A Survey on Security Analysis Methods of Smart Contracts

Huijuan Zhu, Lei Yang, Liangmin Wang, and Victor S. Sheng, *Senior Member, IEEE*

(Survey-Tutorial Paper)

***Abstract*—Smart contracts have gained extensive adoption across diverse industries, including finance, supply chain, and the Internet of Things. Nevertheless, the surge in security incidents of smart contracts over recent years has led to substantial economic losses. Therefore, ensuring the security of smart contracts has become a critical and complex challenge in both academic and industrial domains. Based on 539 real-world security incidents in the Ethereum platform and audit reports from 10 authoritative auditing institutions, we summarize 27 types of exploited security vulnerabilities and draw insights into their principles, typical cases, relevant research and recommended prevention strategies. Besides, we also gather 7 other potentially threatening vulnerability types as supplements. On this basis, we conduct an in-depth analysis of the root causes of vulnerabilities and further formulate eight safety practical rules. Moreover, we perform a comprehensive review of 178 recent papers on smart contract security analysis, classifying detection methods into formal verification, fuzz testing, machine learning, program analysis, and others. For each category, we seize the specific detection tools and analyze them comprehensively. Then, we conduct an extensive analysis and synthesis from various angles, presenting a comprehensive overview of the current research landscape in smart contract security detection. We also discuss current on-chain and off-chain repair methods. Finally, this review outlines major challenges and highlights potential areas for future research in this field.**

***Index Terms*—Smart contracts, blockchain, ethereum, security analysis, vulnerabilities.**

I. Introduction

SMART contracts are programs that can run on blockchain platforms. The design and functionality of smart contracts can be flexibly programmed and adjusted to meet specific business requirements. Similar to traditional applications, smart contracts go through a lifecycle consisting of creation, deployment, execution, and completion. The application scenarios of smart contracts are rapidly expanding, such as the Internet of Things, supply chain, voting, and gaming. As they run in a decentralized network environment and often involve financial assets, they are susceptible to security vulnerabilities. Given the rapid expansion of smart contracts and their crucial role in managing high-value digital assets, the escalating frequency of smart contract attacks poses a critical threat to the security of blockchain platforms.

Ethereum, a typical blockchain platform that supports smart contract deployment and execution, allows developers to create programs with diverse functionalities using programming languages such as Solidity. These smart contracts are executed by each Ethereum node through the Ethereum Virtual Machine (EVM). Currently, more than 65 million smart contracts have been deployed on Ethereum, and this number is continuously growing [1]. Correspondingly, security incidents caused by smart contract vulnerabilities continue to occur. Our study begins with Ethereum security incidents, focusing on three key areas: contract vulnerabilities, analysis methods, and detection tools.

Given the critical importance of smart contract security, related papers have been continually emerging in recent years. As shown in Fig. 1, we surveyed 178 papers on smart contract security from a total of 1332 related papers published between January 2016 and June 2024. The main sources of these papers include ACM Digital Library, IEEE Xplore Digital Library and arXiv. Although some literature reviews have examined contract vulnerabilities, analysis methods, and detection tools, their focuses differ significantly. We summarize these focuses in Table I, where the cells indicate three levels of research: 1) ● for detailed exploration, 2) ◐ for brief coverage, and 3) ○ for no coverage. From Table I, we can notice that, similarly to our work, studies [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39] investigated the security of smart contracts

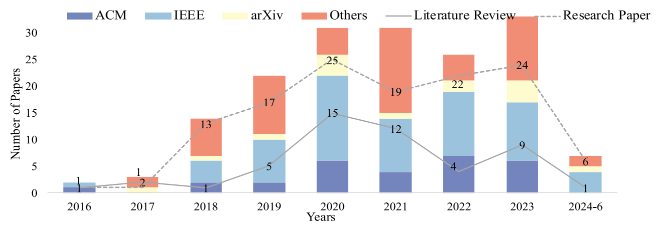

Fig. 1. Distribution of Collected Papers from Jan. 2016 to Jun. 2024.

Received 21 September 2023; revised 9 August 2024; accepted 10 September 2024. Date of publication 18 September 2024; date of current version 30 December 2024. This work was supported by the National Natural Science Foundation of China under Grant 62272204 and Grant 62372105, and in part by the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant BK20202001. *(Corresponding authors: Liangmin Wang; Victor S. Sheng.)*

Huijuan Zhu is with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China (e-mail: huijuanzhu@ujs.edu.cn).

Lei Yang is with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China (e-mail: 2212208026@stmail.ujs.edu.cn).

Liangmin Wang is with the School of Cyber Science and Engineering, Southeast University, Nanjing 211189, China (e-mail: liangmin@seu.edu.cn).

Victor S. Sheng is with the Department of Computer Science, Texas Tech Uni-versity, Lubbock, TX 79409 USA (e-mail: victor.sheng@ttu.edu).

Digital Object Identifier 10.1109/TSC.2024.3463394

TABLE I
RELEVANT STUDIES

Papers	**Vulnerabilities**	**Methods**	**Tools**
Studies with Significantly Different Focuses			
[2], [3], [4], [5], [6]	●	○	○
[7], [8]	○	○	●
[9], [10]	●	○	○
[11], [12]	○	●	◐
[13]	◐	○	●
[14], [15]	◐	◐	●
[16]	○	●	●
[17], [18]	●	○	●
[19], [20], [21]	○	●	●
[22], [23]	●	●	◐
[24], [25]	◐	●	●
Studies Similar to Our Work			
[26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39]	●	●	●
Our Work	●	●	●

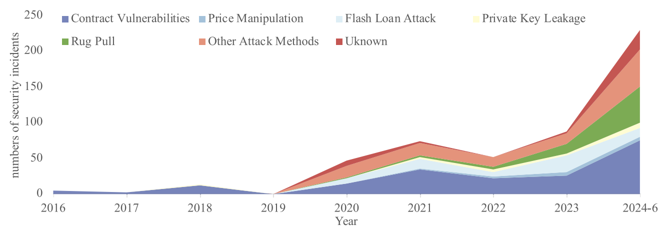

Fig. 2. Distribution of security incidents in ethereum, Jan. 2016 – Jun. 2024.

from three perspectives (i.e., contract vulnerabilities, analysis methods, and detection tools). We further provide a detailed comparison between our work and these methods in Table II, highlighting three research perspectives: contract vulnerabilities (i.e., number of types, principles, typical cases, relevant research and prevention strategies), analysis methodology (whether they conduct principle analysis), and detection tools (whether they conduct comparative analysis). It also shows whether these studies are based on real security data and literature statistics.

From Table II, we can notice that studies [32], [37], [38], and [39] are based on extensive literature statistics and analysis. As we know, academic literature often suffers from publication delays. To address this, real security data (e.g., deployed contracts and security incidents) from the industry can effectively supplement [12]. However, only the study [38] combines literature analysis with real security incidents. We can also observe that only studies [31], [33], [36] and [39] involved typical cases and prevention strategies. These factors indicate an insufficient consideration for practical applications in this field. To bridge this gap, we initially collected 539 real security incidents that occurred on Ethereum from January 2016 to June 2024, as presented in Fig. 2. Audit reports of authoritative auditing institutions are another effective supplement, providing more up-to-date information. Therefore, we integrated 539 real security incidents and audit reports from 10 authoritative auditing institutions, as well as a rigorous statistical analysis of 178 pieces of literature, to provide an overview of the current research landscape. Specifically, we summarized 27 types of exploited vulnerabilities associated with these security incidents and 7 other potentially threatening vulnerability types. Then, we provide a detailed analysis of these identified vulnerabilities, covering their principles, typical cases, relevant research and prevention strategies. Finally, based on an exhaustive review of existing smart contract security analysis methods, as well as detection tools and repair methods, we outline major challenges and highlight potential areas in smart contract security. The main contributions of this paper can be summarized as follows:

- We investigated 539 real security incidents by combining extensive audit reports and research literature to elucidate the principles, typical cases, prevention strategies and relevant research of exploited and other potential vulnerabilities. We then distill the root causes of vulnerabilities and outline eight practical principles.
- We reviewed 178 papers on smart contract security analysis and detection, published between January 2016 and June 2024. This encompasses an exploration of five major categories and the main detection tools available in each category, as well as general repair methods. We also perform a comprehensive summary to provide an overview of the current research landscape in this field.
- We comprehensively examine existing detection and repair works from various aspects and identify current research challenges. Based on these findings, we provide recommendations to support further research in this field.

II. SMART CONTRACTS VULNERABILITIES

In this section, we thoroughly review recent security incidents caused by smart contract vulnerabilities on the Ethereum, which have been disclosed by security auditing platforms (e.g., [40], [41], [42], [43], [44], [45], [46], [47], [48] and [49]). We analyze vulnerabilities and classify them into three groups based on their occurrence stages, i.e., the Solidity code layer, the EVM layer, and the Blockchain system layer [2]. Then, we conduct an in-depth analysis of 27 types of prevalent security vulnerabilities involved in at least one reported security incident. In Table III, we also supplement 7 types of potential threats that present risks to the system without causing any known security incidents. Finally, we refine the seven root causes of these vulnerabilities and outline eight practical principles, as illustrated in Fig. 3.

A. At the Solidity Code Layer

1) Reentrancy: Reentrancy vulnerabilities usually stem from external calls, allowing malicious users or attackers to exploit these calls by recursively invoking a function. This enables them to repeatedly withdraw funds without affecting their balance, resulting in economic losses [50]. Attacks that exploit these reentrancy vulnerabilities are also known as recursive calling attacks.

Typical Cases: Decentralized Autonomous Organization (DAO) (2016) [51], Uniswap (2020) [46], and Lendf.Me (2020) [45].

Existing Relevant Research: As one of the most important security vulnerabilities, it has been the subject of extensive research. ReGuard (2018) [52], Echidna (2020) [53] and ReDefender (2022) [54] utilized fuzz testing techniques for the reentrancy vulnerability detection respectively. RA (2020) [55]

TABLE II
COMPARISON OF OUR WORK WITH RELEVANT STUDIES

Papers	Vulnerability					Method Principle	Tool Comparison	Real Data	Literature Statistics	Year
	Types	Principles	Cases	Relevant Research	Prevention					
Fu et al. [26]	10	√	×	×	×	√	√	×	×	2019
Xu et al. [27]	8	√	×	×	×	√	√	×	×	2020
Batina et al. [28]	21	√	×	×	×	√	√	×	×	2020
López Vivar et al. [29]	12	√	×	×	×	√	√	×	×	2020
Praitheeshan et al. [30]	16	√	×	×	×	√	√	×	×	2020
Samreen et al. [31]	8	√	√	×	√	√	√	×	×	2021
Qian et al. [32]	15	√	×	×	×	√	√	×	√	2021
Pendleton et al. [33]	40	√	√	×	√	√	√	×	×	2021
Tang et al. [34]	15	√	×	×	×	√	√	×	×	2021
Tu et al. [35]	10	√	√	×	×	√	√	×	×	2021
Chen et al. [36]	20	√	√	×	√	√	√	√	×	2022
Piantadosi et al. [37]	36	√	×	×	×	√	√	×	√	2023
Chu et al. [38]	12	√	×	×	×	√	√	√	√	2023
Haouari et al. [39]	6	√	√	×	√	√	√	×	√	2024
Our Work	34	√	√	√	√	√	√	√	√	2024

TABLE III
SUMMARY OF OTHER SMART CONTRACT VULNERABILITY TYPES WITH POTENTIAL THREAT

Layer	Vulnerabilities	Principles and Prevention Strategies
Solidity Code Layers	Hash Collision With Multiple Variable-Length Arguments	**Principle:** Abi.encodePacked() may result in hash collisions or be utilized to manipulate the result through element reordering.
		Prevention: a) Pass a single value instead of passing an array as a parameter; b) Use abi.encode() instead of abi.encodePacked().
	Type Casts	**Principle:** Insufficient type validation by the Solidity compiler.
		Prevention: a) Utilize the “new” keyword to ensure that the instances created by the callee contract are not arbitrarily modified [31]; b) Avoid unnecessary type conversions.
	Strict Balance Equality	**Principle:** When using the strict equality operator ‘==’, an attacker can send Ether to contract through a selfdestruct function, causing the contract to fail check and enter a deadlock [23].
		Prevention: Replace ‘==’ with a range, such as greater than or equal operator ‘>=’ [36], [23].
	Write to Arbitrary Storage Location	**Principle:** Smart contract data is generally stored in a designated storage location. If the attacker can write to any storage location of the contract, it can ignore the permission check and destroy the storage.
		Prevention: Avoid overwriting problems with written data structures [94].
Ethereum Virtual Machine Layer	Short Address	**Principle:** This vulnerability allows attackers to manipulate the contract address to exploit the EVM’s auto-completion attribute for obtaining unauthorized Ether transfers.
		Prevention: a) Implement robust data input validation and comprehensive address format validation; b) Recognize the significance of reasonable parameter sorting, as padding only occurs at the end [88].
	Ether Lost in Transfer	**Principle:** If an invalid address is specified during a transaction, the contract automatically registers the address instead of terminating the transaction, and causes the transferred Ether permanently lost [40].
		Prevention: Validate the accuracy of the smart contract address meticulously [2].
	Authentication Through Tx.Origin	**Principle:** Using Tx.Origin for identity verification.
		Prevention: a) Use msg.sender instead of Tx.Origin; b) Avoid returning the original address in a transaction and controlled authorization mechanism [33].

