Demystifying Exploitable Bugs in Smart Contracts

***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. Introduction

Since the Bitcoin and blockchain techonology were introduced in 2008, their market capitalization has experienced an explosive growth, reaching over $438 billion (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 billion were exploited from various smart contracts as of 1 May 2022.

A large body of techniques have been proposed to detect smart contract vulnerabilities such as reentrancy and integer overflows, and they can be classified into categories such as fuzzing [3]–[7], formal verification [8]–[14], and runtime verification [15], [16]. Despite the success of these 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, security vulnerabilities are largely different from functional bugs. The former has limited forms such as buffer overflow (leading to control flow 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 exploitable smart contract bugs to understand the underlying critical properties. In this paper, we study a large set of 516 exploitable bugs from 167 real-world contracts reported/exploited in years 2021-2022, and aim to summarize their root causes and distributions. We collect these bugs from the highly reputable *Code4rena* contests [21] (with a total of 462 bugs), which invite individuals and companies from all over the world to audit real-world contracts by providing substantial bounties [22], and from various real-world exploit reports (e.g., those from [23], [24]), with a total of 54 exploits. The real-world exploits account for $256.3 millions monetary loss. In the study, we answer a few research questions such as how many such bugs can be detected by existing tools, how difficult is it to detect such bugs, the root causes of those that cannot be detected by tools, their consequences, repair strategies, and distributions. The detailed setup of our study is in §III. Compared to existing surveys and studies on smart contract bugs (e.g., [25]–[28]), we collect the latest bugs and study them from many unique perspectives such as tool coverage, distributions and difficulty levels. We have 11 findings. Some of them are highlighted in the following.

- More than 80% exploitable bugs are beyond existing tools. This is largely due to the lack in describing and checking the corresponding domain-specific properties.
- Majority of exploitable bugs in the wild are hard to find, including those within and beyond the scope of tools.
- The 80% exploitable bugs that are beyond tools, called *machine unauditable bugs* (MUB), can be classified to 7 categories. Two of the categories (accounting for 40% of the MUBs) are project/implementation specific such that general oracles may not exist. The remaining 5 categories have clear symptoms and can be properly abstracted such that automated oracles may be devised.
- Different types of MUBs have different distributions and different difficulty levels, with *price oracle manipulation* (38%) and *privilege escalation* most popular in real-world exploits and *accounting errors* most popular in bugs found during audit contests.
- MUBs are easy to fix, requiring 15 LoC on average.

Contributions. We make the following contributions.

- We conduct the first comprehensive study of a large number of smart contract vulnerabilities and identify the missing gaps and difficulties in exploitable bug detection.
- We classify the exploitable bugs into different categories, and extract their essence and root causes. We have a

number of findings, which may have ramifications for future tool building.
- We compile and explain the needed background to ensure the paper is self-contained, as understanding many bugs requires substantial domain knowledge. We provide real-world examples for each bug category as well.
- We demonstrate the importance of our findings by our preliminary success in finding 15 zero-day exploitable bugs in real-world smart contracts. These bugs could endanger $22.52 millions funds if exploited.

II. Background

To make the paper self-contained, we briefly introduce the terms (italic and underlined) needed to understand the rest of the paper. Experienced readers can skip this section.

Ethereum Blockchain. *Ethereum* [29] is an advanced framework for the development of custom financial products on the web. This is made possible through the underlying *blockchain* [30], which 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. Ethereum requires anyone who submits a transaction to provide an appropriate amount of *gas*, which is a fee paid to miners when they process transactions. Transactions on Ethereum are transparent and decentralized. As of 29 August 2022, Ethereum has a market capitalization of more than $187 billions [31].

Smart Contracts. *Smart contracts* are applications that provide functionalities to realize some business model. They are usually implemented by specific programming languages such as *Solidity* [32] or *Serpent* [33], leveraging the primitive services provided by Ethereum. Smart contracts are publicly available for *users* to access and no changes to the internal states of a smart contract can be hidden, hence the service *transparency*. 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 has two kinds of functions: *external* and *internal*. The former can be invoked by a user or the owner, and the latter can only be invoked by another function within the contract. A *transaction* starts when a user invokes an external function. A transaction has *atomicity*, meaning that 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 undone. 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 an external function (by user) when one is going on. Smart contracts can interact with each other, constituting a *decentralized finance* (DeFi) [34].

```
1  contract ERC20 {
2    // owner => spender => amount
3    mapping (address => mapping (address => uint256))
4      internal _allowances;
5
6    function _approve(address owner, address spender,
7      uint256 allowance) internal {
8      _allowances[owner][spender] = allowance;
9    }
10
11   function transferFrom(address from, address to,
12     uint256 amount) external {
13     require(_allowances[from][msg.sender] >= amount);
14     _approve(from, msg.sender,
15       _allowances[from][to] - amount);
16     _transfer(from, to, amount);
17   }
18 }
```

Fig. 1: The Redacted Cartel exploit

Solidity. Syntax-wise, Solidity is similar to Java/JavaScript. A *contract* is similar to a class in Java. Solidity provides a *require* operation that asserts a condition. If the assertion fails, an error message is emitted and the current transaction is reverted. Solidity uses *msg.sender* to denote the caller of current (external) function, and *this* to denote the current contract.

