

In this type of bugs, developers forget to have correlated updates when they update some variable(s), or the updates do not respect their inherent relations.

Example. Figure 4 is an example of real world exploit that was caught during a Code4rena audit. The bug was caught

by only one auditor. The contract itself is an AMM for exchanging two types of tokens, and also a part of the Sushi organization, which has a market capitalization of $171 millions [6]. At lines 2-3, `reserve0` and `reserve1` are the reserve balances of the two tokens. The bug is in the `burn` function, beginning at line 5, which is supposed to burn a specific type of ownership token called LP Token, a separate token which designates ownership (shares) of the entire pool of the two reserve tokens, and return the amounts of the reserve tokens corresponding to the burned ownership. This is analogous to cashing out stocks in the physical world. First at line 7, variables `amount`, representing the amount of LP tokens to burn, and `recipient` address, are unwrapped from the `data` parameter. Then, based on the `amount` (of LP token) (`amount0`, `amount1`) of the two reserve tokens are generated at line 10. Fees for the burn operation are decided at line 13. Eventually, the amounts of each reserve token are updated at lines 16-17. The burner is then transferred the amounts of each token at line 19. Observe that only the fees are subtracted from the reserves, and not the actual reserves returned to the receiver. As such, there appears to be more tokens within the contract than there actually are, leading to all sorts of problems like incorrect pricing. Developers patched this bug by subtracting `amount0` and `amount1` as well. The essence is that when the reserves are updated by the fees, they should be updated by the burned amounts as well. □

Abstract Bug Model and Remedy. Without losing generality, there are two variables x and y, their operations (e.g., reads, writes, and arithmetic operations) tend to co-occur due to inherent (and often implicit) relations, such as `amount0` and `amount0fee` in our example. However, developers forget some operations that are supposed to co-occur. Inferring such co-occurrence relations is the key to detecting these bugs. The fixes entail adding/correcting updates.

VII. Atomicity Violations

This type of bugs is caused by the inference between concurrent business flows that are supposed to have high level atomicity (higher than the transaction level atomicity).

Example. eFig. 5 presents a real-world vulnerability in th PancakeSwap lottery contract [7]. It was reported by an anonymous whitehat and awarded with an undisclosed bounty [8]. Like lottery in the physical world, the contract users can buy tickets and the owner randomly draws a winner every day. Lines 3-6 define the key state variables, including a three-level mapping `tickets` indicating the amount of each ticket bought by each user (multiple tickets of the same ID can be bought by the same or different users), the winner (`winningId`), and a boolean variable indicating whether the owner is drawing the winner (`drawing`). Function `reset` (line 9) is a privileged function for the owner to start a new round. Function `buy`, starting from line 13, allows users to buy tickets of a specified ID. It first checks that the owner is not drawing and has not drawn the winner, at lines 14 and 15, and further processes the payment and updates `tickets` accordingly. At line 19, function `enterDrawingPhase` is used to

```
1  contract Lottery {
2    //  user address -> lottery id -> count
3    mapping (address => mapping(uint64 => uint))
4      public tickets;
5    uint64 winningId; // the winning id
6    bool drawingPhase; // whether the owner is drawing
7
8    // invoked every day to reset a round
9    function reset() external onlyOwner {
10       delete tickets;
11       winningId = 0; drawingPhase = false;
12   }
13   function buy(uint64 id, uint amount) external {
14     require(winningId == 0, "already drawn");
15     require(!drawingPhase, "drawing")
16     receivePayment(msg.sender, amount),
17     tickets[msg.sender][id] += amount;
18   }
19   function enterDrawingPhase() external onlyOwner {
20     drawingPhase = true;
21   }
22   // id is randomly chosen off-chain, i.e., by chainlink
23   function draw(uint64 id) external onlyOwner {
24     require(winningId == 0, "already drawn");
25     require(drawingPhase, "not drawing");
26     require(id != 0, "invalid winning number");
27     winningId = id;
28   }
29   // claim reward for winners
30   function claimReward() external {
31     require(winningId != 0, "not drawn");
32     ...
33   }
34   function multiBuy(uint64[] ids, uint[] amounts)
35     external {
36     require(winningId == 0, "already drawn");
37     uint totalAmount = 0;
38     for (int i = 0; i < ids.lengths; i++) {
39       tickets[msg.sender][ids[i]] += amounts[i];
40       totalAmount += amounts[i];
41     }
42     receivePayment(msg.sender, totalAmount),
43   }
44 }
```