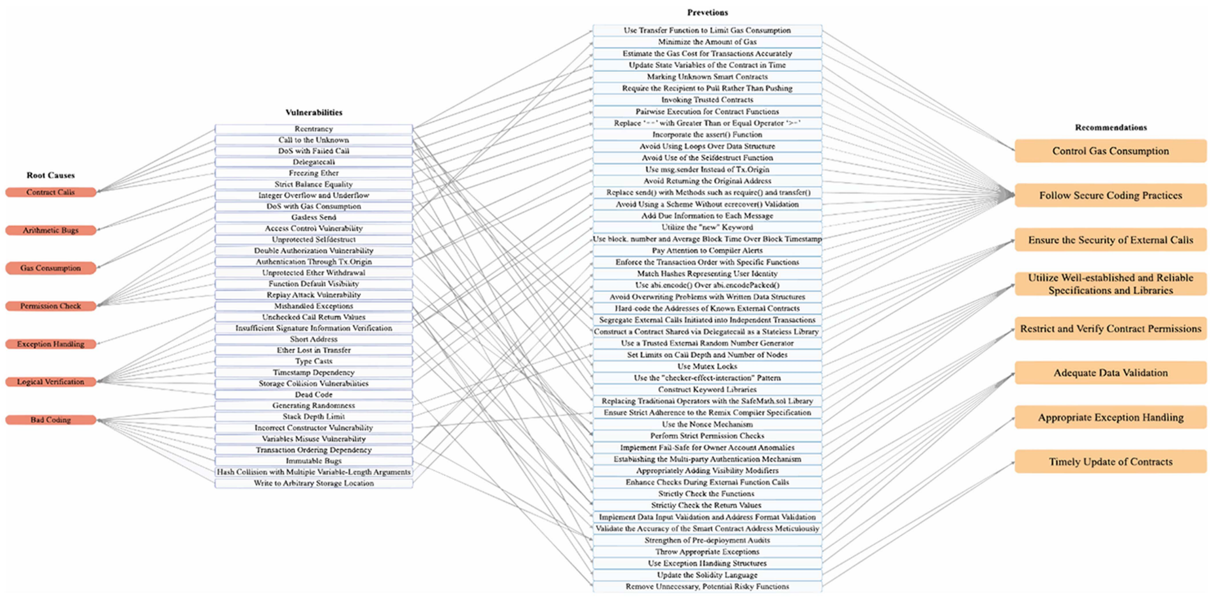
Root Causes
Contract Calls
Arithmetic Bugs
Gas Consumption
Permission Check
Exception Handling
Logical Verification
Bad Coding
Vulnerabilities
Reentrancy
Call to the Unknown
DoS with Failed Call
Delegatecall
Freezing Ether
Strict Balance Equality
Integer Overflow and Underflow
DoS with Gas Consumption
Gasless Send
Access Control Vulnerability
Unprotected Selfdestruct
Double Authorization Vulnerability
Authentication Through Tx.Origin
Unprotected Ether Withdrawal
Function Default Visibility
Replay Attack Vulnerability
Mishandled Exceptions
Unchecked Call Return Values
Insufficient Signature Information Verification
Short Address
Ether Lost in Transfer
Type Casts
Timestamp Dependency
Storage Collision Vulnerabilities
Dead Code
Generating Randomness
Stack Depth Limit
Incorrect Constructor Vulnerability
Variables Misuse Vulnerability
Transaction Ordering Dependency
Immutable Bugs
Hash Collision with Multiple Variable-Length Arguments
Write to Arbitrary Storage Location
Prevetions
Use Transfer Function to Limit Gas Consumption
Minimize the Amount of Gas
Estimate the Gas Cost for Transactions Accurately
Update State Variables of the Contract in Time
Marking Unknown Smart Contracts
Require the Recipient to Pull Rather Than Pushing
Invoking Trusted Contracts
Pairwise Execution for Contract Functions
Replace '==' with Greater Than or Equal Operator '>='
Incorporate the assert() Function
Avoid Using Loops Over Data Structure
Avoid Use of the Selfdestruct Function
Use msg.sender Instead of Tx.Origin
Avoid Returning the Original Address
Replace send() with Methods such as require() and transfer()
Avoid Using a Scheme Without ecrecover() Validation
Add Due Information to Each Message
Utilize the "new" Keyword
Use block. number and Average Block Time Over Block Timestamp
Pay Attention to Compiler Alerts
Enforce the Transaction Order with Specific Functions
Match Hashes Representing User Identity
Use abi.encode() Over abi.encodePacked()
Avoid Overwriting Problems with Written Data Structures
Hard-code the Addresses of Known External Contracts
Segregate External Calls Initiated into Independent Transactions
Construct a Contract Shared via Delegatecall as a Stateless Library
Use a Trusted External Random Number Generator
Set Limits on Call Depth and Number of Nodes
Use Mutex Locks
Use the "checker-effect-interaction" Pattern
Construct Keyword Libraries
Replacing Traditional Operators with the SafeMath.sol Library
Ensure Strict Adherence to the Remix Compiler Specification
Use the Nonce Mechanism
Perform Strict Permission Checks
Implement Fail-Safe for Owner Account Anomalies
Establishing the Multi-party Authentication Mechanism
Appropriately Adding Visibility Modifiers
Enhance Checks During External Function Calls
Strictly Check the Functions
Strictly Check the Return Values
Implement Data Input Validation and Address Format Validation
Validate the Accuracy of the Smart Contract Address Meticulously
Strengthen of Pre-deployment Audits
Throw Appropriate Exceptions
Use Exception Handling Structures
Update the Solidity Language
Remove Unnecessary, Potential Risky Functions
Recommendations
Control Gas Consumption
Follow Secure Coding Practices
Ensure the Security of External Calls
Utilize Well-established and Reliable Specifications and Libraries
Restrict and Verify Contract Permissions
Adequate Data Validation
Appropriate Exception Handling
Timely Update of Contracts

Fig. 3. Vulnerability root causes and recommended measures to prevent vulnerabilities.

used symbolic execution and equivalence checking by a satisfiability modulo theories solver to detect vulnerability. Clairvoyance (2020) [56] enabled cross-contract reentrancy vulnerability detection using static analysis methods. ReDetect (2021) [57] also used static analysis methods to detect and effectively reduce false positives. Peculiar (2021) [58], GraBit (2023) [59], ReVulDL (2022) [60] and G-Scan (2023) [61] used machine learning techniques for detection successively. Both Peculiar and GraBit improved the detection accuracy based on pre-trained models, while ReVulDL and G-Scan further attempted to locate vulnerabilities based on graphs. Additionally, Liu et al. (2021) [62] used a Manifold Pigeon Optimization Algorithm to cover the reentrant loop pathway used to detect reentrancy vulnerabilities. In addition to pre-deployment security detection methodologies for smart contracts, there are also some detection schemes proposed for post-deployment smart contracts. For instance, Sereum (2019) [63] used runtime monitoring and verification to safeguard deployed (legacy) smart contracts against reentrancy attacks. Alkhalifah et al. (2021) [64] suggested detecting and preventing reentrancy attacks during smart contract execution by calculating the disparity between the contract balance and the total balance of all participants before and after each operation. Eshghie et al. (2021) [65] proposed a monitoring framework, Dynamit, which relies only on transaction metadata and balancing data for reentrancy vulnerability detection. Cecchetti et al. (2021) [66] introduced the Secure-Reentrancy Information Flow Calculus (SeRIF) to enhance reentrancy security by combining static and dynamic locking mechanisms. Finally, Zheng et al. (2023) [67] evaluated the performance of existing reentrancy vulnerability detection methods and advocated for exploring new reentrancy paradigms.

Prevention Strategies: a) Use the Checks Effects Interactions pattern [68] to ensure that the state change logic is committed before sending gas to external smart contracts; b) Use mutex locks to prevent attackers from making recursive calls to the withdraw function; c) Use transfer function to limit gas consumption under 2300 [36]; d) Update state variables of the contract in time before calling another contract [33].

2) Denial of Service (DoS) with Failed Call: It typically occurs when an attacker disrupts smart contract interactions by exploiting unexpected failures or intentionally corrupting external smart contract calls.

Typical Cases: King of the Ether Throne (KotET) (2016) [69].

Existing Relevant Research: SmartScan (2021) [70] combined static and dynamic analysis to detect DoS vulnerabilities in Ethereum smart contracts caused by unexpected revert.

Prevention Strategies: a) Segregate external calls initiated by the callee contract into independent transactions [70]; b) Require the recipient to initiate the withdrawal of funds rather than sending them out using push [31].

3) DoS With Gas Consumption: If an attacker manipulates the gas consumption to reach the upper limit, the transaction transfer operation will fail.

Typical Cases: GovernMental (2016) [31], Fomo3D (2018), FishPool (2021) [40].

Existing Relevant Research: Chen et al. (2020) [71] introduced a simulation-based framework and a gas cost mechanism to automatically measure resource consumption and dynamically adjust the cost to mitigate DoS attacks based on the number of EVM operations. MadMax (2020) [72] used static program analysis to detect DoS attacks caused by gas restrictions. Li et al. (2021) [73] studied gas estimation and optimization from the perspective of loop functions. Besides, Gas Gauge (2022) [74] combined static analysis and runtime verification to automatically detect DoS vulnerabilities due to gas limit issues in Ethereum contracts. Ghaleb et al. (2022) [75] used taint tracking to detect gas-related vulnerabilities.

Prevention Strategies: a) Avoid using loops over data structures in smart contracts [33]; b) Implement a fail-safe for owner account anomalies by setting the owner to a multi-signature contract or using a timelock that requires the user to call after a specified period [88].

4) Integer Overflow and Underflow: It usually occurs when the result of an operation exceeds the maximum value (overflow) or falls below the minimum value (underflow) for a particular numeric data type [36]. The discrepancy in the number of available integer types between Solidity and EVM, along with their non-strict mapping relationship, contributes to this vulnerability [32]. If it is not detected promptly, this vulnerability may not only cause smart contract execution errors but also be exploited by attackers to perform illegal operations, such as circumventing checks and tampering with crucial data.

Typical Cases: BeautyChain (2018) [45], SmartMesh Token (2018), BAI (2018) [76], YAM (2020) [47], Cover Protocol (2020) [47], and Poolz Finance (2023) [48].

Existing Relevant Research: Osiris (2018) [77] combined symbolic execution and taint analysis to detect integer bugs. EASYFLOW (2019) [78] used taint analysis-based tracking techniques to capture manifested overflows. Lai et al. (2020) [79] identified 11 features of integer overflow vulnerabilities and abstracted them into three corresponding XPath patterns. VERISMART (2020) [80] proposed a new CEGIS-style algorithm that can automatically detect arithmetic bugs using transaction invariance. Sun et al. (2020) [81] utilized mutation testing to detect integer overflows and introduced five distinct mutation operators to detect.

Prevention Strategies: a) Use SafeMath.sol libraries, which effectively mitigate the risks of integer overflows without any modifications to the Solidity compiler or EVM [82]; b) Incorporate the *assert()* function to prevent vulnerabilities by conducting rigorous validation checks and terminating the contract if any of the checks fail.

5) Unchecked Call Return Values: When using functions like *send, call, delegatecall* and *callcode* for smart contract invocations, it is essential to check their return value rigorously. Neglecting this step may potentially expose unexpected security problems [33].

Typical Cases: GovernMental (2016) [31], KotET (2016) [69], and SushiSwap (2023) [49].

Prevention Strategies: a) Replace *send()* with methods like *require()* and *transfer()*; b) Checking their return values thoroughly is also critical.

6) Mishandled Exception: On Ethereum, a smart contract will automatically throw an exception in abnormal situations. In such cases, the callee contract will terminate, restore its state and return an error. However, if the exception handling is not properly implemented, exceptions in the callee contract may not be propagated to the caller contract as intended. Then the exception information can not be appropriately accessed by the caller contract, resulting in a security vulnerability [83].

Typical Cases: GovernMental (2016) [31], Opyn (2020) [41], Furucombo (2021) [46], and Force DAO (2021) [45].

Prevention Strategies: a) Use exception handling constructs and carefully check the return values of functions, throwing exceptions where necessary; b) When invoking underlying functions, it is essential to rigorously examine the return value and adopt a consistent approach to handle exceptions [32]; c) Update the Solidity language to ensure a uniform handling of exceptions can also serve as a viable solution [31].

7) Function Default Visibility: In Solidity, there are four visibility descriptors, namely external, public, internal and private [68]. The default visibility descriptor for functions is usually public, which means that they can be internally and partially externally invoked. If this function involves important sensitive information, this may pose a security risk.

Typical Cases: VETH (2020) [43] and Bancor (2020) [40].

Prevention Strategies: Appropriately add visibility modifiers to control permissions effectively.

