# A Systematic Study of Recent Smart Contract Security Vulnerabilities

*Abstract*—**We systematically study 502 unique real-world smart contract vulnerabilities/exploits in years 2021-2022. We study how many of them can be exploited by malicious users and cannot be detected by existing analysis tools. We categorize these bugs into seven types and study their root causes, distributions, difficulties to audit (by human auditors), consequences, and repair strategies. For each type, we abstract them to a bug model, facilitating finding similar bugs in other contracts and future automation. We have six main findings. We also leverage these findings to catch 15 critical zero-day vulnerabilities in auditing, all capable of inducing severe monetary loss if exploited. They have been confirmed and fixed by the developers.**

## I. Introduction

Since the Bitcoin and blockchain techonology was introduced in 2008, the market capitalization of Bitcoin has experienced an explosive growth, reaching over $438 billions (as of 5 August 2022) [1]. Nowadays, there exists countless blockchain-based products and services for anyone to interact with, such as those in travel, healthcare, finances, and lately virtual reality. Blockchains such as Ethereum, Solana, and Polygon handle millions of transactions everyday. High-level programming languages like Solidity enable the creation and integration of numerous innovative ideas with blockchains, in the form of *smart contracts*. Just like traditional software applications, smart contracts are composed by developers and hence susceptible to human errors. Many of them are exploitable. According to [2], $1.57 billions were exploited from various smart contracts as of 1 May 2022.

Due to the importance of ensuring smart contract security, there have been a large body of existing techniques, including fuzzers such as [3]–[7], formal verification methods such as [8]–[14], and runtime verification tools [15], [16]. Despite the success of existing techniques, smart contract exploits are still commonly seen in the wild [17]. This may root at the fundamental differences between smart contract and traditional software vulnerabilities.

**Differences between Smart Contract and Traditional Software Vulnerabilities.** For traditional software applications, security vulnerabilities are largely different from functional bugs. The former has limited forms such as buffer overflow (leading to code hijacking) [18], information leak [19], and privilege escalation [20], whereas the latter is very diverse, denoting violations of domain-specific and even application-specific properties. Moreover, functional bugs in traditional software usually lead to incorrect outputs and/or interrupted services, which may not cause direct security concerns. In contrast, smart contract vulnerabilities are in many cases functional bugs, because due to their unique nature, incorrect outputs in smart contract usually indicate monetary loss. Finding these vulnerabilities hence requires checking domain-specific properties, which is much harder than checking a limited set of general security properties in traditional software. □

Therefore, we consider that it is highly valuable to summarize recent smart contract vulnerabilities/exploits to understand the underlying properties critical to security. In this paper, we study a large set of 502 unique exploitable bugs from 158 real-world contracts reported/exploited in the past year or so and aim to summarize their root causes and distributions. We collect these bugs from the highly reputable *Code4rena* contests (with a total of 462 bugs), which invite individuals, companies, and institutes from all over the world to audit real-world contracts by providing substantial bounties [21], and from various real-world exploit reports (e.g., those from [22], [23]), with a total of 40 exploits. The real world exploits account for $249.8 millions monetary loss. Among all the bugs and exploits, we focus on those that can be exploited by malicious users (instead of contract owners) and cannot be detected by existing automated tools. For example, we exclude *re-entrance bugs* [24] because they could have been detected by existing tools such as [25]. In the study, we answer seven research questions such as the root causes of these vulnerabilities, their consequences, repair strategies, and distributions. The detailed setup of our study is in Section III. Compared to existing surveys and studies on smart contract bugs (e.g., [26]–[29]), we collect the latest bugs and study them from many unique perspectives such as distributions and difficulty levels (details in Section XII). Our main findings are highlighted in the following. Details can be found in Section IV.

- (Finding 1) Majority of the 502 security bugs (57%) are in scope, meaning that 43% of them either can be detected by existing automatic scanning/verification tools or cannot be exploited by malicious users.
- (Finding 2) These bugs can be grouped into seven categories based on their root causes: price oracle manipulation, erroneous accounting, ID uniqueness violations, inconsistent state updates, privilege escalation, atomicity violations, and implementation specific bugs. Except the last category (implementation specific bugs) accounting for 15%, the other categories are general meaning that they could appear in many contracts. All of them have severe monetary consequences.
- (Finding 3) Different types of bugs have various levels of difficulties in auditing (by humans). For example, erroneous accounting bugs are the most difficult ones as they require substantial domain knowledge of the underlying business model formulas. Atomicity violations rank the second due to the difficulties of reasoning about

transaction interleavings. ID uniqueness violations are the easiest bugs. Automated tools may offer help as some of hard ones for humans may not be that hard for machines.

- (Finding 4) Price oracle manipulations are the most popular type of real-world exploits (38%), although they only account for 6% of the audit bugs (i.e., bugs found by auditors in Code4rena) and they are not difficult to find (the 3rd easiest). The reason is that most the exploited contracts had not gone through any public auditing. This suggests the importance of auditing.
- (Finding 5) A few categories are instantiations of known bugs in the new context of smart contracts, such as atomicity violations (8.1%) and privilege escalation (10.4%). Existing bug detectors may be adapted to automatically scan them, although there are some domain-specific challenges to address.
- (Finding 6) The repairs for these bugs are mostly easy, requiring a few lines of code changes. They also follow certain patterns, depending on the bug types.

We conduct guided auditing based on these findings and find 15 critical zero-day vulnerabilities, which could endanger $22.52 millions funds if exploited. We are able to find at least one bug for each of the seven categories. Details are in Section XI. Our contributions are summarized as follows.

- We conduct a comprehensive study of a large number recent smart contract security bugs.
- We summarize our findings, which may have ramifications for future work in the area.
- We demonstrate the importance of our findings by our preliminary success in finding zero-day bugs.
- We compile and explain the needed background knowledge to make sure the paper is self-contained, as the explanations of many bugs require substantial domain-knowledge. We provide concrete real-world examples for each bug category.

Discussions of threats to validity are in Appendix XIV-A.

## II. Background

Experienced readers can skip this section.

**Ethereum Blockchain.** The Ethereum Virtual Machine (EVM) or *Ethereum* [30] is an advanced framework for the development of custom financial products and services on the web. This is made possible through the underlying logic of *blockchain* [31], which very abstractly, provides secure information processing and storage by an append-only public ledger that keeps track of transactions. Groups of transactions are collected within a *block*, where transactions can be *mined* by other users known as miners, who use a visible public key and the hash of a transaction to determine whether the transaction is valid. Miners then vote on whether to accept or revert a transaction. This process results in a "consensus view" of all the transactions. Once transactions within a block are finished, the block is appended onto the blockchain, and the internal state of the asset is updated based on transactions that were accepted by miners. Ethereum requires anyone that submits a transaction to provide an appropriate amount of *gas*, which is a fee paid to miners when they process transactions. This provides incentives for miners. Ethereum serves as a platform for developers to create personalized services and products that utilize not only the security of the blockchain but also its decentralized nature. There exists no central authority in the blockchain, as processes are dependent on the consensus of miners, and everything on the blockchain is visible to the public, leading to transactions being transparent and decentralized. As of 29 July 2022, Ethereum has a market capitalization of more than $210 billions [32].