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, in order to realize the business models that developers envision. Ethereum resolved this issue with the creation of *Ethereum Request for Comment* (ERC) tokens. Intuitively, assets are denoted by various kinds of tokens. Tokens can be *fungible* or *non-fungible* (i.e., NFTs). ERC20 [35] tokens are fungible, meaning that they are non-unique and interchangeable. An example would be that denoting a real-world dollar bill. ERC721 [36] and ERC1155 [37] 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. Users can buy/sell tokens by dealing with their central contracts.

Exploitable Bugs and A Real-world Example. We call bugs that can cause direct monetary loss *exploitable bugs*. Figure 1 depicts an example in Redacted Cartel [38]. An ethical hacker reported this bug and was rewarded with a $560,000 bounty. Specifically, it is a fungible token contract which piggybacks on real-world assets (e.g., USDC tokens backed by US Dollars). The first line defines a contract `ERC20`. Lines 2-4 define `_allowances`, a two-level mapping denoting the

amount of fungible tokens that the owner allows a spender to spend. Note that the first-level key is the address of owner, and the second-level key is the address of spender. It is an *internal* field that can only be accessed by the contract's functions. Lines 6-9 defines an internal function, `_approve()`, which updates the amount of allowance. It internally updates `_allowances[owner][spender]` at line 8. Lines 11-17 define a function `transferFrom` that transfers `amount` tokens from address `from` to address `to`. It is an *external* function that can be invoked by any parties including users and other smart contracts. At line 13, the function first validates that `msg.sender` has sufficient allowance from address `from`. It is achieved by the `require` operation. Lines 14-15 update the caller's allowance via function `_approve`. Line 16 invokes `_transfer` to update the balances of `to` and `from`. The bug happens at line 15, where the contract mistakenly uses the allowance of `to` instead of `msg.sender`. That is, the correct allowance to update is `_allowances[from][msg.sender]`. Considering that a victim user Alice grants Bob an allowance of 10 tokens, an adversary Eve can invoke `tranferFrom(Alice, Bob, 0)` without any token transferred. However, since line 15 updates Eve's allowance as `_allowance[Alice][Bob] - 0`, Eve illegally gains 10-token allowance of Bob.

Observe that this bug aligns better with functional bug in traditional software while being exploitable. Human auditors and automatic tools can hardly detect it without understanding the meaning of `_allowances` and `transferFrom`, as well as the business model. The bug survived multiple rounds of auditing where automatic tools have been applied.

III. Research Questions and Study Methodology

In this section, we first present the scope and research questions of this study. We then explain our methodology of collecting and analyzing data, as well as the threats to validity.

Threat Model and Scope of Our Study. In our threat model, the adversary is a contract user who crafts special inputs to exploit the on-chain contract and further cause monetary loss. Other attacks such as insider attacks and spam attacks are out of scope. Insider attacks are launched by privileged users of the contract (e.g., owners who might steal funds by leveraging the 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. Since our study focuses on vulnerabilities of on-chain contracts, we also exclude attacks where off-chain components get involved.

Research Questions. We target the following four key research questions. We call exploitable bugs that can be detected by existing automatic tools *machine auditable bugs* and the others *machine unauditable bugs*.

- **(RQ1)** What kinds of exploitable bugs are machine auditable by existing tools? How many real-world exploitable bugs are machine auditable?
- **(RQ2)** How difficult is it to audit exploitable bugs?
- **(RQ3)** What are the root causes, categories, and distributions of machine unauditable bugs?

TABLE I: Categories of on-chain projects

Categories	Description
Lending	Allow users to borrow and lend assets
Dexes	Allow users to swap/trade crypto-currency
Yield	Reward users for their staking
Services	Service providers, e.g., tokenization
Derivatives	Projects that get the value, risk, and basic term structure from an underlying asset, e.g., options
Yield Aggregator	Aggregate yield from a set of other projects
Real World Assets	Projects that associate their values with real-world assets, e.g., stocks
Stablecoins	Cryptocurrencies that attempt to peg their market value to some external reference, e.g., US Dollar
Indexes	Projects that have a way to track the performance of a group of related assets
Insurance	Projects that provide monetary insurance
NFT Marketplace	Projects where users can buy/sell/rent NFTs
NFT Lending	Allow users to colletarize NFTs for loans
Cross Chain	Provide interoperability among blockchains

TABLE II: Basic information of Code4rena contests. **# Cont** and **# vuln** denote the numbers of hosted contests and in-scope bugs, respectively. **# Atten** denotes the number of auditors who have attended at least one contest of the corresponding category, while the **total # atten** denotes the total number of auditors who have ever participated in Code4rena contests. **TVL** denotes the overall value of crypto assets deposited in the corresponding DeFi projects, i.e., the worth of these projects.

Categories	# Cont	Bounty	# Atten	# Vuln	TVL
Lending	20	$1,145K	180	53	$304.8M
Dexes	13	$1,020K	139	70	$898.9M
Yield	12	$ 970K	193	85	$304.8M
Services	11	$ 532K	123	21	$219.8M
Derivatives	9	$ 525K	123	13	$147.8M
Yield Aggregator	9	$ 365K	124	22	$265.5M
Real World Assets	7	$ 405K	69	10	$ 41.8M
Stablecoins	6	$ 365K	102	7	$364.7M
Indexes	6	$ 215K	101	7	$ 1.0M
Insurance	5	$ 298K	74	19	$ 42.9M
NFT Marketplace	4	$ 266K	126	8	$ 46.6M
NFT Lending	4	$ 230K	108	10	$ 8.2M
Cross Chain	4	$ 250K	81	7	$ 32.0M
Others	3	$ 110K	25	9	$118.3M
Total	113	$6.696M	358	341	$2.797B

- **(RQ4)** What are the symptoms and fixes of machine unauditable bugs? Can they be properly abstracted such that automated oracles can be devised.