8) Incorrect Constructor Vulnerability: The constructor of smart contracts plays a crucial role in initializing and binding them to specific owner addresses. If the constructor is declared improperly, it may grant attackers complete access to the smart contract.

Typical Cases: MorphToken (2018).

Prevention Strategies: Ensure constructors adhere strictly to the Remix compiler specification.

9) Access Control Vulnerability: It primarily arises from improper code implementations or inadequate permission checks, allowing attackers to gain unauthorized access and compromise the system's functionality and data. For example, the absence of access permission declarations for certain essential functions can allow attackers to exploit this vulnerability and disrupt the execution of the smart contract.

Typical Cases: Parity Wallet (2017) [84], Soda (2020) [75], and DeFi Saver (2020) [41].

Existing Relevant Research: SPCon (2022) [85] extracted role structures from historical transactions and optimized these structures by introducing a novel partial-observation role mining (PORM) technique. AChecker (2023) [86] detected access control vulnerabilities in smart contracts by analyzing data dependencies and used symbolic execution to distinguish between intended behavior and actual vulnerabilities. SoMo (2023) [87] utilized a modifier dependency graph structure, and iteratively explored the feasibility of slicing paths and symbolic test paths while using natural language processing to analyze the use of modifiers.

Prevention Strategies: Perform strict permission checks for sensitive operations, such as transfer operations.

10) Call to the Unknown: Signatures, comprised of function names and parameter types, serve as a unique identifier for functions in smart contracts. When one smart contract calls a function in another smart contract, if the given signature does not match the called smart contract, certain primitives in Solidity (i.e., *call, send,* and *delegatecall*) will invoke the fallback function instead of throwing an exception. If the fallback function is manipulated by an attacker, it could potentially lead to security issues [2].

Typical Cases: DAO (2016) [2] and brahTOPG (2022) [41].

Prevention Strategies: a) Enhance vigilance and strengthen checks during external function calls; b) Mark unknown smart contracts as potentially unsafe and take appropriate precautions when interacting with them; c) Hard-code the addresses of known external smart contracts [88]; d) Build a keyword library to prevent attackers from manipulating the smart contract state and launching attacks [31].

11) Freezing Ether: This vulnerability refers to a scenario where a smart contract account has no private keys and can only manage digital assets through the execution of smart contract code. Some smart contracts do not possess their transfer functions but instead rely on calling the transfer functions implemented in external smart contracts [32]. This situation can arise vulnerabilities as follows: a) Contracts have receive functions but lack send functions, leading to permanently frozen ETH assets [89]; b) When a contract implements the transfer function by invoking an external transfer function, any failure in the execution of the transfer function can lead to the freezing of all Ether associated with the contract. Additionally, contracts relying on others or libraries may encounter failures due to unexpected selfdestruct commands or unintended behaviors [18].

Typical Cases: Parity Wallet (2017), Hegic (2020) [41], and Akutars (2022) [41].

Prevention Strategies: a) Strictly verify the integrity of the functions in the contract, it is possible to remove the separate receive function [36]; b) It is recommended to use pairwise execution of receive and send functions in the contract.

12) Unprotected Selfdestruct: The selfdestruct instruction can potentially release the Ether held within a contract, effectively rendering the contract invalid. If a smart contract lacks sufficient authentication measures, unauthorized attackers may invoke the self-destruct function to destroy the smart contract [33]. In addition, if attackers have illegally gained the ownership of a smart contract, they can also initiate self-destruction by exploiting the kill function [18].

Typical Cases: Parity Wallet (2017).

Existing Relevant Research: LifeScope (2022) [90] reduced contract self-destruction by identifying Unmatched ERC20 Token, Confusing Contracts and Limits of Permission.

Prevention Strategies: a) Avoid the use of the selfdestruct function; b) Establish selfdestruct guards, which ensure that only authorized users can access the instruction [23]; c) Establish a multi-party authentication mechanism to increase the threshold for instruction execution.

13) Gasless Send: In Ethereum smart contracts, the maximum gas limit for the fallback function is set to 2300 units. If the fallback function in its callee smart contract involves complex operations that consume a significant of gas, the caller may encounter an out-of-gas exception. If the caller does not respond to this exception promptly and correctly, attackers can exploit the vulnerability for their own profit.

Typical Cases: KotET (2016) [69] and YAM (2020) [47].

Existing Relevant Research: Prechtel et al. (2019) [91] researched the gasless send vulnerability and enhanced Mythril with additional analysis modules to detect this issue.

Prevention Strategies: a) Throw an appropriate exception when a failure occurs due to gas exhaustion; b) Minimize the amount of gas or reduce the cost of enforcing the contract in the design of the fallback function to reduce the chances of failure; c) Estimate the gas cost for transactions accurately, especially for contract looping functions [73].

14) Delegatecall: The delegatecall opcode allows one smart contract to call another and executes the code of the called contract in the context of the caller contract. This working mode determines that a malicious callee contract may update (or manipulate) the state variable of the caller contract to trigger unexpected executions, like self-destruction or loss of contract balance.

Typical Cases: Parity Wallet (2017) and Furucombo (2021) [46].

Prevention Strategies: a) Invoke trusted contracts to ensure the safety of caller contracts; b) Construct a smart contract shared via delegatecall as a stateless library to prevent malicious changes to the smart contract state during delegatecall operation [33].

15) Insufficient Signature Information Verification: A sender can transfer funds to multiple recipients using a proxy contract. During the transaction, the proxy contract verifies the validity of the relevant digital signature (e.g., nonce, proxy contract address). If the signature does not provide the necessary information, a malevolent recipient can replay the message several times to acquire extra payments.

Typical Cases: El Dorado Exchange (EDE) [92].

Existing Relevant Research: SIGUARD [93] built control flow graphs (CFGs) based on the bytecode of smart contracts, explored signature-related execution paths and performed dynamic taint analysis to detect potential vulnerabilities.

Prevention Strategies: a) This vulnerability could be mitigated by adding due information (e.g., nonce value and timestamp) to each message [33]; b) It is recommended to avoid other authentication methods that do not involve a proper signature verification process using *ecrecover()* [94].

16) Double Authorization Vulnerability: While multiple authorization mechanisms in contracts can increase operational flexibility, they also introduce certain risks. The platform or contract might misuse these authorizations, such as allowing attackers to illegally withdraw user funds.

Typical Cases: Degen.Money (2020) [43].

Prevention Strategies: Careful authorization and timely revocation of unnecessary authorizations.

17) Variables Misuse Vulnerability: Improper use of variables in the contract can lead to data inconsistency, enabling attackers to exploit the function for unlimited issuance.

Typical Cases: Cover Protocol (2020) [43].

Prevention Strategies: Strict adherence to smart contract writing standards and specifications.

18) Unprotected Ether Withdrawal: If the smart contract lacks proper access control and fails to adequately verify user permissions, a malicious actor can withdraw Ether from the contract without restriction.

Typical Cases: Rubixi (2016) [2].

Prevention Strategies: Strictly control the permissions and trigger conditions of the withdrawal operation [94].

19) Dead Code: If unnecessary functions or code remain in the smart contract due to early design or review errors, attackers may exploit them to carry out an attack.

Typical Cases: Ethereum Alarm Clock (2022).

Prevention Strategies: Remove unnecessary, potential risky functions.

20) Storage Collision Vulnerabilities: When storage slots are accidentally shared between contracts, this may lead to unexpected behavior due to an inconsistent understanding of the data types and uses of their shared storage.

Typical Cases. AUDIUS (2022) [95].

Existing Relevant Research: CRUSH [95] used symbolic execution and program slicing techniques to identify vulnerabilities and generate a proof-of-concept exploit to confirm their existence.

Prevention Strategies: Pay attention to compiler alerts during compilation in a timely manner [95].

21) Uninitialized Storage Pointer (Fixed): In Solidity, composite local variables such as structs, arrays, and mappings are stored in unused storage slots by default and may be mistaken as part of storage by the compiler. If they are not properly initialized, attackers could exploit this vulnerability to unlawfully overwrite specific state variables, causing data loss or disrupting smart contract execution. Fortunately, this defect has been resolved with the Solidity version update [33].

Typical Cases: OpenAddressLottery [96].

B. At the Ethereum Virtual Machine Layer

1) Immutable Bugs: The unique immutable nature of smart contracts makes it challenging to modify them once deployed, unlike traditional programs. Hence, attackers capitalize on this unique attribute to identify weaknesses in smart contracts and formulate their attack strategies accordingly [2].

Typical Cases: Rubixi (2016) [2].

Prevention Strategies: Prioritize a robust auditing process before deployment to effectively deal with potential threats.

2) Stack Depth Limit: During the invocation process of a smart contract, each invocation adds a layer to the relevant call stack. Once the stack reaches its limit of 1024 layers, any subsequent calls will throw an exception. Attackers can exploit this by creating a sequence of nested calls to approach the stack depth limit and then invoking the victim smart contract. As a result, any further calls made to the victim smart contract will fail. This vulnerability has been addressed by the hard fork of EIP-150, which introduces new gas consumption rules for external calls [33].

Typical Cases: GovernMental (2016) [31].

Prevention Strategies: It is advisable to set limits on the call depth and number of nodes during the deployment and operation of smart contracts.

C. At the Blockchain System Layer

1) Timestamp Dependency: It usually refers to a smart contract that adopts block timestamps as trigger conditions for certain important operations or as a source of randomness, which can be manipulated by an attacker. For instance, if a smart contract relies on a timestamp-based condition to decide whether to transfer funds, a malicious miner can manipulate the timestamp slightly to fulfill the condition and consequently benefit the attacker [33].

Typical Cases: GovernMental (2016) [31].

Existing Relevant Research: Scruple (2023) [97] obtained potential data propagation paths for timestamp vulnerabilities and learned from pre-trained models to detect the vulnerabilities.

Prevention Strategies: a) It is advisable to refrain from relying on block timestamps as a deterministic factor for altering critical smart contract states [33]; b) For time-sensitive logic, it is advisable to estimate time using *block. number* and average block time rather than block timestamps, as miners cannot easily manipulate block numbers. It may be safer to specify a block number to modify the state of the smart contract [88].

2) Transaction Ordering Dependency: If a block includes two transactions that invoke the same contract, the final state of the smart contract may be influenced by the order of the transactions. For instance, attackers can exploit this vulnerability to carry out specific attacks by monitoring transactions in the transaction pool. If an attacker is a miner, they can manipulate the gasPrice of their transactions or collaborate with other miners to prioritize their transactions.

Typical Cases: GovernMental (2016) [31] and Balancer (2018) [41].

Existing Relevant Research: Natoli et al. [98] used Ethereum-based functions such as *SendIfReceived* to enforce transaction orders. Bose et al. [99] proposed an extensible system, SAILFISH, which combines the storage dependency graph (SDG) with symbolic execution to automatically find errors in the inconsistency of states in smart contracts.

Prevention Strategies: a) Enforce the transaction order with specific functions [98]; b) During the transaction, a commit reveals that a hash scheme can be used to avoid this vulnerability by matching hashes representing the user's identity [94].

3) Generating Randomness: A common solution is to incorporate blockchain-related parameters, such as timestamps, to build functions for generating pseudo-random numbers. Malicious miners can exploit this functionality to generate biased and pseudo-random blocks to gain an unfair profit [2].

Typical Cases: Fomo3D (2018).

Existing Relevant Research: Qian et al. (2023) [100] examined common pseudo-random number generation techniques and summarized corresponding four types of attacks. They proposed a tool, named RNVulDet, to identify random number vulnerabilities in smart contracts.

Prevention Strategies: Use a trusted external random number generator to ensure that the source of the random numbers does not depend on block variables.

4) Replay Attack Vulnerability: A replay attack occurs when an attacker uses captured authentication information to fraudulently delay or retransmit it, leading to unauthorized fund transfers.

Typical Cases: bZx (2020).

Prevention Strategies: Use the nonce mechanism to guarantee the uniqueness of a contract transaction.

We provide seven other types of vulnerabilities as a supplement in Table III, which pose potential threats to contract security and may lead to security incidents in the future. It is crucial to comprehend the underlying principles, implement preventive measures, to proactively mitigate the related risks. As shown in Fig. 3, based on our in-depth investigation into the root causes of these vulnerabilities, and drawing inspiration from [102], [103], we formulate a set of development recommendations for smart contract developers. Secure programming practices are an effective way to prevent smart contract vulnerabilities.

III. Smart Contract Security Analysis and Detection

Existing efforts for analyzing and detecting security issues in smart contracts are roughly categorized into five types: formal verification, fuzz testing, machine learning, program analysis, and others. We have also examined current repair methods.

A. Existing Detection Methods

1) Formal Verification: Formal verification methods rely primarily on mathematical modeling and reasoning [104]. These methods usually involve transforming a smart contract into a rigorous mathematical model using principles of mathematical logic. Then proofs and theorems are used to verify whether the smart contract complies with a particular property or specification [187].