**Smart Contracts.** *Smart contracts* are applications that provide a collection of functionalities to realize some business model. They are usually implemented by specific programming languages such as *Solidity* [33] or *Serpent* [34], leveraging the primitive services provided by Ethereum. Smart contracts provide a wide variety of services such as currency (e.g. NFT tokens), trading markets (e.g. OpenSea), borrowing services (e.g. Nexo), and many more to the developer's design. One of the biggest advantages of smart contracts compared to their physical world counterparts is that they are publicly available for any willing individual to access and no changes to the internal states of a smart contract can be hidden, hence the service *transparency*. Anyone who interacts with a smart contract to consume its services by calling its functions is known as a *user*. Smart contracts are owned by *contract owners*, usually the developers. They have access to special functions in the smart contract that are not *callable* by other users.

A smart contract provides multiple functionalities, each wrapped in some function. It has two kinds of functions, *external* and *internal*. The former can be invoked by a user or the owner and the later can only be invoked by another function within the contract (not by any user). A *transaction* starts when a user invokes some (external) function of a smart contract. A transaction has *atomicity*, meaning changes within a transaction are not visible to the outside world until it is committed/mined. Usually, the transaction ends when it is mined and the root external function call returns. A transaction may fail due to a variety of reasons. When this happens, the transaction is not applied to the blockchain, and any internal state changed by the function is undone. Additionally, all premiums aside from gas fees that were paid through the function are returned. This is known as a *revert*. In general, the execution model of smart contract allows one atomic transaction at a time, meaning that it does not allow another invocation of external function when one is going on. Smart contracts can interact with each other, constituting a *decentralized finance (DeFi)* [35].

**Solidity.** Solidity [33] is one of the most popular programming languages for smart contract. Syntax-wise, it is similar to Java/JavaScript. A *contract* is similar to a class in Java. Solidity supports inheritance as well. Inside a contract, there can be data fields (just like data fields in Java) that may be *public*, *private*, or *internal*. Solidity supports functions which may have modifiers such as *external* indicating a function can be called by a user and *internal*, meaning that only other functions within the contract can call this function. Additionally, Solidity provides a *require* operation that asserts a certain condition. If the assertion fails, an error message is emitted and the current transaction is reverted. Solidity uses *msg.sender* to denote the address of remote user that invokes the current function, and *this* to denote the current contract. Solidity also supports type cast, for example, *address(this)* casts the current contract to its address.

**Address.** On Ethereum and the blockchain, entities such as users and smart contracts are represented by an *address*, or a 20 byte value (e.g., `0xbfDD66a7dE4bB8f494f92A5f8D00443CA6cdaFf6`).

**Tokens and Crypto-currency.** With the introduction of the blockchain, Ethereum, and smart contracts came the need for currency to be developed in order to realize the business models that developers envision. Ethereum resolved this issue with the creation of the *ERC (Ethereum Request for Comment)* tokens. Intuitively, assets are denoted by various kinds of tokens. Tokens can be split into two categories: *fungible* tokens and *non-fungible* tokens (NFTs). ERC20 [36] tokens are fungible, meaning that they are non-unique and inter-changeable. An example would be that denoting a real-world dollar bill. Another typical use scenario of fungible tokens are those denoting ownership of some asset. ERC721 [37] and ERC1155 [38] tokens are non-fungible, meaning that they are unique in the making. For example, houses and paintings in the physical world can be represented by NFTs on Ethereum. Tokens can be *minted* (created), *transferred*, or *burned* (destroyed) from a *central* contract, influencing tokens' values, which depend on the amount of real-world assets stored within the central contract against the amount of tokens in circulation. For example, one could mint 100 fungible tokens to denote the ownership of an asset, namely, each token denotes 1% of ownership. When the asset appreciates, more tokens may be minted to satisfy additional ownership requests. One could also choose to burn some tokens to make the remaining ones more valuable. Users can buy/sell tokens by dealing with their central contracts.

## III. Study Setup

In this section, we explain the detailed setup of our study.

**Data Collection.** We collect and study 502 unique bugs (not bug reports) from two sources: the Code4rena contests and real-world reports. These denote bugs with the criticality level of high/critical. Public smart contract bugs are usually assigned a criticality level: low, medium, or high/critical. We exclude reports at the low and medium levels as these usually cannot lead to monetary loss. Each Code4rena contest lasts for 3-7 days and has a small number of real-world contracts (e.g., 1-3), enlisted by developers who commit a bounty in the range of $20k-1million). Auditors from all over the world, including companies and individuals, can participate and try to find bugs in these contracts. After contest, a group of judges and the developers get together to inspect the bug reports. They will confirm the real ones and pay the bounties accordingly. The reward is decided by the criticality level of bug and the number of reports submitted for the bug (more submissions lead to a lower reward[1]). Most the contracts in the contests are after in-house testing but before deployment. Developers want to leverage the contests to enhance their confidence on the product. A badge from Code4rena also provides a certain level of quality certification. We focus on all the confirmed bug reports that are published after each contest at the Code4rena's official website [39]. Most of them come with *proof of concepts* (PoC). There are 462 of them covering 118 contracts in the past one and a half years.

The second source is reports of real-world exploits on Twitter in 2022, which are published by highly-reputable security researchers (e.g., samczsun [22]) and companies (e.g., [23], [40], [41]). For each exploit, we collect its report, the on-chain contract, as well as the on-chain exploit transaction(s) . Overall, we collect 40 real-world exploits in 2022, all of which have caused a tremendous amount of direct monetary loss.

**Threat Model and Scope of Our Study.** In our threat model, the adversary is a contract user and he crafts special inputs to exploit the contract. Other attacks such as insider attacks and spam attacks are out of scope. Insider attacks are launched by contract owners (who might steal funds leveraging their owner privileges). In spam attacks, the adversary only setups a trap and the user has to be lured to take actions leading to undesirable consequences. We call bugs that do not align with our threat model the *beyond threat model bugs*.

We also exclude bugs that (we believe) can be detected by existing automatic tools, such as [3], [5], to the best of our knowledge. We call them the *machine-auditable bugs*. They include the following: reentrance bugs [24], rounding and overflow bugs [42], uninitialized variables [43], arbitrary external function call without control [44], and gas computation bugs [45].