The first two questions target all exploitable bugs, including machine auditable and unauditable, to understand the success and limitations of existing tools. The last two focus on the latter kind on which the community shall place their efforts.

Data Collection. We collect two datasets of bugs, from the Code4rena contests and real-world exploit reports.

Code4rena Contests. Code4rena [21] is a highly reputable audit contest platform. Each Code4rena contest lasts for 3-7 days and aims to have real-world DeFi projects audited *before official deployment* (pre-deployment), for which the developers of subject projects commit a bounty in the range of $20K to $1M as incentive. Individuals, companies, and institutes from all over the world can participate. After the contest, a group of Code4rena judges (i.e., very experienced auditors elected

TABLE III: Basic information of surveyed real-world exploits

Categories	Attacks		Bug Bounties	
	# Bugs	Fund loss	# Bugs	Bounties
Lending	1	$ 5,000K	2	$ 1,630K
Dexes	7	$ 13,950K	3	$ 65K
Yield	6	$ 20,300K	1	$ 10K
Services	3	$ 5,600K	2	$ 610K
Derivatives	-	-	2	$ 200K
Yield Aggregator	1	$ 2,100K	2	$ 300K
Real World Assets	2	$ 1,127K	1	$ 50K
Stablecoins	5	$211,360K	-	-
Indexes	-	-	1	$ 90K
NFT Marketplace	1	$ 20K	-	-
NFT Lending	2	$ 5,800K	-	-
Cross Chain	-	-	1	$10,000K
Others	-	-	1	$ 1,050K
Total	28	$265,257K	16	$14,005K

by the community) and the project's developers get together to inspect the bug reports, where they confirm the valid ones, classify reports based on root causes, and decide the criticality level of bugs. Note that each bug is assigned a criticality level: low, medium, or high, where only high-risk bugs can cause assets loss (and hence are exploitable) [39]. The final reward is decided by both the criticality level of bug and the number of reports submitted for the bug (more submissions lead to a lower reward as the bug is easier than others).

We collect and analyze 462 unique high-risk bugs from 113 Code4rena contests hosted between April 2021 and June 2022. For each case, we inspect the bug report, the faulty contracts (which are available through Github), and the project's documentation. Following the suggestions in Claes et al. [40], each bug is checked by at least two individual researchers. Any disagreement will be turned to an additional researcher. We reach consensus for all cases after the new researcher gets involved. All our researchers are experienced auditors, having participated 23 contests from February 2022 to June 2022. One of them was invited to be a consultant for judges.

Among the 462 surveyed bugs, we identify 341 in-scope bugs (exploitable by remote users). Table II presents the basic information of surveyed contests and the in-scope bugs. The first column presents the categories of on-chain projects, following the taxonomy by DefiLlama [41], a leading DeFi analytics platform. The description of each category is presented in Table I, while details are available in §I of our supplementary material. Observe that around $2.8 billions are protected by Code4rena auditing, indicating the representativeness of the dataset, and $6.7 millions are committed as bounties.

Real-world Exploits. Our second dataset comprises 54 real-world exploits, collected from postmortems and bugfix reviews of real-world exploits from January 2022 to June 2022. These reports are published by highly-reputable security researchers (e.g., [23], [42]) and companies (e.g., [24], [43]–[45]). We follow the aforementioned study methodology (for Code4rena reports). Overall, we identify 44 (out of 54) in-scope bugs. Table III presents the basic information. Real-world exploits target *post-deployment* contracts, including real attacks launched against on-chain contracts and caused real asset damage (i.e., *attacks*), and the cases in which ethical hackers demonstrated vulnerabilities in a local off-chain environment and were awarded bug bounties by the projects (i.e., *bug bounties*). The first column of Table III denotes the categories. Columns 2-3 denote the number of in-scope bugs and fund loss by attacks respectively, while columns 4-5 denote the ones for bug bounties. Observe that, while $14 million were paid as incentives to ethical hackers, over $265 million were lost due to real attacks in the first half of 2022; despite the substantial auditing efforts paid prior to deployment, there are still many post-deployment exploitable bugs.

> ***Finding 1:*** *Although the DeFi community has heavily invested on protecting their products, the current supply of tools and human auditor resources have not met the demand.*

Threats to Validity The *internal* threat to validity mainly lies in 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. Disagreement will be turned to an additional author. The categorization is agreed on by all the authors. Most authors have extensive smart contract auditing experience and cyber-security/software-engineering 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.

IV. (RQ1) On the Effectiveness of Exsisting Automatic Tools

To understand the capabilities of existing techniques, we examine the literature to summarize the kinds of bugs that can be detected by existing methods. We then study how many of the exploitable bugs in our datasets fall in their scope. To empirically support the correctness of our examination, we also apply two state-of-the-art commercial tools to our datasets to see whether they can detect the bugs.