Theorem Proving is a formal way to provide proofs in symbolic logic by deductive inference. It specifies the code of a smart contract as an abstract mathematical model, introduces axioms and premises, and then uses predicate logic to deduce whether the contract satisfies certain properties and attributes [75], [76]. The *KEVM framework* [105] establishes executable formal semantics for EVM bytecode in the K-framework, passing official test suites and providing a corresponding formal verification tool. *Isabelle/HOL* [106] enhances a current formalization of EVM by introducing a sound program logic specifically designed for bytecode. It organizes bytecode sequences into blocks of straight-line code and establishes a program logic that enables reasoning about these blocks. *VaaS* [40] is designed to automatically identify common security vulnerabilities and functional logic flaws in contracts. It also identifies the exact positions of high-risk codes and offers suggestions for necessary modifications. *VerX* [107] focuses on verifying valid external callback freedom contracts, using a unique blend of abstraction and symbolic execution to achieve precise transaction verification.

Model Checking employs a formal model of a finite set of states that can simulate the interaction of contract, and analyses and verifies all the states in the generated state space [26], [24]. *FsolidM* [108] is a formal model based on rigorous semantics. It offers a graphical editor for designing smart contracts as Finite State Machines (FSMs) and an automatic code generator to translate them into Solidity code. *ZEUS* [109] utilizes abstract interpretation, symbolic model checking and CHCs to quickly verify contracts. *VeriSolid* [110] extends FsolidM by integrating formal verification capabilities. It enables the generation of Solidity code from the verified models, facilitating the development of smart contracts. *Duo* et al. [111] applied Hoare's condition to create a Coloured Petri net (CPN) model, which improves the program logic rules of bytecode, shows the complete state space and error execution paths, and achieves multi-perspective analysis.

Symbolic execution is one of the techniques based on formal verification [22]. It utilizes symbols to represent variables in a program [112]. Any operation involving these symbols generates a symbolic expression and passes it on. By simulating the execution of program instructions and gathering constraints on branch paths, it periodically uses a constraint solver to verify whether the current path constraints can be satisfied simultaneously. When symbolic execution matches code patterns corresponding to potential vulnerabilities, those vulnerabilities will be reported. *Oyente* [83] utilizes symbolic execution to navigate through feasible execution paths on CFGs and works with EVM bytecode to discover security bugs in smart contracts. *Mythril* [113], following a methodology similar to Oyente, is designed to detect common vulnerabilities using symbolic execution, such as integer underflows. However, it should be noted that it cannot identify issues within an application's business logic. *Securify* [114] encodes the dependence graph of smart contracts and employs a datalog solver to verify smart contract behaviors. *Annotary* [115] is a concolic execution framework that integrates symbolic execution with Ethereum blockchain data parsing through source code annotations to analyze vulnerabilities in smart contracts. *TEETHER* [116] focuses on identifying key paths that lead to critical instruction and then transforms them into sets of constraints using symbolic execution to deduce target transactions. *SMARTEST* [117] utilizes a trained statistical model that leverages known vulnerable transaction sequences to guide the process of symbolic execution. *DefectChecker* [118] employs various rules based on the CFG and stack events to identify multiple types of defects present in smart contracts effectively. *Pluto* [119] analyses semantic information using Inter-Contract CFG (ICFG) and Inter-Contract Path Constraints (ICPC) to detect vulnerabilities in inter-contractual scenarios and performs path reachability checking based on predefined rules. *Park* [120] utilizes the Parallel-Fork symbolic execution technique for vulnerability detection, which relies on the power of multiprocessing and CPU cores to improve efficiency. *SAILFISH* [99] divides the detection process into two phases (i.e., EXPLORE and REFINE). EXPLORE focuses on lightweight analysis by transforming a smart contract into a storage dependency graph (SDG). REFINE uses symbolic evaluation to achieve accurate detection of vulnerabilities. *ExGen* [121] is a cross-platform automatic analysis framework that converts Ethereum and EOS contracts into intermediate representations. It generates Partial-ordered Transactional Set (PTS) and executes symbolic attack contracts to extract and solve all constraints. *Maian* [122] identifies vulnerabilities in long sequences of smart contract invocations, characterizes them as properties of the entire execution trace, and employs inter-procedural symbolic analysis and a verifier to uncover actual vulnerabilities. *Manticore* [123] generates code assessment reports to detect bugs and verify code invariants within smart contracts by monitoring the state of individual contracts and also the inter-contract state. *ETHBMC* [124] leverages the precise reasoning of cryptographic hash functions. It allows users to explore predefined models based on smart contract code and automatically generate concrete inputs for further analysis. *NFTGuard* [125] dynamically constructs CFGs based on symbolic execution and records key events to detect defects in non-fungible tokens smart contracts.

Our Observation: After a thorough review of existing research, it can be found that: 1) formal verification-based detection tools usually suffer from low automation, and existing research struggles to balance full automation with the accessible program path problem [30]; 2) formal verification relies heavily on human experience. In the future, specific validation specifications can be developed for different detection targets to enhance the coverage of reachable paths, thereby improving the detection accuracy for complex smart contracts [20].

2) Fuzz Testing: Fuzz testing is a prominent software testing technique that involves the automatic and rapid generation of diverse test inputs. These inputs are then systematically executed against a target program to detect and identify hidden vulnerabilities and bugs [126]. In recent years, fuzz testing has been integrated with other technologies such as symbolic execution, taint analysis, static analysis, and machine learning to enhance its performance in detecting vulnerabilities in smart contracts. Based on the generation of test cases, fuzz testings can be categorized into generation and mutation based ones [127].

Generation-based Fuzzing generates test cases using a formal input format specification, such as Application Binary Interface (ABI), models, and grammar [127], [33]. *ILF* [126] is designed to learn an effective and fast fuzzer from symbolic execution for generating transactions and detecting vulnerabilities in smart contracts. To address potential unknown vulnerabilities. *SoliAudit* [128] designs a dynamic fuzzer for abnormal analysis. *Ethploit* [129], a fuzzing-based exploit generator, can be utilized to uncover vulnerable smart contracts. It incorporates a static taint analysis methodology to guide the generation of transaction sequences. It also uses a dynamic seed strategy to pass hard constraints and leverages an instrumented EVM to simulate blockchain effects, ensuring efficient and accurate testing. *xFuzz* [130] uses machine learning models to filter paths and guide fuzz testing, greatly reducing the search space and focusing on detecting cross-contract vulnerabilities. *RLF* [131] utilizes reinforcement learning to detect sophisticated vulnerabilities in smart contracts, modeling fuzzing as a Markov decision process. It introduces a reward mechanism that balances vulnerabilities and code coverage, guiding the fuzzer to generate transaction sequences that expose vulnerabilities.

Mutation-based Fuzzing generates test inputs by randomly modifying or mutating existing valid inputs to assess the target system for vulnerabilities or unexpected behavior [127]. *ContractFuzzer* [89] includes an offline EVM instrumentation tool for monitoring smart contract execution and an online fuzzing tool that generates inputs from ABI specifications to detect vulnerabilities. *sFuzz* [132] integrates the AFL fuzzer strategy with a lightweight multi-objective adaptive approach specifically designed for challenging branch coverage scenarios. *Harvey* [133] is an advanced grey-box vulnerability detection tool that randomly mutates program inputs to explore new paths in smart contracts. It focuses on targeted, demand-driven fuzzing of transaction sequences to thoroughly analyze critical areas. *SMARTIAN* [134], a combined static and dynamic analysis approach for fuzzing smart contracts to address the limitations of existing tools in detecting critical transaction sequences, as smart contracts uniquely share internal states across transactions. *ConFuzzius* [135] generates meaningful inputs by merging evolutionary fuzzing and constraint solving to enhance code coverage. By employing dynamic data dependency analysis across state variables, it generates sequences of transactions with implicit vulnerabilities. It further models both block-related and contract-related information as fuzzable inputs to effectively address environment dependencies. *IR-Fuzz* [136] generates function invocation sequences based on data dependencies, pushing the fuzz tester to trigger more complex states and improving the chances of detecting vulnerabilities. *ItyFuzz* [137], a Snapshot-based fuzzer, integrates multiple customized waypoint mechanisms to efficiently categorize and store interesting states. *EF↯CF* [138], a high-performance fuzzer for Ethereum smart contracts, boosts fuzzing efficiency by using a structure-aware mutation engine for transaction sequences and leveraging the contract's ABI to generate valid transaction inputs.

Our Observation: After investigating the above testing tools, it can be found that: 1) test cases generated by fuzz testing may not achieve the expected path coverage due to limited runtime information and semantic understanding [32]; 2) the dependency of smart contracts on the current state and the external environment affects the detection accuracy of fuzz testing [135]; and 3) along with the increment in the complexity of the contract, the path explosion problem may arise during the testing process [21]. Some studies have combined other security analysis methods with fuzz testing to optimize test case generation and guide more effective path exploration. State-aware fuzz testing is also an emerging research focus.

3) Machine Learning: Machine learning-based approaches involve extracting features from various representations of smart contracts, during the data preprocessing stage. These extracted features are subsequently fed into various machine learning algorithms to train vulnerability detection models. They have shown remarkable proficiency in detecting vulnerabilities in smart contracts. Deep learning, as a particular branch of machine learning, has experienced rapid development in this field. Thus, we categorize machine learning-based methods into traditional and deep learning based approaches.

Traditional Machine Learning-based methods typically involve utilizing classical machine learning algorithms, such as ensemble learning, to detect vulnerabilities in smart contracts. *ContractWard* [139] extracts bigram features from simplified opcodes of smart contracts and combines SMOTETomek sampling with the XGBoost integrated learning to detect six vulnerability types. *SCSCAN* [140] is a Support Vector Machine (SVM) based vulnerability detection system. *Eth2Vec* [141] analyses bytecode through a neural network to automatically learn the inherent syntax and semantics of each contract, it identifies function-level vulnerabilities by comparing similarities, even if the code structure has changed.

Deep Learning-based methods: leveraging techniques like various deep neural networks and natural language processing excel at automatically extracting meaningful features from diverse raw data to facilitate effective vulnerability detection [20]. We roughly categorize these works into sequence-based, graph-based, and hybrid methods.

Sequence-based methods: *SmartEmbed* [142] employs a code embedding learning method to encode code elements and bug patterns into numerical vectors. Then, it utilizes a similarity checking technique to detect code clones and bugs associated with clones. *ESCORT* [143] learns the generic bytecode semantics of smart contracts through a generic feature extractor and uses separate branches to learn specific features for each vulnerability type. It easily scales to detect new vulnerability types even with limited data by integrating transfer learning. *DeeSCVHunter* [144] introduces the concept of Vulnerability Candidate Slice (VCS) and incorporates code embedding techniques to enhance the detection performance. *MODNN* [145] introduces the concept of a crucial operation sequence (COS) to reduce the dimensionality of embedded data. It then uses pre-trained BERT to convert COS into eigenvectors for Multi-Objective Detection (MOD) algorithms, enabling parallel vulnerability identification and providing the probability of each vulnerability. *SmartConDetect* [146] extracts code fragments from contracts and leverages a pre-trained BERT model to further detect vulnerable code patterns. *DLVA* [147] is a deep learning-based vulnerability analyzer by transforming contract bytecode into high-dimensional floating-point vectors. The detection process involves two steps, namely Sibling Detector (SD) using Euclidian-nearby and Core Classifier (CC) using neural networks. *Blass* [148] enhances characterization by creating program slices with complete semantic information and converting them into code chains. It also uses a custom neural network, Bi-LSTM-Att. *VulHunter* [149] is an innovative ML-assisted detection method designed for analyzing Ethereum smart contracts without manual rules. It uses the multi-instance learning mechanism to handle coarse labels and employs a custom Bi^2-LSTM model to capture subtle features, effectively identifying vulnerable instances. *DL4SC* [150], a novel deep learning framework for opcode-level vulnerability detection in smart contracts, combines a Transformer encoder and CNN. It uses Sparrow Search Algorithm (SSA) to optimize model hyperparameters, enhancing detection accuracy and efficiency. *GPTScan* [151] innovatively combines a Generative Pre-training

Transformer (GPT) with static analysis for effective logic vulnerability detection, using GPT to identify vulnerable functions and distinguish key variables and statements. *PSCVFinder* [152] utilizes a slicing methodology to eliminate irrelevant code and employs large language models with prompt-tuning to detect reentrancy and timestamp dependency vulnerabilities.

Graph-based methods: DR-GCN and TMP [153] are Graph Neural Networks (GNNs) based methods for detecting vulnerability in smart contracts. A normalized contract graph is constructed to capture the syntactic and semantic information of a smart contract function. *EtherGIS* [154] utilizes GNNs to extract global feature representation from CFGs of smart contracts for vulnerability detection. It uses expert knowledge to analyze the bytecodes of different vulnerabilities and define EVM instructions that may trigger attacks. *MANDO-GURU* [155] combines CFGs and call graphs (CGs) to capture the semantics of smart contract code, and employs specialized metapaths and heterogeneous attention graph neural networks to learn embeddings at various granularity levels to detect vulnerabilities in smart contracts. *MRN-GCN* [156] employs an edge-enhanced graph convolution network and a self-attention mechanism for fine-grained vulnerability detection, pinpointing which specific function in the smart contract is vulnerable. *GraphSA* [157] combines GNNs and static analysis for smart contract vulnerability detection using a contract tree, where nodes represent crucial opcode blocks and edges represent control flow between them. *SCVHunter* [158] constructs a heterogeneous semantic graph from intermediate representations and detects vulnerabilities using a heterogeneous graph attention network.