**Study Procedure and Research Questions.** We conduct the study as follows. For each case, we inspect the bug report, the faulty contract (which is available through Github or on the blockchain) and its documentation. For those that did not have clear explanations, we further run their PoCs (around 1/5 of the total cases). To mitigate human mistakes, we ensure that each bug report has been checked by at least two authors. We answer the following questions when inspecting each case.

- **(RQ1)** Is the bug in scope?
- **(RQ2)** What are the root cause and consequences of the bug?
- **(RQ3)** How difficult is it to find the bug in auditing?
- **(RQ4)** What are the symptoms of the bug? That is, what kind of code features might indicate its presence?
- **(RQ5)** How is the bug fixed?

In addition, we answer two more research questions by aggregating these case studies.

[1]In some extreme situations, a unique medium level bug may get a bounty of tens of thousands of US dollars, whereas a critical bug found by many is only rewarded with ten dollars (per report).

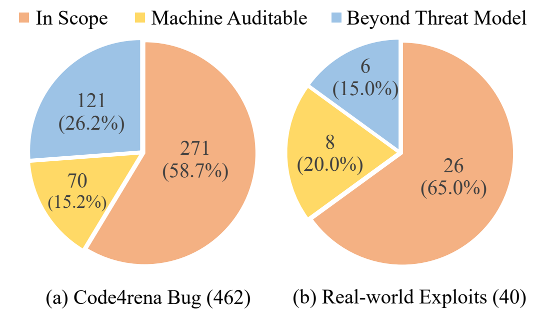


Fig. 1: Break down of bug reports

TABLE I: Auditing difficulty

| Type | # Reports | Avg. Difficulty ↓ |
|---|---|---|
| In scope bugs | 271 | 2.43673469 |
| Machine auditable | 70 | 2.50819672 |
| Beyond threat model | 121 | 2.62626263 |

- **(RQ6)** What are the distribution of these bugs?
- **(RQ7)** Are there distribution differences between auditing bugs (pre-deployment) and real-world exploits (post-deployment)? What are the reasons for such differences?

It is in general very difficult to answer RQ3 because it is subjective. However, the Code4rena contests provide a perfect platform to quantify bug difficulty. Specifically, each contest is participated by a large number of independent auditors, who submit their reports separately. The judges and the developers classify the bug reports based on the root causes. Although there are skill level variations of the auditors, the number of submitted reports for a bug suggests the *relative* difficulty level in finding this bug.

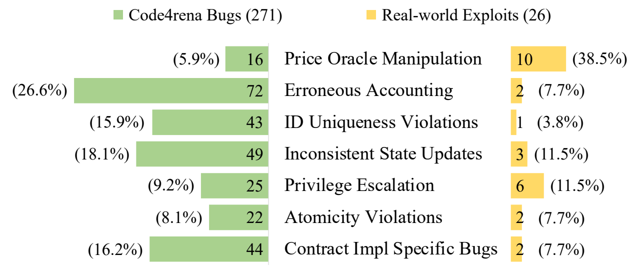
## IV. Study Results

In this section, we present our study results, which are organized by the research questions.

**(RQ1) In-scope Bugs.** Fig. 1 presents the break-down of the 502 bugs that we study. The left is for the Code4rena bugs and the right is the real-world exploits. Observe that 26% of the Code4rena bugs are beyond our threat model (e.g., not exploitable by malicious users) and 15% (we believe) can be identified by existing automated tools. The remaining 58.7% bugs require substantial manual efforts in auditing. The distribution of real-world exploits is similar. This suggests the importance of our study, which hopefully can shed light on further automating some of these problems.

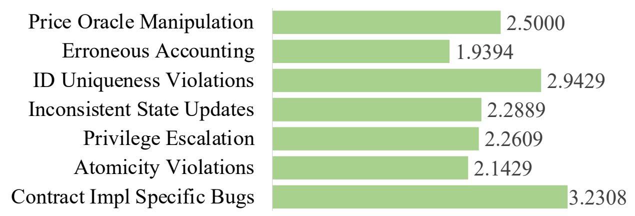
Table I shows the difficulty in auditing these bugs. The last column presents the average number of reports for each bug. Therefore, a smaller number indicates harder-to-find bugs (denoted by the downward arrow besides the column title). Observe that the bugs that we study are the most difficult to find. It is unclear if the auditors indeed have used any automated tools to find the machine auditable bugs.

**(RQ2, RQ3, RQ6, and RQ7) Root Causes and Distributions.** The 271+26 in-scope bugs can be grouped into the following 7 categories: (C1) price oracle manipulation; (C2) erroneous accounting; (C3) ID uniqueness violations; (C4) inconsistent state updates; (C5) privilege escalation; (C6) atomicity violations; and (C7) implementation specific bugs. Their distributions can be found in Fig. 2 and their difficulty levels can be found in Fig. 3.



| Code4rena % | Code4rena Bugs (271) | Type | Real-world Exploits (26) | Real-world % |
|---|---|---|---|---|
| (5.9%) | 16 | Price Oracle Manipulation | 10 | (38.5%) |
| (26.6%) | 72 | Erroneous Accounting | 2 | (7.7%) |
| (15.9%) | 43 | ID Uniqueness Violations | 1 | (3.8%) |
| (18.1%) | 49 | Inconsistent State Updates | 3 | (11.5%) |
| (9.2%) | 25 | Privilege Escalation | 6 | (11.5%) |
| (8.1%) | 22 | Atomicity Violations | 2 | (7.7%) |
| (16.2%) | 44 | Contract Impl Specific Bugs | 2 | (7.7%) |

Fig. 2: Breakdown of different types of bugs in scope



| Type | Difficulty |
|---|---|
| Price Oracle Manipulation | 2.5000 |
| Erroneous Accounting | 1.9394 |
| ID Uniqueness Violations | 2.9429 |
| Inconsistent State Updates | 2.2889 |
| Privilege Escalation | 2.2609 |
| Atomicity Violations | 2.1429 |
| Contract Impl Specific Bugs | 3.2308 |

Fig. 3: Difficulty levels of auditing different types of bugs

*(C1) Price Oracle Manipulation.* Smart contracts usually resort to external authorities on Ethereum, which usually are also contracts called *price oracles*, to determine the price of an asset. Oracles use certain rules to determine prices (e.g., based on reserve balances). However, if an application contract does not use a price oracle's APIs properly, the adversary can interact with the price oracle in a legit way to influence the price query result returned to the application contract to gain undeserved profits. More detailed explanation and an example can be found in Section V. It is one of the most notorious types of vulnerabilities in the DeFi history, causing at least $44.8 millions loss in the first half of 2022 alone. As shown in Fig. 2, it constitutes 6% of the Code4rena bugs (the least common bug) and 38% of the real-world exploits (the most common exploit). The difficulty level of auditing such bugs, as shown in Fig. 3, ranks the 5th out of the 7 types. There is an interesting discrepancy of their popularity in auditing and in real-world exploits, even though these bugs are not that difficult to find. Further inspection shows that most exploited contracts had not gone through any public audit. They were hence easy and highly profitable targets.