In particular, we examine papers published on top-tier Software Engineering, Security, and Programming Language venues from 2017 to 2022. Overall, we include 38 existing methods and summarize the bugs handled by them into 17 types. We call them *machine-auditable bugs* (MAB). Table IV presents the MABs, with more details available in §II of our supplementary material. An important observation is that their test oracles are *general* and *sufficiently simple to support instantiations in a wide range of projects*. They hence have a similar nature to general oracles used in traditional software such as buffer-overflow and use-after-free. For example, control-flow hijack bugs (**CH**) use an oracle similar to that used in *control flow integrity* (CFI) [80], [81] in traditional software. Reentrancy bugs (**RE**) use an oracle that detects cycles in an transaction, which is generally applicable to all contracts. Therefore, they can hardly cover functional

TABLE IV: Categories of machine-auditable bugs

ID	Bug Name	Description
AF	Assertion Failure	Assertion is not satisfied.
AW	Arbitrary Write	Arbitrary storage data gets overwritten due to mismanaged objects or improper proxies
BD	Block-state Dependency	Ether transfer depends on block states, e.g., `block.timestamp` or `block.number`.
CE	Compiler Error	The contract mis-behaves due to using an outdated compiler which contains known bugs.
CH	Control-flow Hijack	Users can arbitrarily control the destination of a control-flow transfer.
EL	Ether Leak	User can freely retrieve ether from the contract.
FE	Freezing Ether	No one can retrieve a (large) portion of locked ether from the contract.
GI	Gas-related Issue	Execution fails due to insufficient gas.
IB	Integer Bug	Integer overflows or underflows.
ME	Mishandled Exception	The contract does not check an exception from external function invocations.
PL	Precision Loss	Significant precision loss during calculation.
RE	Reentrancy	A victim function gets re-entered by an untrusted callee, leading to state inconsistency.
SC	Suicidal Contract	An arbitrary user can destroy the contract.
TD	Transaction-ordering Dependency	The result of an execution trace depends on another trace sent by a different sender.
TO	Transaction Origin Use	The result of an execution trace depends on `tx.origin` for user authorization.
UV	Uninitialized Variable	Uses of uninitialized storage variables.
WP	Weak PRNG	A pseudo-random number generator (PRNG) relies on predictable variables.

bugs that require domain-specific or even application-specific oracles [82] (e.g., the Redacted Cartel bug in Figure 1).

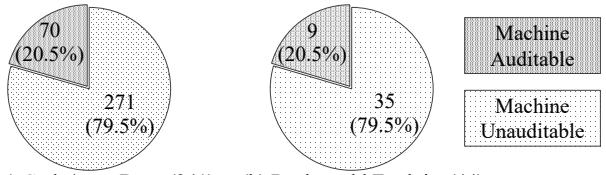
Table V provides the examination results of existing works. We classify them into four categories: fuzzing, static analysis, formal verification, and symbolic execution. Overall, there are 12 commercial tools, developed by leading companies, such as Trail of Bits [83] and ConsenSys [84]. Also observe that most commercial tools provide coverage for a wide variety of bugs. Most existing works (35 out of 38) rely on general and simple oracles, or hand-coded specifications (e.g., Echidna and VeriSol). ContraMaster proposes an interesting general oracle that has the potential to cover a wide range of functional bugs. That is, for a single asset, the total balances of all parties should not change. Although the advantages of having such a general invariant are prominent, many modern DeFi projects employ aggressive and complex business models that are beyond this invariant. For example, lending projects are naturally designed for multi-asset business within which the total balances of a single asset is volatile. It is interesting to see if similar invariants can be developed for these new contracts.

> ***Finding 2:*** *Existing techniques rely on simple and general oracles or hand-coded ones that are project specific. Such oracles may not be sufficient for functional bugs in general.*

For any bug in our datasets, as long as it falls into the scope of any existing work in Table V (assuming 100% precision and recall of these tools), we consider it machine-auditable. Figure 2 depicts the breakdown of machine auditable and unauditable bugs in our datasets. Observe that, despite the over-approximation, only 20% exploitable bugs can be detected by existing works, disclosing a significant supply shortage of

TABLE V: Summary of existing tools. **Com** denotes whether it is a commercial tool widely used in real-world auditing. **Orcl** stands for *test oracles*, where ○, ◐, and ● denote fixed and simple oracles, hand-coded oracles, and oracles that can automatically adapt to cover a wide range of functional bugs, respectively. The remaining columns present bug coverage.