Hybrid methods: VulnSense [159], a multimodal learning-based detection method, combines BERT, BiLSTM, and GNN, using source code, opcodes, and CFGs as features, to overcome the limitations of single-feature or single-model deep learning methods. *VDM-AEI* [160] is a vulnerability detection method that uses grayscale images, contract graphs, and expert-defined feature vectors from smart contracts. These features are input into VGG16, GRU, AutoInt, and DCN networks to learn more meaningful features and enhance vulnerability detection performance.

Our Observation. From the above survey, we can notice that machine learning-based detection methods, especially those using deep learning, have surged in recent years due to their high automation and efficiency. However, it is essential to recognize that, in addition to the elegance of deep learning, their detection effectiveness also relies on the choice of feature representation methods and the availability of high-quality labeled samples. Enhancing feature representation and reducing reliance on high-quality labeled samples through unsupervised learning techniques may require additional research. Besides, most existing learning-based detection schemes focus on binary or multivariate classification to identify the presence or the types of vulnerabilities. Enhancing the interpretability of the learning-based detection model to increase user confidence is equally important. Finally, the security of the model itself must not be overlooked.

4) Program Analysis: Typically, program analysis methods construct a model represented by abstract data structures, extract features such as an intermediate representation (IR), and input the contract's features and specifications into an analyzer for inspection [21], [150].

An Abstract interpretation method: is often used to verify the correctness of a program by approximating and abstracting its semantics while ignoring specific instructions [12], [15]. *Vandal* [161] converts bytecode into higher-level intermediate representations through abstract interpretation and uses a logic-driven approach to detect multiple vulnerabilities. *MadMax* [72] utilizes a smart contract decompiler and semantic queries in datalog to automatically detect gas-focused vulnerabilities at the EVM bytecode level. *eThor* [162] abstracts the semantics of bytecode as a set of Horn clauses and employs reachability analysis to acquire both security properties and functional properties of smart contracts.

An Intermediate Representation (IR) method involves transforming smart contract source code or bytecode into an intermediary structured representation format. It acts as an intermediary between the original program and vulnerability detection tools, facilitating the detection process. *SASC* [163] is a static security assurance method for detecting potential security risks in smart contract source code. It has two main functions: topological analysis of contract invocation relationships to help developers understand code structure, and logic risk detection and localization using symbolic execution and syntax analysis to identify and locate six types of risks. *SmartCheck* [164] works by compiling the contract source code into an XML-based intermediate representation and checks against the XPath schema. It enables the identification of potential vulnerabilities in the code. *Slither* [165] leverages the intermediate representation SlithIR to provide in-depth information about smart contracts. It not only facilitates automated vulnerability detection but also enables automated optimization detection, code understanding, and assisted code review. *SolAnalyser* [166] is a vulnerability detection tool using both static and dynamic analysis. It follows a three-phase approach: assertion injection, input generation, and execution and analysis of instrumented contracts. Additionally, it includes the MuContract tool to assess the effectiveness of existing vulnerability detection tools. *TXSPECTOR* [167] incorporates a trace extractor to execute Ethereum transactions and generate bytecode-level traces, which are used to construct Execution Flow Graphs (EFGs). The EFGs are then traversed by a Logic Relation Builder to extract data and control dependencies, expressed as logic relations. Additionally, it provides an attack detector that enables users to define their own detection rules for uncovering different types of attacks in transactions. *NeuCheck* [168] is a smart contract analysis tool that utilizes the ANTLR parser to introduce a syntax tree in the syntactical analyzer, enabling the transformation of source code into an intermediate representation. It subsequently leverages dom4j to parse this tree and provides users with security reports, aiding in the identification of potential security vulnerabilities. *SmartDagger* [169] focuses on analyzing the intricate contextual information involved in cross-contract invocations. It does not perform typical whole-program analysis, but selectively analyzes a subset of functions and reuses the results of data flow, which significantly improves detection efficiency.

SmartFast [170] detects and locates vulnerable code by representing contract source code as a new intermediate representation named SmartIR, combined with pre-defined rules and taint tracking techniques. *SmartGraph* [171] uses graphs as an IR to capture the structure and dependencies of code, enabling analysis at a higher level of abstraction. *SmartState* [172] integrates control flow, data flow, assertion-related state dependency, and temporal-order state dependency to build a State Dependency Graph (SDG) for detecting state-related vulnerabilities. *SoliDetector* [173] detects and locates vulnerabilities by building a knowledge graph and defining defect patterns to overcome Solidity's multiple version compatibility and detection time constraints.

Our Observation: It can be found that procedural analysis methods mainly rely on fixed semantic rules, and inadequate or inappropriate rules can easily lead to false positives. Hence, forthcoming studies should focus on developing a more flexible Intermediate Representation (IR) that can proficiently manage diverse vulnerabilities and multiple iterations of smart contracts.

5) Other Security Detection Methods: Taint Analysis is commonly used to designate data flows from external sources as tainted [26]. It aids in uncovering potential security vulnerabilities in programs, as well as detecting and mitigating the risk of attacks such as injection attacks or data leaks. *Ethainter* [174] employs the notion of information flow to monitor contaminated values and expands it to a key domain concept model, tailored to detect complex attacks that involve the enhancement of tainted information across various transactions. Other detection tools such as Mythril also utilize taint analysis assistance techniques to enhance the accuracy of the analysis.

Online Detection tools strive to identify attacks targeting smart contracts or safeguard them from attacks post-deployment. *SODA* [175] utilizes a layered structure to enable comprehensive implementation information acquisition and online attack detection for smart contracts on blockchain platforms. Similarly, there are some works used to protect and verify the security of deployed smart contracts. *ContractGuard* [176] is an intrusion detection system (IDS) that actively monitors anomalous control flow paths in smart contracts. *VULTRON* [177] is designed to accurately differentiate abnormal transactions from normal ones by identifying mismatches between the transferred amount and the expected amount. *FSFC* [178] utilizes vulnerability location tools and generates filter rules to effectively prohibit malicious inputs.

Reverse Engineering: The absence of publicly available smart contract source code complicates security detection. To address this, *Erays* [179] was proposed. It audits opaque smart contracts by processing styles from the Ethereum blockchain and generating high-level pseudocode for analysis. Using a "fuzzy hash" mechanism, Erays identifies function-level syntactic similarities between contracts, providing valuable insights into opaque contracts within the ecosystem.

Abnormal Behavior Identification (AB): It is often combined with other methods. A typical anomaly-based method monitors dynamic program behavior against established norms and raises an alert when it detects any deviations [176]. This approach involves detecting abnormal transaction behaviors with tools, such as VULTRON [177], TXSPECTOR [167], SoliAudit [128] and ReGuard [52], as well as the detection of abnormal data states and control flow, using tools such as Annotary [115], VulHunter [149] and ContractGuard [176].

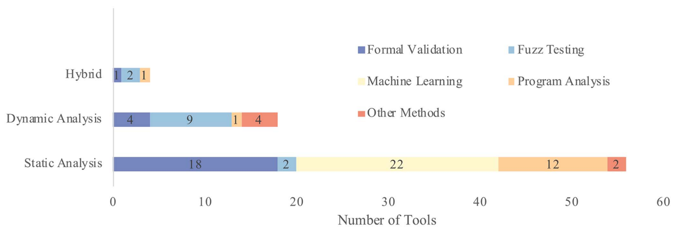

Fig. 4. Summary of detection modes.

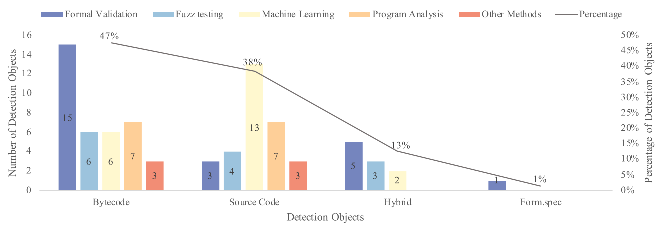

Fig. 5. Summary of detection objects.

B. Summary of Detection Tools

A comprehensive study has been conducted on various methodologies used to detect security issues in smart contracts. As shown in Table IV, we compare them in terms of six aspects: detection mode (static/dynamic), detection state (online/offline), detection object (bytecode/source code), automation level (semi-automated/fully automated), year, and types of vulnerabilities that can be supported for detection.

From Table IV, we can notice that most security detection works are carried out offline, mainly relying on static detection techniques. We also counted the number of detection tools from different analysis perspectives in Fig. 4, where static detection methods are predominant. Static detection methods are able to obtain rich semantic information closely related to vulnerabilities from source code or bytecode without the need to actually execute smart contracts, which enables cost-effective detection of multiple vulnerability types. In contrast, dynamic detection methods typically involve executing the smart contract and require a thorough understanding of the runtime environment. Although they can detect vulnerabilities that may be missed by static detection methods, they are more complex to deploy and maintain [35]. It is worth mentioning that currently only symbolic execution and fuzz testing involve a combination of static and dynamic methods.

We can also find that the primary detection objects for these existing solutions are source code and bytecode. In Ethereum, bytecode serves as an intermediary representation of the source code, allowing EVM to interpret and execute the smart contract. The statistical analysis in Fig. 5 shows that there are slightly more detection tools for bytecode-oriented analysis compared

TABLE IV
SUMMARY OF DETECTION TOOLS BASED ON FIVE METHODS

Methods		Tools	Assistive Methods	Mode		State		Object		Automation		Types	Year
				Static	Dynamic	Offline	Online	Byte	Source	Semi	Full		
Formal Verification (FV)	Theorem Proving (TP)	KEVM [105]	—	✓		✓		✓		✓		—	2018
		Isabelle/HOL [106]	—	✓		✓		✓		✓		—	2018
		VerX [107]	SE	✓		✓		✓		✓		2	2020
		Vaas [40]	—	✓			✓	✓			✓	42	2019
	Model Checking (MC)	FsolidM [108]	—	✓		✓		Formal Specification		✓		3	2017
		Zeus [109]	AI	✓		✓		✓			✓	7	2018
		VeriSolid [110]	—	✓		✓			✓	✓		6	2019
		Duo et al. [111]	—	✓		✓		✓			✓	4	2020
	Symbolic Execution (SE)	Oyente [83]	SE	✓		✓		✓	✓		✓	6	2016
		Mythril [113]	TA	✓		✓		✓	✓		✓	13	2017
		Maian [122]	—		✓	✓		✓			✓	4	2018
		Securify [114]	—	✓		✓		✓	✓		✓	12	2018
		TEETHER [116]	—	✓		✓		✓			✓	4	2018
		Annotary [115]	AB	✓		✓			✓	✓		13	2019
		Manticore [123]	—		✓	✓		✓	✓		✓	8	2019
		ETHBMC [124]	—		✓	✓		✓			✓	3	2020
		SmarTest [117]	ML	✓		✓		✓			✓	6	2021
		DefectChecker [118]	—	✓			✓	✓	✓	✓		8	2022
		Pluto [119]	—	✓		✓		✓	✓		✓	3	2022
		Park [120]	—		✓	✓		✓			✓	9	2022
		SAILFISH [99]	—	✓		✓		✓			✓	2	2022
		ExGen [121]	IR	✓		✓			✓		✓	4	2023
		NFTGuard [125]	—	✓	✓	✓		✓	✓		✓	5	2023
Fuzz Testing (FT)	Generation -based	ILF [126]	SE		✓	✓		✓	✓		✓	6	2019
		SoliAudit [128]	ML & AB	✓	✓		✓	✓	✓		✓	13	2019
		Ethploit [129]	TA		✓	✓			✓		✓	2	2020
		RLF [131]	ML		✓	✓			✓		✓	6	2022
		Xfuzz [130]	ML	✓		✓			✓		✓	3	2022
	Mutation -based	ContractFuzzer [89]	—		✓		✓	✓	✓		✓	7	2018
		Harvey [133]	—		✓	✓		✓			✓	2	2020
		sFuzz [132]	—		✓	✓		✓			✓	9	2020
		ConFuzzius [135]	TA		✓	✓		✓			✓	10	2021
		SMARTIAN [134]	—	✓	✓	✓		✓			✓	13	2021
		IR-Fuzz [136]	—	✓		✓			✓		✓	8	2023
		ItyFuzz [137]	—		✓		✓	✓			✓	4	2023
		EF↯CF [138]	—		✓	✓		✓			✓	2	2023
Machine Learning	Theorem Proving (TP)	ContractWard [139]	—	✓		✓		✓			✓	6	2020
		SCSCAN [140]	—	✓		✓			✓		✓	7	2020
		Eth2Vec [141]	—	✓		✓		✓			✓	7	2021
	Deep Learning -based (Sequence -based)	SmartEmbed [142]	—	✓			✓		✓		✓	7	2019
		ESCORT [143]	—	✓		✓		✓				✓	11
		DeeSCVHunter [144]	—	✓		✓			✓		✓	2	2021
		MODNN [145]	—	✓		✓		✓	✓		✓	12	2022
		SmartConDetect [146]	—	✓		✓			✓		✓	23	2022
		DLVA [147]	—	✓		✓		✓			✓	29	2023
		Blass [148]	—	✓		✓			✓		✓	4	2023
		VulHunter [149]	SE & AB	✓		✓		✓	✓		✓	30+	2023
		DL4SC [150]	—	✓		✓		✓			✓	3	2024
		GPTScan [151]	PA	✓		✓			✓		✓	10	2024
		PSCVFinder [152]	—	✓		✓			✓		✓	2	2023
	Deep Learning -based (Graph -based)	DR-GCN, TMP [153]	—	✓		✓			✓		✓	3	2020
		EtherGIS [154]	—	✓		✓		✓			✓	6	2022
		MANDO-GURU [155]	—	✓		✓			✓		✓	7	2022
		MRN-GCN [156]	—	✓		✓			✓		✓	3	2023
		GraphSA [157]	IR	✓		✓			✓		✓	6	2023
		SCVHunter [158]	IR	✓		✓			✓		✓	4	2024
	Both	VulnSense [159]	—	✓		✓		✓	✓		✓	2	2023
		VDM-AEI [160]	—	✓		✓			✓		✓	3	2023
Program Analysis (PA)	Abstract Interpretation (AI)	Vandal [161]	—	✓		✓		✓			✓	5	2018
		eThor [162]	FV	✓		✓		✓			✓	2	2020
		MadMax [72]	—	✓		✓		✓			✓	3	2020
	Intermediate Interpretation (IR)	SASC [163]	SE	✓		✓			✓		✓	3	2018
		SmartCheck [164]	—	✓		✓			✓		✓	21	2018
		Slither [165]	TA	✓		✓			✓		✓	93	2019
		SolAnalyser [166]	—	✓	✓	✓		✓			✓	8	2019
		TXSPECTOR [167]	AB		✓		✓	✓		✓		8	2020
		NeuCheck [168]	—	✓		✓			✓		✓	5	2021
		SmartDagger [169]	TA	✓		✓		✓			✓	4	2022
		SmartFast [170]	TA	✓		✓			✓		✓	119	2022
		SmartGraph [171]	—	✓		✓			✓		✓	3	2023
		SmartState [172]	—	✓		✓		✓			✓	2	2023
		SoliDetector [173]		✓		✓			✓		✓	20	2024
Taint Analysis (TA)		Ethainter [174]	—	✓		✓			✓		✓	5	2020
Online Detection (OD)		ContractGuard [176]	AB		✓		✓	✓		✓		11	2019
		VULTRON[177]	AB		✓		✓		✓		✓	4	2019
		FSFC [178]	—		✓		✓		✓		✓	3	2020
		SODA [175]	—		✓		✓	✓			✓	9	2020
Reverse Engineering (RE)		Erays [179]	IR	✓			✓	✓		✓		—	2018