*(C2) Erroneous Accounting.* Many smart contracts implement complex domain-specific financial models. The implementations hence involve a lot of difficult-to-interpret numerical computation. We call incorrect implementations of underlying financial model formulas *erroneous accounting* bugs (e.g., using a wrong coefficient or a wrong formula). These bugs usually introduce small errors every time they are exercised. However, these errors can be accumulated to induce substantial loss. For example, the *Compound Finance* [46], a flagship lending contract supporting time-sensitive variable interest rates, was exploited and had over $80 millions stolen, due to an unnoticeable problematic calculation of annual percentage yield [47]. The bug survived 9 rounds of auditing by top

security companies [48] and even formal verification by a state-of-the-art commercial tool [49]. It was not found until being exploited. Fig. 2 shows that it is the most popular type of Code4rena bugs (27%) and the 4th most popular type of real-world exploits. Fig. 3 shows that it is also the most difficult type of bugs. The reason is that finding such bugs requires substantial domain knowledge. The very broad participation of the Code4rena contests seems to provide a good coverage of domain expertise such that a lot of these bugs can be captured (although each only by very few auditors). More details can be found in Section VI.

*(C3) ID Uniqueness Violations.* Most smart contract functionalities are in the form of some *entity* (e.g., a user or contract) operating on some *asset* (e.g., an NFT token). As such, access control is critical in these processes. For example, many gambling contracts have functions that blacklist users from participating, similar to how certain people can be barred from gambling facilities in the physical world. Accesses are essentially relations between entities and assets, stating the permission an entity has on an asset. It is hence critical to uniquely represent entities and assets. Within smart contract implementation, entities and assets are usually denoted as data structures, which usually have an ID field that *uniquely* represents an entity/asset. However, developers may forget to ensure uniqueness of ID fields; they may mistakenly consider other data fields are unique and use them as as replacement IDs. As such, the adversary could impersonate an entity or create a fake/duplicate asset that has the same field value as some real entity/asset to pass the access control checks and then perform illegal operations. We call this type of bug *ID uniqueness violations*. These vulnerabilities could lead to direct fund loss of the contracts and/or victim users. One of the most recent exploits of this category caused 100+ NFTs or around \$1.4 millions to be stolen from a contract [50]. It constitutes 16% of the Code4rena bugs (43 out of 271) and 4% of real-world exploits (1 out of 29). It is the 3rd and the 7th most commonly seen type of bugs in the two respective data sets. The usual difficulty level of finding such bugs is relatively low, with an average of around 2.9 auditors per bug in the audit stage. This could explain the difference in distribution between contracts in the auditing stage (Code4rena) compared to those deployed onto the blockchain (real-world).

*(C4) Inconsistent State Updates.* Smart contracts have many state variables (e.g., debts and collaterals). There are implicit correlations between these state variables, such as the credit limit of a user is proportional to her collateral in a lending contract. However, when the developers update one variable, they may forget to update the correlated variable(s) or update incorrectly. An example can be found in Section VIII. Depending on the state variables that are incorrectly updated, the consequences of this kind of bugs range from incorrect statistics to loss of funds. In the recent year, three exploits [51] [52] [53] caused around \$3.8 millions loss and also the complete collapse of one smart contract's internal economy. It constitutes 18% of the Code4rena bugs (49 out of 271) and 10% of the real-world exploits. It is the 3rd most commonly seen type of bugs in both data-sets. The difficulty level of finding such bugs is slightly higher than average, with around 2.3 auditors per bug during the Code4rena auditing process.

*(C5) Privilege Escalation.* Smart contracts often support a number of business flows, each denoting a unique use case. For example, a lottery contract needs to support at least three distinct flows including buying tickets, drawing winners, and claiming prizes. A business flow may consist of a sequence of transactions in the temporal order. Within a flow, sensitive operations are guarded by access control checks. However, there may be some unexpected business flow to a sensitive operation along which the access control is weaker than necessary. This is very similar to *privilege escalation* bugs that are very popular in mobile applications [20]. These bugs have diverse consequences, depending on the sensitive operations that are not well protected. Nearly \$7.5 millions got stolen in 2022, due to privilege escalation bugs. It constitutes 9.2% of the Code4rena bugs and 11.5% of the real-world exploits. It is the second most popular type of real-world exploits. The difficulty of auditing them is about average.

*(C6) Atomicity Violations.* Multiple business flows (i.e., transaction sequences) may interleave and interfere with each other, by accessing the same state variables. Some business flows may require business level atomicity, demanding state variables cannot be accessed by other flows while they are ongoing. Developers do not anticipate such interference and fail to ensure (business level) atomicity. The reason of these bugs is that developers mistakenly think atomicity is guaranteed by the EVM runtime and hence they do not need to be concerned. However, EVM only ensures each transaction is atomic, and business flow atomicity, if needed, has to be ensured by the developers. We provide an example in Section X. Atomicity violations constitute 8.1% of the Code4rena bugs and 7.7% of real-world exploits. It is the least common bugs in auditing, and the second least in the wild. They are much harder to find (than the others), ranked the 2nd in terms of audit difficulty (reported 2.1 times on average for each bug). The reason is that it is difficult to determine business flows and if they need atomicity. Reasoning about interleavings is also hard for humans, although it may not be as hard for tools.

*(C7) Contract Implementation Specific Bugs.* We also find that 16% of the Code4rena bugs and 7.7% of the real-world exploits are implementation specific, meaning that they unlikely appear in a different smart contract implementation. For example, implementation is inconsistent with documentation. We consider these bugs have a low priority because abstracting these bugs may not provide as valuable guidance as the others. An example of such bugs is presented in Appendix XIV-D.

**(RQ4 and RQ5) Symptoms and Repairs.** We will present our study results of RQ4 and RQ5 in the following sections when explaining individual attack types, because the concrete examples in those sections will make such discussions easier. For each bug type, we will summarize its symptoms and procedure to an *abstract model* and also explain its fixes.

## V. Price Oracle Manipulation

These bugs require additional knowledge. We first introduce the concepts and then explain such bugs with an example.

**Price Oracle and Automated Market Maker.** Determining the price of an asset is a critical functionality for a business model. In DeFi, it is done by *price oracles*. Despite a diverse set of price oracle contracts, the predominant sort is *Automate Market Maker* (AMM), which is designed for exchanging two types of assets, e.g., WETH and USDC (similar to USD in real-world), with which users can exchange one asset for another and the exchange rate is decided by a pre-defined invariant law. In Uniswap [54], a leading AMM contract, the invariant is denoted by a constant product formula, expressed as $x \times y = k$, stating that trades must not change the product $k$ of a pair's reserve balances (within the contract), e.g., $x$ for WETH and $y$ for USDC. Driven by the *supply and demand* theory [55], the AMM's reverse balances of the two assets would reach a dynamic balance, implicitly determining the price of one asset over the other by their ratio, e.g., $y/x$ denoting the price of WETH over USDC. Intuitively, more supply of $x$ leads to its depreciation and $y$'s appreciation. A code snippet from Uniswap and its explanation are presented in Appendix XIV-B.