Kind	Tool	Year	Com	Orcl	Machine-auditable Bugs															
					AF	AW	BD	CE	CH	EL	FE	GI	IB	ME	PL	RE	SC	TD	TO	UV	WP
Fuzzing	ReGuard [46]	18		○												✓					
	ContractFuzzer [47]	18		○			✓		✓		✓	✓		✓		✓					
	ILF [7]	19		○			✓		✓	✓	✓			✓		✓	✓				
	Vultron [48]	19		○								✓	✓	✓		✓					
	sFuzz [49]	20	✓		✓	✓	✓	✓	✓	✓		✓	✓								
	Harvey [3]	20	✓	○	✓	✓							✓	✓		✓					
	ContraMaster [50]	20		●								✓	✓	✓		✓					
	ConFuzzius [51]	21		○	✓		✓		✓	✓	✓		✓	✓		✓	✓	✓			
	Smartian [6]	21	✓	○	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓		✓		
	Echidna [52], [53]	21	✓	◐	✓																
	xFuzz [54]	22		○					✓							✓			✓		
Static Analysis	Gasper [55]	17		○								✓									
	Securify [56]	18	✓	○		✓	✓		✓	✓	✓			✓		✓	✓	✓	✓		
	Vandal [57]	18		○						✓	✓			✓		✓	✓		✓		
	MadMax [58]	18		○								✓	✓	✓							
	SASC [59]	18		○			✓	✓								✓			✓		
	SmartCheck [60]	18	✓	○			✓				✓	✓	✓	✓		✓			✓		
	Zeus [61]	18		○			✓						✓	✓		✓	✓	✓	✓		
	Slither [14]	19	✓	○		✓	✓	✓	✓	✓	✓			✓	✓	✓	✓		✓	✓	✓
	Sereum [62]	19		○					✓							✓					
	NPChecker [63]	19		○			✓		✓							✓		✓	✓		
	Sensors [64]	22		○					✓				✓			✓					
	Remix [65]	22	✓	○	✓		✓							✓		✓			✓		
Verification	ECF [66]	17		○												✓					
	Solc-Verify [67]	19		○	✓								✓			✓					
	VeriSol [68]	19	✓	◐	✓																
	VeriSmart [69]	20		○						✓			✓								
	Solid [70]	21		○												✓					
Symbolic Execution	Oyente [8]	16	✓	○	✓		✓				✓		✓			✓	✓	✓			
	Maian [71]	18		○						✓	✓						✓				
	teEther [72]	18		○	✓				✓	✓							✓				
	Osiris [73]	18		○	✓		✓						✓			✓					
	Manticore [74]	19	✓	○	✓		✓		✓	✓			✓	✓		✓	✓		✓		
	sCompile [75]	19		○	✓				✓	✓	✓					✓	✓				
	M-A-R [76]	21		○												✓					
	SmarTest [77]	21		○	✓					✓			✓				✓				
	Mythril [78]	22	✓	○	✓	✓	✓		✓	✓			✓	✓		✓	✓	✓	✓		
	Sailfish [79]	22		○												✓		✓			



(a) Code4rena Bugs (341) (b) Real-world Exploits (44)

Fig. 2: Breakdown of the bugs

automated bug finding capabilities. We empirically validate the finding. Specifically, we run Slither [14] and Oyente [8], two state-of-the-art commercial tools, on our datasets. *Neither can detect any machine unauditable bug (by our classification).*

> ***Finding 3:*** *A large portion of exploitable bugs in the wild (i.e., 80%) are not machine auditable.*

We speculate the main reason is Finding 2 – existing tools have limited oracles, i.e., only checking limited properties.

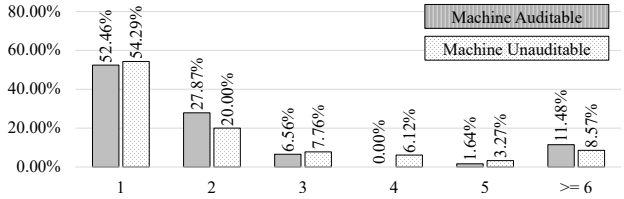


Fig. 3: Overall auditing difficulty. Each bar denotes how many machine (un-)auditable bugs are reported by the given number of auditors, where x-axis and y-axis denote the number of auditors and the ratio w.r.t. the total number of machine (un-)auditable bugs, respectively.

Note that it does not suggest existing tools are ineffective. It is well possible that a large number of machine auditable bugs have been detected and prevented during development (and hence not present in our datasets).

V. (RQ2) On the Difficulty of Auditing Exploitable Bugs

It is in general very hard to determine the difficulty level of detecting certain bugs, by tools or manual efforts. However, the Code4rena contests provide a perfect platform to quantify bug difficulties. Specifically, each contest is participated by a large number of independent auditors, who submit their reports separately. Although there are skill level variations of the auditors, we consider the number of submitted reports for a bug suggests the *relative* difficulty level in finding the bug, intuitively, fewer bug reports, harder to find.

Figure 3 delineates the difficulty of auditing exploitable bugs. It shows that 52.46% of machine auditable bugs and 54.29% of machine unauditable bugs are only reported by a single auditor, and hence most difficult. The ratios for machine auditable and unauditable bugs found by two auditors are 27.87% and 20.00%, respectively. Only around 25% of exploitable bugs are found by three or more auditors.

Finding 4: *Majority of exploitable bugs are difficult to find.*

Also observe that the difficulty distributions of machine auditable and unauditable bugs are quite similar. That is, the majority of bugs in either kind are difficult. There are multiple possible explanations. One is that the machine auditable bugs in the wild are already left-over after tool scanning during development. As such, they are found by manual efforts during contests. Note that it is impossible to know if the auditors used tools or manual efforts to find these bugs. Another explanation is that bugs that are difficult for humans are likely difficult for tools as well due to similar inherent challenges in analysis.

Finding 5: *There are no obvious differences between audit difficulty distributions of machine-auditable and machine-unauditable bugs.*

VI. (RQ3) On the Categories of Machine Unauditable Bugs

Since 80% of exploitable bugs are not machine auditable, we focus on such bugs in the rest of the paper. In this section, we aim to categorize machine unauditable bugs according to their root causes and study their distributions.