to source code analysis. Additionally, some methods support both bytecode and source code inputs. This is primarily because bytecode is generally easier to obtain than source code, and in some cases, the source code of certain smart contracts may not be publicly accessible. It should be noted that uncollected relevant literature may impact the validity of the statistical results.

Compared to other approaches, detection tools based on machine learning exhibit a high level of automation. Besides, from Table IV, we can observe that research based on formal verification and program analysis starts quite early in the field of smart contract security detection. This is mainly because these methods usually offer high levels of security and have been widely used in the traditional software testing field [12], [180]. Correspondingly, it is evident that machine learning-based tools, especially deep learning, have mostly been published in recent years, indicating that this kind of work is still in its early stage compared with other communities. In addition, in line with the various strengths and limitations of different approaches, existing efforts have attempted to use other detection methods as auxiliary techniques to harness the complementary advantages of various methods and ultimately enhance the performance of vulnerability detection.

Smart contracts are constantly evolving, and their functions and characteristics undergo continuous changes. Therefore, detection technologies will be continuously developed and improved in order to effectively detect and address new security vulnerabilities that may arise.

C. Existing Repair Methods

Repair methods are mainly divided into on-chain and off-chain categories. On-chain repairs are generally to bundle a patch contract [181], whereas off-chain repairs involve fixing problematic code fragments directly.

Off-Chain Repair: SCRepair [182] is an automated tool that uses genetic programming and parallelization for quicker smart contract repairs. It also introduces the concept of gas dominance level to evaluate the gas usage of candidate patches. *SMARTSHIELD* [183] is a bytecode repair system that identifies and fixes three common security bugs using extracted semantic information, control-flow transformations, and DataGuard insertion, while also optimizing gas consumption and providing validation feedback. *SGUARD* [184] obtains a limited set of symbolic execution traces to identify four potential vulnerabilities, and designs targeted fixing modes. *EVMPATCH* [185] is a repair framework that automatically patches faulty contracts using a bytecode rewriting engine and verification, requiring minimal human intervention. Elysium [186] is a scalable bytecode-level repair tool that uses template-based and semantic patching to address seven types of vulnerabilities by inferring context from bytecode. Zhou et al. [187] provided a labeled dataset for automated program repair and introduced *SMARTREP*, an end-to-end statistical learning-based method that repairs contracts by adding one-line patches to faulty code. *SmartFix* [188] employs an alternating "generate-verify" approach, employing an enumerative search for patches followed by verification, and leverages statistical models to guide the repair process. *TIPS* [189] is a template-based repair framework that selects appropriate repair templates after identifying contract vulnerabilities and their locations and then generates patches by modifying code at the AST level.

On-Chain Repair: Aroc [181] is an on-chain repair framework that automatically patches vulnerable deployed smart contracts without altering their code. Its core approach involves generating patch smart contracts to preemptively abort malicious transactions.

IV. Research Challenges and Future Trends

A. The Research Challenges in Detection

Security Detection Capabilities: Most current detection methods are focused on specific types and patterns of vulnerabilities, making it difficult to effectively identify new types and patterns. The limited detection capabilities of existing solutions present challenges in making significant improvements to the status of smart contracts and blockchain security.

Widely Accepted Standards and Metrics: There are several stringent standards that can assist researchers in upholding contract security and ensuring contract reliability [12], such as SWC (Smart Contract Weakness Classification) [94] and ERC20 (Ethereum Token Standard) [190]. However, as smart contracts and their attacks continually evolve, there is a need for new standards to cover more comprehensive factors. Similarly, it is necessary to introduce more comprehensive evaluation metrics to objectively and fairly assess the effectiveness of existing solutions.

Studies on Cross-contract Detection: Cross-contract vulnerabilities typically arise when two or more smart contracts interact. The main challenge in cross-contract security detection lies in the significantly larger search space for multiple interacting contracts compared to a single contract scenario [130]. Given the rapid development of Decentralized Applications (DApps), detecting and addressing cross-contract vulnerabilities has become increasingly important.

Studies on Cross-platform Detection: Most existing detection methods are incapable of supporting cross-platform applications. To maximize benefits, it is crucial to leverage existing research foundations to enhance the security of various blockchain platforms supporting smart contracts. For instance, exploring the feasibility of transferring successful security detection schemes from Ethereum to other platforms.

Degree of Automation: Currently, some detection tools still heavily rely on manual inspection and expert knowledge, resulting in low levels of automation. As the number of smart contracts grows, these tools struggle to maintain efficient detection for large-scale contract analysis.

Interpretability of Detection Models: Deep learning-based methods have shown remarkable achievements in automated detection. However, their black-box nature makes it difficult to interpret their inner workings and decision-making processes, posing a significant obstacle to their widespread practical application.

B. Future Trends

Evaluation and Benchmarking: The datasets used by current detection tools vary greatly in terms of data volume, sources, and distribution. Establishing universal vulnerability datasets is crucial, as these benchmarks provide researchers with a consistent basis for evaluating the effectiveness of detection tools. Additionally, reasonable evaluation approaches should account for various factors and mitigate potential risks/biases, ensuring a widely accepted evaluation of solution efficacy. Beyond traditional metrics such as the false negative rate, false positive rate, and detection efficiency, it is crucial to also consider the scalability, flexibility, and reliability of a method itself.

Improving Detection Capability: Several studies have attempted to integrate various detection methods, such as ConFuzzius [135]. It combines fuzz testing with symbolic execution to overcome the limitations of fuzz testing in identifying deeper vulnerabilities. Future research may need to involve a detailed analysis of specific requirements and method characteristics, aiming to fully leverage the complementary advantages of different methods to improve detection capability.

Increasing Coverage Cycle: Static detection usually analyzes code patterns to find vulnerabilities without executing smart contracts, providing high code coverage, low cost and efficiency, and is commonly used for pre-deployment checks. Dynamic detection observes contract execution and interactions to uncover more complex and hidden vulnerabilities. It is meaningful to integrate both static and dynamic detection methods to improve the coverage cycle**:** Likewise, improving contract security should encompass the entire lifecycle, involving secure coding practices, thorough vulnerability analysis, and repair strategies both before and after deployment.

Enhancing the Robustness and Scalability of Models: While deep learning-based methods are highly effective for large-scale automated detection, they still suffer from challenges, such as availability issues, adversarial attacks and version evolution. Solving these problems depends on advancements in feature representation learning and detection modeling. Similarly to other domains, deep learning in smart contract security detection still struggles with modeling the intricate structures and behaviors of contracts. Therefore, it is crucial to obtain effective representations that are highly compatible with deep learning models and are capable of keeping pace with the continually evolving nature of smart contracts and their vulnerabilities. Additionally, efforts should be focused on enhancing resilience against adversarial attacks by developing robust detection mechanisms capable of handling sophisticated attacks and obfuscation techniques.

Improving the Interpretability of Deep Learning Models: Deep learning-based models often suffer from a lack of transparency and interpretability, making it challenging to understand their decision-making processes and formulate effective repair strategies. Hence, it is crucial to prioritize the development of credible and reliable interpretation methods for deep learning models in the future.

V. Conclusion

In this work, after analyzing 539 real security incidents, audit reports from 10 authoritative auditing institutions, and 178 related papers, we identified 27 exploited and 7 potential vulnerabilities. We also provide a detailed survey of existing security analysis methods and detection tools, and outline the current state of research in the field. Our review results indicate that although numerous excellent vulnerability detection schemes have been proposed, there are still core challenges that require further exploration. For this reason, we discussed several potential future research focuses, which include: 1) Developing widely acceptable evaluation metrics and benchmarks to provide an objective and comprehensive assessment of the effectiveness of solutions in a fair manner; 2) Improving the detection capability by exploiting a mixed mechanism combining multiple methods; 3) Increasing the coverage of the smart contract lifecycle by integrating static and dynamic detection methods; 4) Enhancing the scalability and robustness of detection systems by introducing effective representations and strengthening resilience against adversarial attacks; 5) Improving the transparency of deep learning based detection models by establishing more credible and reliable interpretation mechanisms.

References

[1] Dune, "Dune — Crypto analytics powered by community," 2018. [Online]. Available: https://dune.com/home

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SoK)," in *Principles of Security and Trust*, Berlin, Germany: Springer, 2017, pp. 164–186.

[3] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, "Security smells in smart contracts," in *Proc. IEEE 19th Int. Conf. Softw. Qual. Reliab. Secur.*, 2019, pp. 442–449.

[4] W. Dingman et al., "Classification of smart contract bugs using the NIST bugs framework," in *Proc. IEEE/ACIS 17th Int. Conf. Softw. Eng. Res. Manag. Appl.*, 2019, pp. 116–123.

[5] M. Staderini, C. Palli, and A. Bondavalli, "Classification of ethereum vulnerabilities and their propagations," in *Proc. Int. Conf. Blockchain Comput. Appl.*, 2020, pp. 44–51.

[6] R. Pise and S. Patil, "A deep dive into blockchain-based smart contract-specific security vulnerabilities," in *Proc. IEEE Int. Conf. Blockchain Distrib. Syst. Secur.*, 2022, pp. 1–6.

[7] T. Chen et al., "A large-scale empirical study on control flow identification of smart contracts," in *Proc. ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2019, pp. 1–11.

[8] A. Vacca, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges," *J. Syst. Softw.*, vol. 174, Apr. 2021, Art. no. 110891, doi: 10.1016/j.jss.2020.110891.

[9] P. Tantikul and S. Ngamsuriyaroj, "Exploring vulnerabilities in solidity smart contract," in *Proc. Int. Conf. Inf. Syst. Secur. Privacy*, 2020, pp. 317–324.

[10] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *Proc. USENIX Secur. Symp.*, 2021, pp. 1325–1341.

[11] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, "Ethereum smart contract security research: Survey and future research opportunities," *Front. Comput. Sci.*, vol. 15, no. 2, Apr. 2021, Art. no. 152802, doi: 10.1007/s11704-020-9284-9.

[12] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed Ethereum smart contract development: Issues, techniques, and future challenges," *Empir. Softw. Eng.*, vol. 26, no. 6, 2021, Art. no. 117, doi: 10.1007/s10664-021-10018-0.

[13] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proc. IEEE/ACM 42th Int. Conf. Softw. Eng.*, 2020, pp. 530–541.