*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. □

**Price Oracle Manipulation.** Despite being pivotal for DeFi project development, price oracles are occasionally used improperly by application smart contracts, rendering their price queries vulnerable to malicious manipulation. It is not a bug in the price oracle contract, but an issue caused by oracle misuse in the application contract. For example, although Uniswap provides an official (and well protected) API for price queries, application contract developers tend to implement their own queries (to Uniswap) due to the complexity of the official one. Furthermore, the customized queries are usually not exploitable in the traditional finance, but vulnerable in DeFi, rendering the problem more insidious. A common faulty code pattern in the application contract is to simply determine the price by querying the ratio of two assets' instant balances in the oracle contract, e.g., `balance0/balance1` in lines 13 and 14 in Fig 9 (in Appendix). Note that the correctness of such a query is guaranteed in the traditional finance, due to the massive market in which any subtle deviation from the real-world price is immediately eliminated by arbitrage actions. However, the situation drastically changes in DeFi. Recall that block-chain transactions are atomic, so that any action sequence in a single transaction (i.e., a function invocation) cannot be interrupted or interleaved with other actions. Hence, a malicious user can tamper with the price without the interruptions of arbitragers. It is done by first processing an exchange (with the oracle), then invoking a function in the vulnerable application contract which makes an erroneous query (to the oracle), and finally processing another exchange (with the oracle) which is the counter version of the first one. Essentially, the first exchange imbalances the Uniswap contract in order to manipulate the follow-up price query, while the second exchange re-balances the Uniswap contract to avoid losing the (borrowed) funds used in step one. Note that the three actions are wrapped in a single transaction (a piece of code written by the adversary), guaranteeing that no arbitrage behavior can interfere the attack.

```
1  contract LendingContract {
2    IERC20 public WETH;
3    IERC20 public USDC;
4    IUniswapV2Pair public pair; // USDC - WETH
5    // debt --> USDC, collateral --> WETH
6    mapping(address => uint) public debt;
7    mapping(address => uint) public collateral;
8
9    function liquidate(address user) external {
10     uint dAmount = debt[user];
11     uint cAmount = collateral[user];
12     require(getPrice() * cAmount * 800 / 100 < dAmount,
13       "the given user's fund cannot be liquidated");
14     address _this = address(this);
15     USDC.transferFrom(msg.sender, _this, dAmount);
16     WETH.transferFrom(_this, msg.sender, cAmount);
17   }
18   function getPrice() view returns (uint) {
19     return (USDC.balanceOf(address(pair)) /
20         WETH.balanceOf(address(pair)))
21   }
22 }
```

Fig. 4: Price Oracle manipulation exploit in Deus Finance

*Example.* Fig. 4 presents a vulnerable code snippet suffering from price oracle manipulation. The code snippet is slightly modified from a real-world exploit against the Deus Finance which led to a loss of \$3.1 millions. The bug survived at least one publicly-known audit round [56]. Deus is a lending contract that allows users to deposit WETH as collateral and borrow USDC. Lines 2-4 define the addresses of WETH, USDC, and the Uniswap AMM, respectively. Line 6 defines a mapping `debt`, which denotes the amount of borrowed USDC for each user, and line 7 a mapping `collateral` for the amount of each user's deposited WETH. As a lending contract, Deus supports multiple basic functionalities, including depositing collateral, withdrawing collateral, getting loans, and paying debts. The vulnerability lies in function `liquidate` (line 9) which forces to close a given user's *ill position*, i.e., the user's debts exceeds $80\%$ of her collateral. To do so, the function's caller, i.e., `msg.sender`, pays the user's debt and gets her collateral. Specifically, the function first checks whether the position of `user` is ill (lines 10-13) and processes the token transfers (lines 14-16). The price oracle is involved when calculating the real-world value of the collateral, i.e., WETH, through function `getPrice()` (defined in lines 18-21). The function does not use Uniswap API. Instead, it directly queries the instance balances of USDC and WETH in Uniswap and uses their ratio as the price. This may consume less gas compared to using the official API.

To exploit, the adversary drastically decreases the price of a collateral, forcefully making a victim's position liquidable. She then liquidates a valuable collateral with a much smaller amount of fund. Assume Bob (victim) deposits 100 WETH as collateral and borrows $100,000$ USDC. Also assume that the current price of WETH is $\$4,000$ and the Uniswap pair holds 100 WETH and $400,000$ USDC. Note that Bob's current position is healthy and cannot be liquidated, since the value of his debt is $\$100,000$ and his collateral worths $\$400,000$. Alice, the adversary, can exploit the aforementioned vulnerability by encapsulating the following three actions into a single transaction. Specifically, she first exchanges 100 WETH for $200,000$ USDC through UniSwap, making the AMM's balances of WETH and USDC 200 and $200,000$, respectively. Note that although the current real-world price of WETH is $\$4,000$, Alice pays 100 WETH for $200,000$ USDC, according to the constant-product invariant, i.e., $100 \times 400,000 = (100+100) \times (400,000 - 200,000)$. Alice then invokes `liquidate(Bob)`, which succeeds since Bob's position depreciates with a WETH price of $\$1000$ (due to the instant balances of WETH and USDC in the AMM), i.e., $100 \times 1000 \times 0.8 < 100,000$ at line 12. By paying $100,000$ USDC, Alice gets 100 WETH whose real-world value is $\$400,000$. She acquires a large profit of $\$300,000$. After that, Alice re-balances the AMM by exchanging $200,000$ USDC for 100 WETH, retrieving her initial attack funds. The bug was fixed by using the Uniswap official oracle API. □

There are other manifestations of such bugs. For example, when developers use an official API to return an average price within a time window, namely *Time-Weighted Average Price* (TWAP), they may query with a very short window.

**Flash Loans.** Recall that the aforementioned exploit requires a tremendous amount of initial funds, i.e., 100 WETH with $\$400,000$ real-world value, which seems to hinder the impact of price oracle manipulation. However, *flash loan*, a unique and innovative lending model enabled by blockchain techniques, makes such attacks easily realizable. It allows users to borrow (a tremendous amount of) debts without depositing any collateral. It leverages the atomicity of blockchain transactions, that is, the borrow happens at the beginning of a transaction and the debt is payed off at the end. An example can be found in Appendix XIV-C.