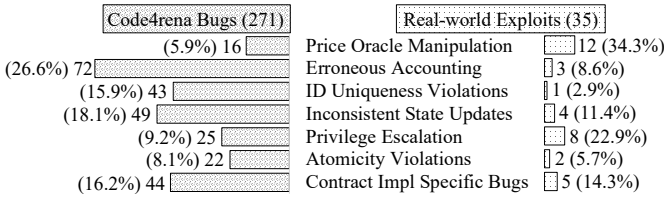


Fig. 4: Breakdown of different types of MUBs

TABLE VI: Auditing difficulties for MUBs of different types

Types	# Auditors					
	1	2	3	4	5	>= 6
Price Oracle Manipulation	75.00%	12.50%	0.00%	0.00%	0.00%	12.50%
Erroneous Accounting	59.09%	21.21%	7.58%	6.06%	3.03%	3.03%
ID Uniqueness Violations	42.86%	17.14%	8.57%	11.43%	5.71%	14.29%
Inconsistent State Updates	53.33%	22.22%	2.22%	6.67%	6.67%	8.89%
Privilege Escalation	56.52%	21.74%	8.70%	4.35%	0.00%	8.70%
Atomicity Violations	57.14%	19.05%	4.76%	4.76%	4.76%	9.52%
Contract Impl Specific Bugs	46.15%	20.51%	17.95%	5.13%	0.00%	10.26%

A. Root Causes and Categorization

The 271+35 machine unauditable bugs can be grouped into 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 Figure 4. We also present their difficulty levels in Table VI, using the same metric as Figure 3.

(C1) Price Oracle Manipulation. Smart contracts usually resort to external authorities on Ethereum, which 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 illegal profits. More detailed explanation and an example can be found in §VII-A. 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 Figure 4, it constitutes 6% of the Code4rena bugs (the least common bug) and 34% of the real-world exploits (the most common exploit). Table VI shows that the auditing difficulty of such bugs is significantly higher than others. As such many of them evade auditing and get exploited after deployment.

(C2) Erroneous Accounting. Many smart contracts implement complex business models. The implementations hence involve a lot of difficult-to-interpret numerical computation. We call incorrect implementations of underlying business model formulas *erroneous accounting* bugs. These bugs usually introduce small errors every time they are exercised. However, these errors can accumulate and induce substantial loss. For example, *Compound Finance* [85], a flagship lending contract, was exploited and had over $80 millions stolen, due to an unnoticeable problematic calculation of annual percentage yield [86]. The bug survived 9 rounds of auditing by top

TABLE VII: Breakdown of MUBs w.r.t. DeFi categories

Categories	Code4rena Bugs						
	C1	C2	C3	C4	C5	C6	C7
Lending	3	6	4	7	9	6	8
Dexes	2	16	8	15	3	1	6
Yield	7	23	17	8	5	6	10
Services	0	4	2	5	0	0	3
Derivatives	1	6	1	0	2	0	1
Yield Aggregator	1	6	0	5	0	1	4
Real world assets	1	0	4	3	0	0	1
Stablecoins	0	2	1	0	0	0	2
Indexes	0	0	1	2	0	2	0
Insurance	0	3	1	3	3	2	4
NFT Marketplace	0	1	1	0	1	2	1
NFT Lending	1	1	1	0	1	1	1
Cross Chain	0	1	1	1	1	0	1
Others	0	3	1	0	0	1	2

security companies [87] and even formal verification [88]. It was not found until being exploited. Figure 4 shows that it is the most popular type of Code4rena bugs (27%) and the 5th most popular type of real-world exploits. As Table VI shows, its auditing difficulty is slightly above average, with around 59% being found by a single auditor. The reason is that finding such bugs requires substantial domain knowledge. The very broad participation of Code4rena contests seems to provide a good coverage of domain expertise such that many these bugs can be captured (although each by very few auditors). More details are in our supplementary material (§IV).

(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 needed in these processes and entities/assets ought to be uniquely represented. Within smart contract implementation, entities and assets are usually denoted as data structures, which often 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 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 bugs *ID uniqueness violations*. It constitutes 16% of the Code4rena bugs (43 out of 271) and 3% of real-world exploits (1 out of 35). It is the 4th and the 7th most commonly seen type of bugs in the two respective datasets. Such bugs are relatively easy to find, with 57% reported by multiple auditors. This could explain its distribution difference in the two datasets as ID bugs may be largely found during auditting (e.g., Code4rena contests).

(C4) Inconsistent State Updates. Smart contracts have many state variables (e.g., debts and collaterals) with implicit correlations. For example, 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. 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 [89]–[91] caused around $3.8 millions loss and also the collapse of a smart contract's internal economy. It constitutes 18% of the Code4rena bugs (49 out of 271) and 11% of the real-world exploits. It is the 2nd and the 4th most commonly seen bugs in the two datasets. The bug difficulty level is about average.

(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 22.9% 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 runtime and hence they do not need to be concerned. However, the runtime only ensures each transaction is atomic, and business flow atomicity, if needed, has to be ensured by the developers. Atomicity violations constitute 8.1% of the Code4rena bugs and 5.7% of real-world exploits. It is the least common bugs in auditing, and the second least in the wild. They are slightly harder to find (than the others), with 57% of bugs found by only one auditor. The reason is that it is difficult to determine business flows and if they need atomicity.

(C7) Contract Implementation Specific Bugs. We find that 16% of the Code4rena bugs and 14% of the real-world exploits are implementation specific, meaning that they do not have a general oracle and unlikely appear in a different smart contract. They hence have a low priority because abstracting them may not provide as valuable guidance as the others. The Redacted Cartel bug in Figure 1 is an example.