[14] M. di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *Proc. IEEE Int. Conf. Decentralized Appl. Infrastruct.*, 2019, pp. 69–78.

[15] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, Australia, 2023, pp. 615–627.

[16] M. Ren et al., "Empirical evaluation of smart contract testing: What is the best choice?," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2021, pp. 566–579.

[17] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *Proc. IEEE Int. Conf. Softw. Maint. Evol.*, 2020, pp. 139–150.

[18] Z. A. Khan and A. Siami Namin, "Ethereum smart contracts: Vulnerabilities and their classifications," in *Proc. IEEE Int. Conf. Big Data*, 2020, pp. 1–10.

[19] A. Singh, R. M. Parizi, Q. Zhang, K.-K. R. Choo, and A. Dehghantanha, "Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities," *Comput. Secur.*, vol. 88, 2020, Art. no. 101654, doi: 10.1016/j.cose.2019.101654.

[20] Y. Wang et al., "Security enhancement technologies for smart contracts in the blockchain: A survey," *Trans. Emerg. Telecommun. Technol.*, vol. 32, no. 12, Dec. 2021, Art. no. e4341, doi: 10.1002/ett.4341.

[21] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, "Security defense for smart contracts: A comprehensive survey," *ACM Comput. Surv.*, vol. 55, Apr. 2023, Art. no. 3593293, doi: 10.1145/3593293.

[22] X. Feng, Q. Wang, X. Zhu, and S. Wen, "Bug searching in smart contract," 2019, *arXiv:1905.00799*.

[23] S. Munir and W. Taha, "Pre-deployment Analysis of Smart Contracts – A Survey," Jun. 30, 2023, *arXiv:2301.06079*.

[24] S. Kim and S. Ryu, "Analysis of blockchain smart contracts: Techniques and insights," in *Proc. IEEE Secure Develop.*, 2020, pp. 65–73.

[25] H. Rameder, M. di Angelo, and G. Salzer, "Review of automated vulnerability analysis of smart contracts on ethereum," *Front. Blockchain*, vol. 5, 2022, Art. no. 814977, doi: 10.3389/fbloc.2022.814977.

[26] M. Fu, L. Wu, Z. Hong, and W. Feng, "Research on vulnerability mining technique for smart contracts," *J. Comput. Appl.*, vol. 39, no. 7, pp. 1959–1966, 2019.

[27] J. Xu, F. Dang, X. Ding, and M. Zhou, "A survey on vulnerability detection tools of smart contract bytecode," in *Proc. IEEE Int. Conf. Inf. Syst. Comput. Aided Educ.*, 2020, pp. 94–98.

[28] B. C. Gupta, N. Kumar, A. Handa, and S. K. Shukla, "An insecurity study of ethereum smart contracts," in *Proc. Int. Conf. Secur. Privacy Appl. Cryptogr. Eng.*, 2020, pp. 188–207.

[29] A. López Vivar, A. T. Castedo, A. L. Sandoval Orozco, and L. J. García Villalba, "An analysis of smart contracts security threats alongside existing solutions," *Entropy*, vol. 22, no. 2, Feb. 2020, Art. no. 203, doi: 10.3390/e22020203.

[30] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," 2020, *arXiv:1908.08605*.

[31] N. F. Samreen and M. H. Alalfi, "A survey of security vulnerabilities in ethereum smart contracts," 2021, *arXiv:2105.06974*.

[32] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey," 2022, *arXiv: 2209.05872*.

[33] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, 2021, doi: 10.1145/3391195.

[34] X. Tang, K. Zhou, J. Cheng, H. Li, and Y. Yuan, "The vulnerabilities in smart contracts: A survey," in *Proc. 7th Int. Conf. Artif. Intell. Secur.*, 2021, pp. 177–190.

[35] L. Tu, X. Sun, J. Zhang, J. Cai, B. LI, and L. Bo, "Survey of vulnerability detection tools for smart contracts," *Comput. Sci.*, vol. 48, no. 11, pp. 79–88, 2021.

[36] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022, doi: 10.1109/TSE.2020.2989002.

[37] V. Piantadosi, G. Rosa, D. Placella, S. Scalabrino, and R. Oliveto, "Detecting functional and security-related issues in smart contracts: A systematic literature review," *Softw. Pract. Exp.*, vol. 53, no. 2, pp. 465–495, Feb. 2023, doi: 10.1002/spe.3156.

[38] H. Chu, P. Zhang, H. Dong, Y. Xiao, S. Ji, and W. Li, "A survey on smart contract vulnerabilities: Data sources, detection and repair," *Inform. Softw. Tech.*, vol. 159, 2023, Art. no. 107221, doi: 10.1016/j.infsof.2023.107221.

[39] W. Haouari, A. S. Hafid, and M. Fokaefs, "Vulnerabilities of smart contracts and mitigation schemes: A comprehensive survey," 2024, *arXiv:2403.19805*.

[40] Chengdu Lianan Tech, "Chengdu Lianan tech," 2018. [Online]. Available: https://www.lianantech.com

[41] SlowMist, "SlowMist hacked - SlowMist Zone," 2019. [Online]. Available: https://hacked.slowmist.io/

[42] SmartDec, "SmartDec," 2022. [Online]. Available: https://smartdec.net/

[43] PeckShield, "PeckShield - Industry leading blockchain security company," 2018. [Online]. Available: https://peckshield.com/

[44] Quantstamp, "Quantstamp: The leader in Web3 security," 2017. [Online]. Available: https://quantstamp.com

[45] NONEAGE, "NONEAGE | focus on blockchain ecosystem security," 2018. [Online]. Available: https://www.noneage.com/

[46] BlockSec, "BlockSec building blockchain security infrastructure," 2020. [Online]. Available: https://www.blocksec.com/

[47] CertiK, "Web3 security leaderboard," 2018. [Online]. Available: https://www.certik.com

[48] Beosin, "Blockchain security solutions - beosin," 2018. [Online]. Available: https://beosin.com/

[49] SharkTeam, "SharkTeam - The world's leading Web3 security service provider," 2021. [Online]. Available: https://sharkteam.org/

[50] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2020, pp. 415–427.

[51] N. F. Samreen, "Reentrancy vulnerability identification in ethereum smart contracts," in *Proc. IEEE Int. Workshop Blockchain Oriented Softw. Eng.*, 2020, pp. 22–29.

[52] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: Finding reentrancy bugs in smart contracts," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.: Companion*, 2018, pp. 65–68.

[53] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2020, pp. 557–560.

[54] B. Li, Z. Pan, and T. Hu, "ReDefender: Detecting reentrancy vulnerabilities in smart contracts automatically," *IEEE Trans. Rel.*, vol. 71, no. 2, pp. 984–999, Jun. 2022, doi: 10.1109/TR.2022.3161634.

[55] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "Hunting for re-entrancy attacks in ethereum smart contracts via static analysis," 2020, *arXiv:2007.01029*.

[56] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 1029–1040.

[57] R. Yu, J. Shu, D. Yan, and X. Jia, "ReDetect: Reentrancy vulnerability detection in smart contracts with high accuracy," in *Proc. Int. Conf. Mobil., Sens. Netw.*, 2021, pp. 412–419.

[58] H. Wu et al., "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *Proc. Int. Symp. Softw. Reliab. Eng.*, 2021, pp. 378–389.

[59] H. Zhu, K. Yang, L. Wang, Z. Xu, and V. S. Sheng, "GraBit: A sequential model-based framework for smart contract vulnerability detection," in *Proc. Int. Symp. Softw. Reliab. Eng.*, 2023, pp. 568–577.

[60] Z. Zhang et al., "Reentrancy vulnerability detection and localization: A deep learning based two-phase approach," in *Proc. IEEE/ACM 37th Int. Conf. Autom. Softw. Eng.*, 2022, pp. 1–13.

[61] C. Sendner, R. Zhang, A. Hefter, A. Dmitrienko, and F. Koushanfar, "G-Scan: Graph neural networks for line-level vulnerability identification in smart contracts," 2023, *arXiv:2307.08549*.

[62] F. Liu, Z. Hao, H. Huang, and Y. Xiang, "Research on detection method for smart contract reentrancy vulnerability based on manifold pigeon optimization," *Sci. Sin.(Technol.)*, vol. 53, no. 11, pp. 1922–1938, 2021, doi: 10.1360/SST-2021-0365.

[63] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 24–27.

[64] A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, "A mechanism to detect and prevent ethereum blockchain smart contract reentrancy attacks," *Front. Comput. Sci.*, vol. 3, Feb. 2021, Art. no. 598780, doi: 10.3389/fcomp.2021.598780.

[65] M. Eshghie, C. Artho, and D. Gurov, "Dynamic vulnerability detection on smart contracts using machine learning," in *Proc. 25th Int. Conf. Eval. Assess. Softw. Eng.*, 2021, pp. 305–312.

[66] E. Cecchetti, S. Yao, H. Ni, and A. C. Myers, "Compositional security for reentrant applications," in *Proc. IEEE Symp. Secur. Priv.*, 2021, pp. 1249–1267.

[67] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.*, 2023, pp. 295–306.

[68] Solidity, "Solidity — Solidity 0.8.26 documentation," 2024. [Online]. Available: https://docs.soliditylang.org/en/v0.8.26/

[69] KotET, "KotET - post-mortem investigation," 2016. [Online]. Available: https://www.kingoftheether.com/postmortem.html
[70] N. F. Samreen and M. H. Alalfi, "SmartScan: An approach to detect denial of service vulnerability in ethereum smart contracts," in *Proc. IEEE/ACM Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2021, pp. 17–26.
[71] T. Chen et al., "An adaptive gas cost mechanism for ethereum to defend against under-priced DoS attacks," in *Proc. 13th Int. Conf. Inform. Secur. Pract. Experience*, 2017, pp. 3–24.
[72] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, pp. 87–95, 2020, doi: 10.1145/3416262.
[73] C. Li, "Gas estimation and optimization for smart contracts on ethereum," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2021, pp. 1082–1086.
[74] B. Nassirzadeh, H. Sun, S. Banescu, and V. Ganesh, "Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities," 2022, *arXiv:2112.14771*.
[75] A. Ghaleb, J. Rubin, and K. Pattabiraman, "eTainter: Detecting gas-related vulnerabilities in smart contracts," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2022, pp. 728–739.
[76] "BUGX.IO," 2018. [Online]. Available: https://www.bugx.io/
[77] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.
[78] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "EASYFLOW: Keep ethereum away from overflow," in *Proc. IEEE/ACM 41th Int. Conf. Softw. Eng.: Companion*, 2019, pp. 23–26.
[79] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proc. Int. Conf. Cryptogr. Secur. Privacy*, 2020, pp. 110–115.
[80] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1678–1694.
[81] J. Sun, S. Huang, C. Zheng, T. Wang, C. Zong, and Z. Hui, "Mutation testing for integer overflow in ethereum smart contracts," *Tsinghua Sci. Technol.*, vol. 27, no. 1, pp. 27–40, Feb. 2022, doi: 10.26599/TST.2020.9010036.
[82] "Ethereum smart contract best practices," 2018. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/
[83] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.
[84] OpenZeppelin, "OpenZeppelin," 2015. [Online]. Available: https://www.openzeppelin.com
[85] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Ana.*, 2022, pp. 716–727.
[86] A. Ghaleb, J. Rubin, and K. Pattabiraman, "AChecker: Statically detecting smart contract access control vulnerabilities," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.*, 2023, pp. 945–956.
[87] Y. Fang et al., "Beyond 'Protected' and 'Private': An empirical security analysis of custom function modifiers in smart contracts," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2023, pp. 1157–1168.
[88] "Sigma prime," [Online]. Available: https://blog.sigmaprime.io/
[89] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing smart contracts for vulnerability detection," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2018, pp. 259–269.
[90] J. Chen, X. Xia, D. Lo, and J. Grundy, "Why do smart contracts self-destruct? Investigating the selfdestruct function on ethereum," *ACM Trans. Softw. Eng. Meth.*, vol. 31, no. 2, pp. 1–37, Apr. 2022, doi: 10.1145/3488245.
[91] D. Prechtel, T. Gros, and T. Muller, "Evaluating spread of 'gasless send' in ethereum smart contracts," in *Proc. IFIP Int. Conf. New Technol. Mobil. Secur.*, 2019, pp. 1–6.
[92] "El dorado exchange," 2023. [Online]. Available: https://www.ede.finance/
[93] J. Zhang, Y. Li, J. Gao, Z. Guan, and Z. Chen, "Siguard: Detecting signature-related vulnerabilities in smart contracts," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.: Companion, 2023*, pp. 31–35.
[94] "Overview · smart contract weakness classification and test cases," 2020. [Online]. Available: http://swcregistry.io/
[95] N. Ruaro, F. Gritti, R. McLaughlin, I. Grishchenko, C. Kruegel, and G. Vigna, "Not your type! Detecting storage collision vulnerabilities in ethereum smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2024, pp. 1–17.
[96] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *Proc. USENIX Secur. Symp.*, Santa Clara, CA, 2019, pp. 1591–1607.
[97] Zhang, L. Y., Xue J., Chen J., Yan M., and Mao X., "Detection of smart contract timestamp vulnerability based on data-flow path learning," *J. Softw.*, vol. 35, no. 5, pp. 2325–2339, 2023, doi: 10.13328/j.cnki.jos.006989.
[98] C. Natoli and V. Gramoli, "The Blockchain Anomaly," in *Proc. IEEE Int. Symp. Netw. Comput. Appl.*, 2016, pp. 310–317.
[99] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "SAILFISH: Vetting smart contract state-inconsistency bugs in seconds," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 161–178.
[100] P. Qian et al., "Demystifying random number in ethereum smart contract: Taxonomy, vulnerability identification, and attack detection," 2023, *arXiv:2304.12645*.
[101] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, 2021, doi: 10.1145/3391195.
[102] M. Dominik, "Smart contract security field guide," 2023. [Online]. Available: https://scsfg.io/
[103] C. Agrawal, "Smart contract best practice," Medium, 2023. [Online]. Available: https://infosecwriteups.com/smart-contract-best-practice-dc1e4a8ca788
[104] T. Sun and W. Yu, "A formal verification framework for security issues of blockchain smart contracts," *Electronics*, vol. 9, no. 2, Feb. 2020, Art. no. 255, doi: 10.3390/electronics9020255.
[105] E. Hildenbrandt et al., "KEVM: A complete formal semantics of the ethereum virtual machine," in *Proc. IEEE Comput. Secur. Found. Symp.*, 2018, pp. 204–217.
[106] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in Isabelle/HOL," in *Proc. ACM SIGPLAN Int. Conf. Certif. Programs Proofs*, 2018, pp. 66–77.
[107] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "VerX: Safety verification of smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1661–1677.
[108] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," 2017, *arXiv:1711.09327*.
[109] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.
[110] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for ethereum," in *Proc. Int. Conf. Financ. Cryptogr. Data Secur.*, Cham, 2019, pp. 446–465.
[111] W. Duo, H. Xin, and M. Xiaofeng, "Formal analysis of smart contract based on colored petri nets," *IEEE Intell. Syst.*, vol. 35, no. 3, pp. 19–30, May/Jun. 2020, doi: 10.1109/MIS.2020.2977594.
[112] Z. Tian, "Smart contract defect detection based on parallel symbolic execution," in *Proc. Int. Conf. Circuits, Syst. Simul.*, 2019, pp. 127–132.
[113] Consensys/mythril, "Python. Consensys," 2018. [Online]. Available: https://github.com/Consensys/mythril
[114] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.
[115] K. Weiss and J. Schütte, "Annotary: A concolic execution system for developing secure smart contracts," in *Proc. 24th Eur. Symp. Res. Comput. Secur.*, 2019, pp. 747–766.
[116] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proc. USENIX Secur. Symp.*, 2018, pp. 1317–1333.
[117] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution," in *Proc. USENIX Secur. Symp.*, 2021, pp. 1361–1378.
[118] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "DefectChecker: Automated smart contract defect detection by analyzing EVM bytecode," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2189–2207, Jul. 2022, doi: 10.1109/TSE.2021.3054928.
[119] F. Ma et al., "Pluto: Exposing vulnerabilities in inter-contract scenarios," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4380–4396, Nov. 2022, doi: 10.1109/TSE.2021.3117966.
[120] P. Zheng, Z. Zheng, and X. Luo, "Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2022, pp. 740–751.