**Abstract Bug Model (RQ4) and Remedy (RQ5).** Given

```
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. 5: The LFW ecosystem exploit

a price oracle $C_{orc}$, an application contract $C$, and lending contract(s) $C_l$ supporting flash loans, $C$ needs to query $C_{orc}$ for prices which are based on instant balances (or balances within a short time) in $C_{orc}$, and $C_l$ needs to have sufficient funds to manipulate the balance ratio in $C_{orc}$. The cost of the attack is minimum, including just gas and fees, as the flash loan is paid off at the end. The profit depends on how much price changes can be induced. To remedy such bugs, developers simply use official APIs strictly following the specification (e.g., TWAP with a proper time window), for both on-chain [54] and off-chain [57] price oracles.

## VI. Erroneous Accounting

As defined in Section IV, 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.* Fig. 5 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 (Section V). 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 $0.022\%$, 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., `balance0` $\times 10 -$ `amount0In` $\times 0.022 \geq$ `reserve0` $\times$ `reserve1` (reduced

```
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. 6: The NFTMarketReserveAuction exploit

from $\texttt{balance0} \times 10000 - \texttt{amount0In} \times 22 \geq \texttt{reserve0} \times \texttt{reserve1} \times 1000$). The adversary pays only one tenth of the expected `token0` to get `token1` he demands. □

**Abstract Bug Model (RQ4) and Remedy (RQ5).** A contract $C$ is supposed to implement a mathematical model $\mathcal{M}$. However, the implementation is inconsistent with $\mathcal{M}$ (e.g., using the wrong coefficients or expressions). The auditor/tool needs to have the domain-knowledge of the precise form of $\mathcal{M}$, which is usually not available (in documentation). This makes finding these bugs difficult (Fig. 3). 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 (Fig. 2)), due to the very broad domain expertise brought by the participants. The fixes are usually simple, including changing coefficients and rephrasing arithmetic expressions.

## VII. 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 [58]. 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 Fig. 6, 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.

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 (RQ4) and Remedy (RQ5).** 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.

## VIII. 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.* Fig. 7 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 [59]. 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

```
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. 7: The SushiTrident exploit

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 (RQ4) and Remedy (RQ5).** 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.

## IX. Privilege Escalation

These bugs arise when an (unexpected) sequence of functions can be invoked to bypass access control.

*Example.* Fig. 8 presents a real-world case from an anonymized contract (upon developers' request). The code is completely rewritten to retain anonymity while its essence is retained. This is a voting contract where users can elect a new contract owner by voting. In lines 2-4, the contract defines a data structure `Proposal` to describe a proposal with `sTime` denoting the start time of voting and `newOwner` the proposed new owner. There are three state variables `votingToken`, `owner`, and `proposal` denoting the token used for voting (line 5), the current contract owner (line 6), and an on-going proposal (line 7), respectively. Function `propose` (line 9) allows a user to propose himself as the new owner, which