Finding 6: *Machine unauditable bugs (MUBs) can be classified to 7 categories, with 85% belonging to categories C1-C6 that are not project specific.*

Finding 7: *Different types of MUBs have different popularity, with accounting errors (C2) and price oracle manipulation (C1) most popular in the Code4rena bugs and the real exploits, respectively. Auditing is particularly effective in preventing certain bugs such as accounting errors.*

Finding 8: *Different types of MUBs have different auditing difficulties, with price oracle manipulation and ID uniqueness violation bugs the hardest and the easiest, respectively.*

B. Bug Distributions in Different Types of Projects

To understand what kinds of bugs are more likely in a specific type of contracts, we study the distributions of MUBs in different DeFi categories. Table VII presents the results of Code4rena bugs. Note that we do not include real-world exploits because only 3 out of the 14 (DeFi) categories have more than 3 exploits, which may induce substantial threat to validity. The grey scale denotes the prevalence. For example, 55% (i.e., $6/11$) of derivative projects' bugs are caused by erroneous accounting (C2) but only 30% (i.e., $23/75$) for yield projects. The former is therefore darker than the latter. Observe that a few bug types are particularly prevalent in some DeFi categories, such as erroneous accounting (C2) and inconsistent state update (C4) bugs in Dex projects. This is mainly due to the unique nature of these projects. For example, Dex projects swap and trade assets. They use complex computation to deal with the volatility of crypto-currency, and are hence prone to erroneous accounting bugs. These suggest that auditors may want to devise different auditing strategies for different types of projects, e.g., prioritizing the prevalent bug types. We follow such strategies during our guided auditing (§VIII).

Finding 9: *Different kinds of DeFi projects tend to be prone to different types of MUBs.*

VII. (RQ4) On the Symptoms and Fixes of Machine Unauditable Bugs

We use real example of (C1) price oracle manipulation and (C5) privilege escalation bugs, the most popular real-world exploits to demonstrate their symptoms and repair strategies. We also provide an abstract model for each bug, which could facilitate future scanning tool and test oracle building.

A. Price Oracle Manipulation (C1)

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 [92], 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) [93], e.g., x for WETH and y for USDC. The price of one asset over the other is hence decided 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

```
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 * 80 / 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. 5: Price oracle manipulation exploit in Deus Finance

from Uniswap, its explanation, and an example are presented in §III-A of our supplementary material.

Price Oracle Manipulation. Despite being pivotal for DeFi project development, price oracles are occasionally used improperly by application contracts, rendering their price queries vulnerable. 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) to avoid the expensive gas costed by the official API. 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. Recall that block-chain transactions are atomic so that any action sequence in a single transaction 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.

Example. Fig. 5 presents a vulnerable code snippet, which is slightly modified from a real-world exploit against the Deus Finance causing a loss of $3.1 millions. The bug survived at least one publicly-known audit round [94]. 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.

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 AMM 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. □

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 §III-B of our supplementary material.

Abstract Bug Model and Remedy. Given 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.

```
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. 6: A voting contract vulnerability

B. Privilege Escalation (C5)

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

Example. Fig. 6 presents a real-world case from an anonymized contract (upon developers' request). The code is completely rewritten to ensure 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 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. Note that both functions `vote` and `end` strictly constrain the invocation time, which constitutes an access control preventing the two functions from being invoked 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 §VII-A). However, an unexpected call sequence can evade the access control. Specifically, consider an adversary Alice proposes herself as the owner. When the time is approaching the deadline `proposal.sTime + 2 days`, she launches an 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 function provided by all ERC20 tokens to directly transfer the loaned amount to the contract without any access control, 3) invoking `end` to become the owner, 4) getting locked funds by function `getLockedFunds`, and 5) paying off the flash-loan debt. Intuitively, Alice "votes" without calling the `vote` function. The developers did not anticipate such a business flow and hence did not guard properly. □

Abstract Bug Model and Remedy. 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. The fixes are to add the missing access control checks or prevent the unexpected paths.

C. Other Machine Unauditable Bug Types

Other machine unauditable bug types are detailed in our supplementary material (§III -§VII).

> ***Finding 10:*** *Five out of the seven MUB categories (accounting for 60% of MUBs), namely, all except (C2) accounting errors and (C7) implementation specific bugs, have general abstract models which may serve as oracles for future automated tools.*

D. Difficulty of Bug Fix

We inspect the patches for Code4rena bugs. For each bug, we use `git blame` on the up-to-date version of its project's repository. We then find the commit or pull request proposed to fix the bug. We exclude comments, blank lines, and unit tests, when counting the lines of code. Table VIII presents the results. Rows **LoC (+)** and **LoC (-)** denote how many lines of code developers added and removed on average to fix bugs of each type. Observe that (C2) erroneous accounting and (C5) privilege escalation bugs only require around 7-10 lines to fix. (C4) Inconsistent state update bugs require more lines to fix, because the patches tend to use a new `struct` to pack all correlated state variables, leading to more changes.

TABLE VIII: Patches for MUBs.

	C1	C2	C3	C4	C5	C6
LoC (+)	9.6	6.0	12.4	14.8	6.6	8.5
LoC (-)	2.6	4.4	8.4	13.2	0.6	5.8

TABLE IX: Guided auditing

	C1	C2	C3	C4	C5	C6	C7
# Bugs (15)	2	2	1	1	4	2	3

Total Bounty Awarded	$102,660
Total Funds Protected	$22.52 **M**

> ***Finding 11:*** *It is usually not difficult to fix MUBs (i.e., with 15 lines of changes on average).*

VIII. Guided Auditing

We started to audit real-world contracts using our findings as the 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. Table IX summarizes the confirmed bugs for individual bug types. All the confirmed ones are rated *critical*. We also participated in three Code4rena contests in July 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 $\$102,660$ so far and the total funds protected due to our reports add up to $\$22.52$ millions. 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 bugs according to our study and found 2 POMs and 4 PEs. We also prioritize the bug types to audit according to the project's category. 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 VII-B) 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.