Fig. 5: The PancakeSwap Lottery vulnerability

start the lottery drawing phase. Variable `drawingPhase` is essentially a lock for the variable `tickets` to prevent further ticket purchase in this round. After entering the drawing phase, function `draw` (lines 23-28) is invoked to set the winner, which enables `claimReward`. There are three business flows, i.e., buying ticket, drawing winner, and claiming prizes. Note that the business flow of drawing winner comprises two functions (`enterDrawingPhase` and `draw`), and hence two transactions. Such a design is critical. Otherwise, an adversary could observe the winner from the `draw` transition in the *mempool*, and bought a huge amount of tickets with the winner's ID. Note that before being mined and finalized on blockchain, transactions are placed in a mempool and visible to the public [9]. Besides, since paying a high gas fee provides incentives for miners, it allows the adversary to preempt the `draw` transaction with his own, and eventually earning a lot of profit illegally. The contract properly prevents this by separating the business flow to two transactions and using a lock `drawingPhase` to ensure atomicity. However, another purchase function `multiBuy` (lines 34-43) does not respect such atomicity. It is a gas-friendly version of function `buy` which allows buying multiple tickets at a time. It updates `tickets` accordingly within the loop in lines 39-40, and receives the payment for all tickets at line 41. However, it does

not use the `drawingPhase` lock, making the aforementioned attack possible. This exploit method (preempting a pending transaction belonging to an atomic business flow by paying a higher gas fee) is also called *front running* [10], whose root cause is usually atomicity violation. □

Abstract Bug Model and Remedy. There are multiple business flows $\mathcal{B}_1$, ... and $\mathcal{B}_m$ that access some common state variables (e.g., *tickets* in our example). An atomicity violation bug occurs when concurrent business flows yield unserializable outcome [11] In our example, after front-running, the amount of tickets for the winner ID is substantially inflated after the winner is decided and before the prizes are claimed. Such a result cannot be achieved by serializing the business flows of drawing winner and claiming prizes. There are a large body of atomicity violation detection tools for traditional programming languages such as Java and C [12]. They may be adapted to detect violations in smart contracts. However, atomic business flows are usually implicit (suggested by boolean flags serving as locks and explicit time bounds). Such challenges need to be addressed during adaptation. Atomicity violation bugs are usually fixed using lock variables (e.g., `drawingPrase` in our example).

VIII. Related work

There exists previous studies of smart contract bugs. Atzei et al. [13] provide 3 classes of bugs based on where they are introduced (Solidity, Ethereum, Blockchain), as well as 12 types of security vulnerabilities within the three classes. Their taxonomy is from a mid-development perspective, including vulnerabilities such as "*calls to the unknown*" and "*stack size limit*", and hence different from ours. Demolino et al. [14] categorize bugs based on common developer pitfalls. Chen et al. [15] classify bugs into 20 groups, pulling data from posts on the *Ethereum Stack Exchange*, a popular Q/A site for users of Ethereum. Our study differs from the previous studies, as we approach smart contracts that are in the post-development auditing stage as well as those that have already been deployed. Zhang et al. [16] provide a classification of 9 different types of bugs. They study 266 bugs in academic literature and Github from 2014. *SmartDec* [17] provides 3 classes of bugs depending on where they take place: blockchain, model, and language. This is further divided into 33 bug types, pulled from bugs before 2018. Dingman et al. [18] categorize smart contract bugs into 49 master classes from research publications dating from 2014 to 2019. Most these studies provide classification but do not study distributions or difficulty levels. Many focus on bugs that are nowadays machine-auditable. In contrast, we study the latest security bugs and exploits that are not machine-auditable from multiple unique perspectives. Some bugs may be categorized differently by different studies. Our classification aims to achieve a good coverage while enabling abstraction. We consider our study complementary to these existing studies.

IX. Data Availability

Our data is available in the online package [19] whose main contents are origanized as follows.

- `Doc/` includes this supplementary material.
- `Contracts/` includes all the smart contracts we examined as well as their corresponding audit reports (if any).
- `Results/` includes all the raw data we analyzed.

References

[1] "Future contract," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Futures_contract

[2] "Option (finance)," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Option_(finance)

[3] "Stock market index," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Stock_market_index

[4] "List of known - solidity 0.8.16 documentation," 2022. [Online]. Available: https://docs.soliditylang.org/en/v0.8.16/bugs.html

[5] "Foundation exploit." [Online]. Available: https://code4rena.com/reports/2022-02-foundation/#h-01-nft-owner-can-create-multiple-auctions

[6] "Sushi price 2022." [Online]. Available: https://coinmarketcap.com/currencies/sushiswap/

[7] "Lottery — pancakswap." [Online]. Available: https://pancakeswap.finance/lottery

[8] "Pancakeswap lottery vulnerability bugfix review and bug bounty." [Online]. Available: https://medium.com/immunefi/pancakeswap-lottery-vulnerability-postmortem-and-bug-4febdb1d2400

[9] "What is a mempool?" [Online]. Available: https://www.alchemy.com/overviews/what-is-a-mempool

[10] "Frontrunning— ethereum best practices documentation." [Online]. Available: https://consensys.github.io/smart-contract-best-practices/attacks/frontrunning/

[11] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, no. 2008, 2008.

[13] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*. Springer, 2017.

[14] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *International conference on financial cryptography and data security*. Springer, 2016.

[15] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, 2020.

[16] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020.

[17] "Classification of smart contract vulnerabilities." [Online]. Available: https://github.com/smartdec/classification

[18] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng, "Classification of smart contract bugs using the nist bugs framework," in *2019 IEEE 17th International Conference on Software Engineering Research, Management and Applications (SERA)*, 2019.

[19] "Web3-icse/supplementarymaterial." [Online]. Available: https://github.com/Web3-ICSE/SupplementaryMaterial