```
1  contract Vote {
2    struct Proposal {
3      uint160 sTime; address newOwner;
4    }
5    IERC20 votingToken;
6    address owner;
7    Proposal proposal;
8
9    function propose() external {
10     require(proposal.sTime == 0, "on-going proposal");
11     proposal = Proposal(block.timestamp, msg.sender);
12   }
13   function vote(uint amount) external {
14     require(proposal.sTime + 2 days > block.timestamp,
15       "voting has ended");
16     votingToken.transferFrom(
17       msg.sender, address(this), amount);
18   }
19   function end() external {
20     require(proposal.sTime != 0, "no proposal");
21     require(proposal.sTime + 2 days < block.timestamp,
22       "voting has not ended");
23     require(votingToken.balanceOf(address(this))*2 >
24         votingToken.totalSupply(), "vote failed");
25     owner = proposal.newOwner;
26     delete proposal;
27   }
28   function getLockedFunds() external onlyOwner { ... }
29 }
```

Fig. 8: A voting contract vulnerability

creates a new proposal (at line 11) and sets the current block time as the start time and `msg.sender` the proposed owner. Observe that there can only be one on-going proposal (line 10). Users vote by function `vote`, in which they send their voting tokens to the contract (lines 16-17) to support a proposal. Note that users can only vote in the first two days after the voting starts, guarded by the `require` in lines 14-15. The voting ends two days later, and the decision is made by function `end`. Function `end` first checks whether there is an on-going proposal (line 20) and whether the voting has lasted for at least 2 days (lines 21-22). In lines 23-24, the function then checks whether over 50% `votingToken` holders have voted for the proposal. If so, a critical operation of setting a new contract owner is performed (line 25). At line 28, a privileged function `getLockedFunds` allows the owner to get all the locked funds, including the `votingToken` used for previous votes. Note that both functions `vote` and `end` strictly constrain the invocation time (i.e., within and beyond the 2-day time slot, respectively), which constitutes an access control preventing the two functions from being invoked at the same time (i.e., in a single transaction). Otherwise, an adversary could invoke function `vote` with a tremendous amount of flash-loaned `votingToken` and force a malicious proposal to go through (similar to the exploit in Section V). Recall that flash loans require the multiple attack steps being packed into a single transaction. The checks on invocation time hence defend such flash-loan attacks. However, an unexpected call sequence can evade the access control. Specifically, consider an adversary proposes herself as the owner. When the time is approaching the deadline `proposal.sTime + 2 days`, she launches a flash-loan attack wrapping the following actions into a single transaction, including 1) flash-loaning a large amount of `votingToken` from its AMM contract, 2) invoking `votingToken.transferFrom`, a fund transfer interface function provided by all ERC20 tokens to directly transfer the loaned amount to the contract without any access control (as a token usually has no idea of the high-level business logic), 3) invoking `end` to become the owner, 4) getting locked funds by function `getLockedFunds`, and 5) paying off the flash-loan debt. The developers did not anticipate such a business flow and hence did not guard properly. □

**Abstract Bug Model (RQ4) and Remedy (RQ5).** Let a business flow $\mathcal{B}$ be a sequence of transactions $t_1$, ..., $t_n$, each denoting an external function invocation, and $n$ the length of flow which may be equal to or larger than 1. Assume $\mathcal{B}$ has some critical operation $f$ guarded by a set of access control checks, denoted as $\mathcal{P}$, a conjunction of multiple checks. However, there exists an (unexpected) business flow $t'_1$, ... $t'_m$ that can reach $f$ with access control $\mathcal{P}'$ and $\mathcal{P}' < \mathcal{P}$ (here the operator $<$ means weaker-than). The challenges of identifying this type of bugs lie in recognizing sensitive operations, which may require domain knowledge, and finding the multiple paths that can lead to the operations. Program analysis tools are likely helpful. The fixes are to add the missing access control checks or prevent the unexpected paths.

## X. Atomicity Violations

Due to space limitations, we move it to Appendix XIV-E.

## XI. Guided Auditing

Inspired by our findings, we started to audit real-world contracts using our study as a guidance since April 2022. By the time of writing, we have found 15 confirmed zero-days with a few more under the inspection of judges. All the confirmed ones are ranked *critical*, meaning that they could induce substantial monetary loss if exploited. Specifically, we reported 4 zero-days through Immunefi [60], a public bounty platform. We also participated in three Code4rena contests and ranked #1 in one of them, out of the $\sim$100 teams/individuals that had submitted at least one valid report. The other two contest results are still in the hands of judges by the time of submission. Our aggregated bounty is $\$102,659.98$ so far and the total funds protected due to our reports add up to $\$22.52$ millions (based on the market cap of each project by the time of submission). More importantly, we have strategized based on our findings. For example, we have focused on finding price oracle manipulations (POM) and privilege escalations (PE), the two most popular kinds of bugs according to our study. The results are also quite rewarding. We were granted over $\$65$k bounty for the two POM bugs we found, and managed to find 4 PE bugs. The abstract bug models are quite helpful too. For example, when we were looking for PE bugs, we first identified a critical operation $f$ (see Section IX) and then listed their enclosing business flows explicit from the code, leveraging documentation and code hints such as time windows and locks. We then exhaustively enumerate other (usually implicit) operation paths reaching the same $f$ and check their access control.

Table II summarizes our auditing results. The first column shows the bug types, the last two columns the number of bugs we report through the bounty programs and the Code4rena contests, respectively. The last two rows report the total bounty received and the total funds protected, respectively. Observe that we are able to find at least one bug for each category, supporting the coverage of our categorization and the effectiveness of our abstract bug models.

TABLE II: Guided auditing results

| Type | Bounty Program (4) | Code4rena (11) |
|---|---|---|
| Price Oracle Manipulation | 2 | 0 |
| Erroneous Accounting | 0 | 2 |
| ID Uniqueness Violations | 0 | 1 |
| Inconsistent State Updates | 0 | 1 |
| Privilege Escalation | 1 | 3 |
| Atomicity Violations | 0 | 2 |
| Contract Impl Specific Bugs | 1 | 2 |
| Total Bug Bounty Awarded | | ***102,659.98 USD*** |
| Total Funds Protected | | ***22.52 million USD*** |

## XII. Related Work

There exists previous studies of smart contract bugs. Atzei et al. [26] 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. [61] categorize bugs based on common developer pitfalls. Chen et al. [29] 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. [28] provide a classification of 9 different types of bugs. They study 266 bugs in academic literature and Github from 2014. *SmartDec* [62] 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. [27] 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.

## XIII. Conclusion

We study 502 smart contract security bugs and exploits. We categorize them by root causes and study their distributions, repair strategies, and audit difficulty levels. We have six findings. We also perform guided auditting based on these findings and have found 15 critical zero-days in three months that could endanger $\$22.52$ millions funds if exploited.

## References

[1] “Bitcoin market cap.” [Online]. Available: https://coinmarketcap.com/

[2] “The growing rate of defi fund loss.” [Online]. Available: https://twitter.com/PeckShieldAlert/status/1520620826613010432

[3] V. Wüstholz and M. Christakis, “Harvey: A greybox fuzzer for smart contracts,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[4] “crytic/echidna.” [Online]. Available: https://github.com/crytic/echidna

[5] “foundry-rs/foundry.” [Online]. Available: https://github.com/foundry-rs/foundry

[6] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, “Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[7] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, “Learning to fuzz from symbolic execution with application to smart contracts,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[8] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[9] N. He, R. Zhang, L. Wu, H. Wang, X. Luo, Y. Guo, T. Yu, and X. Jiang, “Security analysis of eosio smart contracts,” *arXiv preprint arXiv:2003.06568*, 2020.

[10] J. Frank, C. Aschermann, and T. Holz, “{ETHBMC}: A bounded model checker for smart contracts,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2757–2774.

[11] Z. Nehai, P.-Y. Piriou, and F. Daumas, “Model-checking of smart contracts,” in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 980–987.

[12] M. Bartoletti and R. Zunino, “Verifying liquidity of bitcoin contracts,” in *International Conference on Principles of Security and Trust*. Springer, 2019, pp. 222–247.

[13] N. Atzei, M. Bartoletti, S. Lande, N. Yoshida, and R. Zunino, “Developing secure bitcoin contracts with bitml,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1124–1128.

[14] J. Feist, G. Grieco, and A. Groce, “Slither: a static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[15] “Chaos labs.” [Online]. Available: https://chaoslabs.xyz/

[16] “Tenderly — ethereum development platform.” [Online]. Available: https://tenderly.co/

[17] “The nine largest crypto hacks in 2022.” [Online]. Available: https://blockworks.co/the-nine-largest-crypto-hacks-in-2022/

[18] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities,” in *10th USENIX Security Symposium (USENIX Security 01)*, 2001.

[19] F. J. Serna, “The info leak era on software exploitation,” *Black Hat USA*, 2012.

[20] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in *international conference on Information security*. Springer, 2010, pp. 346–360.

[21] “Leaderboard - code4rena.” [Online]. Available: https://code4rena.com/leaderboard

[22] “samczsun twitter.” [Online]. Available: https://twitter.com/samczsun

[23] “Peckshield twitter.” [Online]. Available: https://twitter.com/peckshield

[24] “Reentrancy - ethereum smart contract best practices.” [Online]. Available: https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/

[25] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks,” *arXiv preprint arXiv:1812.05934*, 2018.

[26] 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, pp. 164–186.

[27] 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)*. IEEE, 2019, pp. 116–123.

[28] 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, pp. 139–150.

[29] 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.

[30] V. Buterin *et al.*, “A next-generation smart contract and decentralized application platform,” *white paper*, vol. 3, no. 37, pp. 2–1, 2014.

[31] L. Anderson, R. Holz, A. Ponomarev, P. Rimba, and I. Weber, “New kids on the block: an analysis of modern blockchains,” *arXiv preprint arXiv:1606.06530*, 2016.

[32] “Ethereum market capital 2022.” [Online]. Available: https://coinmarketcap.com/currencies/ethereum/

[33] “Solidity documentation.” [Online]. Available: https://docs.soliditylang.org/en/v0.8.15/

[34] “Serpent documentation.” [Online]. Available: https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm

[35] L. Zhang, X. Ma, and Y. Liu, “Sok: Blockchain decentralization,” *arXiv preprint arXiv:2205.04256*, 2022.

[36] “Erc20 token standard.” [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[37] “Erc721 non-fungible token standard.” [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[38] “Erc1155 multi-token standard.” [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155

[39] “Code4rena contest pocs.” [Online]. Available: https://code4rena.com/reports

[40] “Paradigm twitter.” [Online]. Available: https://twitter.com/paradigm

[41] “Certik twitter.” [Online]. Available: https://twitter.com/CertiK

[42] E. Lai and W. Luo, “Static analysis of integer overflow of smart contracts in ethereum,” in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, 2020, pp. 110–115.

[43] “A complete breakdown of uninitialized storage parameters.” [Online]. Available: https://blog.immunebytes.com/a-complete-breakdown-of-uninitialized-storage-parameters/

[44] “External calls — ethereum best practices documentation.” [Online]. Available: https://consensys.github.io/smart-contract-best-practices/development-recommendations/general/external-calls/

[45] N. F. Samreen and M. H. Alalfi, “Smartscan: an approach to detect denial of service vulnerability in ethereum smart contracts,” in *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2021, pp. 17–26.

[46] “Compound finance website.” [Online]. Available: https://compound.finance/

[47] “Defi money market compound overpays millions in comp rewards in possible exploit; founder says $80m at risk.” [Online]. Available: https://www.coindesk.com/tech/2021/09/30/defi-money-market-compound-overpays-15m-in-comp-rewards-in-possible-exploit/

[48] “Compound — doc — audits.” [Online]. Available: https://compound.finance/docs/security#audits

[49] “Compound — doc — formal verification.” [Online]. Available: https://compound.finance/docs/security#formal-verification

[50] “Treasury dao exploit.” [Online]. Available: https://news.bitcoin.com/attacker-hacks-arbitrums-treasure-dao-for-over-100-nfts-by-leveraging-marketplace-exploit/

[51] “Wiener doge exploit.” [Online]. Available: https://www.certik.com/resources/blog/Br4j8oVnz9zKqW3okCyD9-wiener-doge-exploit

[52] “Carnival lab exploit.” [Online]. Available: https://watcher.guru/news/did-this-hacker-get-away-with-a-3-8-million-nft-hack

[53] “Pancakeswap exploit.” [Online]. Available: https://www.bsc.news/post/pancakeswap-emergency-brake-on-syrup-pools

[54] “Home — uniswap protocol.” [Online]. Available: https://uniswap.org/

[55] R. F. Muth, “The derived demand curve for a productive factor and the industry supply curve,” *Oxford Economic Papers*, vol. 16, no. 2, pp. 221–234, 1964.

[56] “Deus.finance - smart contract audit report.” [Online]. Available: https://solidity.finance/audits/DEUS/

[57] "Blockchain oracles for hybrid smart contracts." [Online]. Available: https://chain.link/

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

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

[60] "Immunefi." [Online]. Available: https://immunefi.com/explore/

[61] 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, pp. 79–94.

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

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

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

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

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

[67] 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, pp. 329–339.

[68] 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.