IX. Related Work

There are a body of existing empirical studies of smart contract bugs [25]–[28], [95], [96]. Compared to these studies, we do not focus on bugs in the development stage. We study a large number of latest exploitable/exploited bugs. We study unique perspectives such as prevalence, difficulty level, abstract models, and fixes. Detailed comparison can be found in §VIII of our supplementary material.

X. Conclusion

We study 516 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.

[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.

[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.

[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.

[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.

[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.

[12] M. Bartoletti and R. Zunino, "Verifying liquidity of bitcoin contracts," in *International Conference on Principles of Security and Trust*. Springer, 2019.

[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.

[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.

[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.

[21] "Code4rena." [Online]. Available: https://code4rena.com

[22] "Leaderboard - code4rena." [Online]. Available: https://code4rena.com/leaderboard

[23] "samczsun (@samczsun)." [Online]. Available: https://twitter.com/samczsun

[24] "Peckshield twitter." [Online]. Available: https://twitter.com/peckshield

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[30] 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.

[31] "Ethereum market capital 2022." [Online]. Available: https://coinmarketcap.com/currencies/ethereum/

[32] "Solidity documentation." [Online]. Available: https://docs.soliditylang.org/en/v0.8.15/

[33] "Serpent documentation." [Online]. Available: https://www.cs.cmu.edu/~music/serpent/doc/serpent.htm

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

[35] "Erc20 token standard." [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

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

[37] "Erc1155 multi-token standard." [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155

[38] "Redacted cartel custom approval logic bugfix review." [Online]. Available: https://medium.com/immunefi/redacted-cartel-custom-approval-logic-bugfix-review-9b2d039ca2c5

[39] "Judging criteria - code4rena." [Online]. Available: https://docs.code4rena.com/awarding/judging-criteria#estimating-risk

[40] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[41] "Defillama - defi bashboard." [Online]. Available: https://defillama.com/

[42] "pwning.eth (@pwningeth)." [Online]. Available: https://twitter.com/PwningEth

[43] "Paradigm twitter." [Online]. Available: https://twitter.com/paradigm

[44] "Certik twitter." [Online]. Available: https://twitter.com/CertiK

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

[46] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018.

[47] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018.

[48] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "Vultron: catching vulnerable smart contracts once and for all," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2019.

[49] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[50] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

[51] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021.

[52] A. Groce and G. Grieco, "echidna-parade: a tool for diverse multicore smart contract fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[53] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[54] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao, "xfuzz: Machine learning guided cross-contract fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[55] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017.

[56] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[57] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[58] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

[59] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018.

[60] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018.

[61] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018.

[62] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," 2021.

[63] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 2019.

[64] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart contract vulnerability detection model based on multi-task learning," *Sensors*, 2022.

[65] "ethereum/remix-project," 2022. [Online]. Available: https://github.com/ethereum/remix-project

[66] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, 2017.

[67] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Working conference on verified software: theories, tools, and experiments*. Springer, 2019.

[68] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2019.

[69] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[70] B. Tan, B. Mariano, S. Lahiri, I. Dillig, and Y. Feng, "Soltype: Refinement types for solidity," *arXiv preprint arXiv:2110.00677*, 2021.

[71] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, 2018.

[72] J. Krupp and C. Rossow, "{teEther}: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[73] C. Ferreira Torres, J. Schütte *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *34th Annual Computer Security Applications Conference (ACSAC'18), San Juan, Puerto Rico, USA, December 3-7, 2018*, 2018.

[74] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019.

[75] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," in *International Conference on Formal Engineering Methods*. Springer, 2019.

[76] Z. Wang, B. Wen, Z. Luo, and S. Liu, "Mar: A dynamic symbol execution detection method for smart contract reentry vulnerability," in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2021.

[77] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[78] "Consensys/mythril," 2022. [Online]. Available: https://github.com/ConsenSys/mythril

[79] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[80] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[81] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, 2017.

[82] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[83] "Trail of bits." [Online]. Available: https://www.trailofbits.com/

[84] "Blockchain technology solutions." [Online]. Available: https://consensys.net/

[85] "Compound finance website." [Online]. Available: https://compound.finance/

[86] "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/

[87] "Compound — doc — audits." [Online]. Available: https://compound.finance/docs/security#audits

[88] "Compound — doc — formal verification." [Online]. Available: https://compound.finance/docs/security#formal-verification

[89] "Wiener doge exploit." [Online]. Available: https://www.certik.com/resources/blog/Br4j8oVnz9zKqW3okCyD9-wiener-doge-exploit

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

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

[92] "Home — uniswap protocol." [Online]. Available: https://uniswap.org/

[93] R. F. Muth, "The derived demand curve for a productive factor and the industry supply curve," *Oxford Economic Papers*, vol. 16, no. 2, 1964.

[94] "Deus.finance - smart contract audit report." [Online]. Available: https://solidity.finance/audits/DEUS/

[95] 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.

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