```
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. 9: The swap function of Uniswap

## XIV. Appendix

### A. Threats to Validity

The *internal* threat to validity mainly lies in the human mistakes in the study. Specifically, we may misclassify a bug and miss a category. To reduce this threat, we ensure each bug has been examined by at least two authors. The categorization is agreed on by all the authors. Most authors have extensive smart contract auditing experience and cyber-security expertise in general. The *external* threat to validity mainly lies in the subjects used in our study. The bugs we study may not be representative. We mitigate the risk using highly reputable data sources and a large number of bugs. Since we focus on recent bug reports, the study may not represent historic bugs well. However, we argue that studying up-to-date bugs is of importance due to the fast evolution pace of the field.

### B. Price Oracle Example.

Fig 9 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.

### C. Flash Loan Example

Uniswap inherently supports flash loans. Specifically, in Fig 9, 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.

### D. Contract Implementation Specific Bug Example

Fig. 10 is a real-world example of an implementation bug from an anonymous contract. The code is completely rewritten to retain anonymity and simplified for illustration while retaining its essence. This is an NFT market contract, where users can buy and sell NFTs. Users can be split into two groups: sellers, who want to sell their NFTs and buyers, who want to buy NFTs. Sellers sell NFTs by filling orders represented by the `Order` data structure (lines 2-5), which contains the seller's address (the `signer` field), a signature (the `signature` field), a price for a singular NFT (the `priceperNFT` field), and the NFTs that the seller wants to sell (the `nfts` array). This order is then made visible for

```
1  contract MarketContract{
2    struct Order {
3      address signer; bytes signature;
4      uint pricePerNFT; Item[] nfts;
5    }
6
7    /**
8    Batch buys or sells orders with specific `1` NFTs.
9    Transaction initiated by an end user.
10   */
11   function takeMultipleOneOrders(
12     Order[] calldata orders) external payable {
13     uint256 totalAmount = 0;
14     for (uint256 i = 0; i < orders.length; i++) {
15       require(validateSignature(orders[i]));
16       totalAmount += orders[i].pricePerNFT;
17       _takeOneOrder(orders[i].signer, orders[i].nfts);
18     }
19     require(msg.value >= totalAmount);
20   }
21   function _takeOneOrder(addresss holder, Item[] nfts)
22     internal {
23     _transferNFTs(holder, nfts);
24   }
25 }
```

Fig. 10: Implementation specific bug example

any user to see off-chain. Any buyer who is interested in an order can call a variety of functions to "take" or "fullfill" the order by paying the price. One of these functions is the `takeMultipleOneOrders` function starting at line 11. The function will through all the orders (line 14) that a buyer wants to fill, and aggregate a total price by updating the `totalAmount` variable at line 16. Then, the NFTs included within the orders are transferred to the buyer via the function call at line 17, which eventually does the transfers via the operation at line 23. Finally, there is a statement at line 19 that requires that the amount the buyer pays to the function, `msg.value`, is at least the `totalAmount`. However, according to the developer's comment in lines 7-10, the function `takeMultipleOneOrders` should only take orders with 1 NFT inside. Hence, the `totalAmount` increase at line 16 only does so by the `pricePerNFT` per `Order`. The function does not prevent buyers from calling the function with orders with more than one NFT inside. Therefore, malicious buyers could call the `takeMultipleOneOrders` function with orders including more than one NFT and receive all of the NFTs for the listed price of one. This exploit is unlikely to appear in any other contract as it is a very specific inconsistency between code and specification. Developers eventually patched the bug by providing the necessary checks to satisfy the conditions needed to call the function. □

## E. 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. 11 presents a real-world vulnerability in th PancakeSwap lottery contract [63]. It was reported by an anonymous whitehat and awarded with an undisclosed bounty [64]. 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

```
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][id[i]] += amounts[i];
40       totalAmount += amounts[i];
41     }
42     receivePayment(msg.sender, totalAmount),
43   }
44 }
```

Fig. 11: The PancakeSwap Lottery vulnerability

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 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 [65]. 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* [66], whose root cause is usually atomicity violation. □

**Abstract Bug Model (RQ4) and Remedy (RQ5).** 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 [67] 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 [68]. 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).