[121] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "ExGen: Cross-platform, automated exploit generation for smart contract vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 1, pp. 650–664, Jan. 2023, doi: 10.1109/TDSC.2022.3141396.

[122] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.

[123] M. Mossberg et al., "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proc. IEEE/ACM 34th Int. Conf. Autom. Softw. Eng.*, 2019, pp. 1186–1189.

[124] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *Proc. USENIX Secur. Symp.*, 2020, pp. 2757–2774.

[125] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in NFT smart contracts," in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2023, pp. 373–384.

[126] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 531–548.

[127] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Comput. Secur.*, vol. 75, pp. 118–137, Jun. 2018, doi: 10.1016/j.cose.2018.02.002.

[128] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, "SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing," in *Proc. Int. Conf. Internet Things: Syst., Manag. Secur.*, 2019, pp. 458–465.

[129] Q. Zhang, Y. Wang, J. Li, and S. Ma, "EthPloit: From fuzzing to efficient exploit generation against smart contracts," in *Proc. IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2020, pp. 116–126.

[130] Y. Xue et al., "xFuzz: Machine learning guided cross-contract fuzzing," *IEEE Trans. Dependable Secure Comput.*, vol. 21, no. 2, pp. 515–529, Mar./Apr. 2024, doi: 10.1109/TDSC.2022.3182373.

[131] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2022, pp. 1–12.

[132] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. IEEE/ACM 42th Int. Conf. Softw. Eng.*, 2020, pp. 778–788.

[133] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proc. 28th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 1398–1409.

[134] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "SMARTIAN: Enhancing smart contract fuzzing with static and dynamic dataflow analyses," in *Proc. IEEE/ACM 36th Int. Conf. Autom. Softw. Eng.*, 2021, pp. 227–239.

[135] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "ConFuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2021, pp. 103–119.

[136] Z. Liu et al., "Rethinking smart contract fuzzing: Fuzzing with invocation ordering and important branch revisiting," *IEEE Trans. Inf. Forensics Secur.*, vol. 18, pp. 1237–1251, 2023, doi: 10.1109/TIFS.2023.3237370.

[137] C. Shou, S. Tan, and K. Sen, "ItyFuzz: Snapshot-based fuzzer for smart contract," in *Proc. ACM SIGSOFT Int. Symp. Softw Test. Anal.*, 2023, pp. 322–333.

[138] M. Rodler et al., "EF*****CF: High performance smart contract fuzzing for exploit generation," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2023, pp. 449–471.

[139] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated vulnerability detection models for ethereum smart contracts," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 1133–1144, Second Quarter 2021, doi: 10.1109/TNSE.2020.2968505.

[140] X. Hao, W. Ren, W. Zheng, and T. Zhu, "SCScan: A SVM-based scanning system for vulnerabilities in blockchain smart contracts," in *Proc. IEEE Int. Conf. Trust Secur. Priv. Comput. Commun.*, Dec. 2020, pp. 1598–1605.

[141] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: Learning contract-wide code representations for vulnerability detection on ethereum smart contracts," in *Proc. 5th ACM Int. Symp. Blockchain Secure Crit. Infrastructure*, 2021, pp. 47–59.

[142] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "SmartEmbed: A tool for clone and bug detection in smart contracts through structural code embedding," in *Proc. IEEE Int. Conf. Softw. Maint. Evol.*, 2019, pp. 394–397.

[143] C. Sendner et al., "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–18.

[144] X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, "DeeSCVHunter: A deep learning-based framework for smart contract vulnerability detection," in *Proc. Int. Jt. Conf. Neural Netw.*, 2021, pp. 1–8.

[145] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, and H. Chen, "Smart contract vulnerability detection combined with multi-objective detection," *Comput. Netw.*, vol. 217, 2022, Art. no. 109289, doi: 10.1016/j.comnet.2022.109289.

[146] A. Kammoun, R. Slama, H. Tabia, T. Ouni, and M. Abid, "Generative adversarial networks for face generation: A survey," *ACM Comput. Surv.*, vol. 55, 2022, Art. no. 94, doi: 10.1145/1122445.1122456.

[147] T. Abdelaziz and A. Hobor, "Smart learning to find dumb contracts," Apr. 20, 2023, *arXiv:2304.10726*.

[148] X. Ren, Y. Wu, J. Li, D. Hao, and M. Alam, "Smart contract vulnerability detection based on a semantic code structure and a self-designed neural network," *Comput. Electr. Eng.*, vol. 109, Aug. 2023, Art. no. 108766, doi: 10.1016/j.compeleceng.2023.108766.

[149] Z. Li et al., "VulHunter: Hunting vulnerable smart contracts at EVM bytecode-level via multiple instance learning," *IEEE Trans. Softw. Eng.*, vol. 49, no. 11, pp. 4886–4916, Nov. 2023, doi: 10.1109/TSE.2023.3317209.

[150] Y. Liu, C. Wang, and Y. Ma, "DL4SC: A novel deep learning-based vulnerability detection framework for smart contracts," *Autom. Softw. Eng.*, vol. 31, no. 1, 2024, Art. no. 24, doi: 10.1007/s10515-024-00418-z.

[151] Y. Sun et al., "GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[152] L. Yu, J. Lu, X. Liu, L. Yang, F. Zhang, and J. Ma, "PSCVFinder: A prompt-tuning based framework for smart contract vulnerability detection," in *Proc. Int. Symp. Softw. Reliab. Eng.*, 2023, pp. 556–567.

[153] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network," in *Proc. Int. Joint Conf. Artif. Intell.*, 2020, pp. 3283–3290.

[154] Q. Zeng et al., "EtherGIS: A vulnerability detection framework for ethereum smart contracts based on graph learning features," in *Proc. IEEE 46th Annu. Comput. Softw. Appl. Conf.*, 2022, pp. 1742–1749.

[155] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "MANDO-GURU: Vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *Proc. 30th ACM Jt. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1736–1740.

[156] H. Liu, Y. Fan, L. Feng, and Z. Wei, "Vulnerable smart contract function locating based on multi-relational nested graph convolutional network," *J. Syst. Softw.*, vol. 204, Oct. 2023, Art. no. 111775, doi: 10.1016/j.jss.2023.111775.

[157] L. He, X. Zhao, Y. Wang, J. Yang, and X. Sun, "GraphSA: Smart contract vulnerability detection combining graph neural networks and static analysis," in *ECAI 2023*, Amsterdam, Netherlands: IOS Press, 2023, pp. 1020–1027, doi: 10.3233/FAIA230374.

[158] F. Luo et al., "SCVHunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[159] P. T. Duy et al., "VulnSense: Efficient vulnerability detection in ethereum smart contracts by multimodal learning with graph neural network and language model," 2023, *arXiv:2309.08474*.

[160] L. Li, Y. Liu, G. Sun, and N. Li, "Smart contract vulnerability detection based on automated feature extraction and feature interaction," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 9, pp. 4916–4916–4929, Sep. 2024, doi: 10.1109/TKDE.2023.3333371.

[161] L. Brent et al., "Vandal: A scalable security analysis framework for smart contracts," 2018, *arXiv:1809.03981*.

[162] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: Practical and provably sound static analysis of ethereum smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 621–640.

[163] E. Zhou et al., "Security assurance for smart contract," in *Proc. IFIP Int. Conf. New Technol., Mobil. Secur.*, 2018, pp. 1–5.

[164] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. IEEE/ACM Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16.

[165] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.

[166] S. Akca, A. Rajan, and C. Peng, “SolAnalyser: A framework for analysing and testing smart contracts,” in *Proc. 26th Asia Pac. Softw. Eng. Conf.*, 2019, pp. 482–489.
[167] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, “TXSPECTOR: Uncovering attacks in ethereum from transactions,” in *Proc. USENIX Secur. Symp.*, 2020, pp. 2775–2792.
[168] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, “NeuCheck: A more practical Ethereum smart contract security analysis tool,” *Softw. Pract. Exp.*, vol. 51, no. 10, pp. 2065–2084, 2021, doi: 10.1002/spe.2745.
[169] Z. Liao, Z. Zheng, X. Chen, and Y. Nan, “SmartDagger: A bytecode-based static analysis approach for detecting cross-contract vulnerability,” in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2022, pp. 752–764.
[170] Z. Li et al., “SmartFast: An accurate and robust formal analysis tool for Ethereum smart contracts,” *Empir. Softw. Eng.*, vol. 27, no. 7, Dec. 2022, Art. no. 197, doi: 10.1007/s10664-022-10218-2.
[171] A. Zhukov and V. Korkhov, “SmartGraph: Static analysis tool for solidity smart contracts,” in *Proc. Int. Conf. Comput. Sci. Appl.*, 2023, pp. 584–598.
[172] Z. Liao, S. Hao, Y. Nan, and Z. Zheng, “SmartState: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis,” in *Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2023, pp. 980–991.
[173] T. Hu, B. Li, Z. Pan, and C. Qian, “Detect defects of solidity smart contract based on the knowledge graph,” *IEEE Trans. Rel.*, vol. 73, no. 1, pp. 186–202, Mar. 2024, doi: 10.1109/TR.2023.3233999.
[174] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, “Ethainter: A smart contract security analyzer for composite vulnerabilities,” in *Proc. ACM SIGPLAN Conf. Prog.. Lang. Des. Implementation*, 2020, pp. 454–469.
[175] T. Chen et al., “SODA: A generic online detection framework for smart contracts,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 23–26.
[176] X. Wang, J. He, Z. Xie, G. Zhao, and S. C. Cheung, “ContractGuard: Defend ethereum smart contracts with embedded intrusion detection,” *IEEE Trans. Serv. Comput.*, vol. 13, no. 2, pp. 314–328, Mar./Apr. 2020, doi: 10.1109/TSC.2019.2949561.
[177] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, “VULTRON: Catching vulnerable smart contracts once and for all,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: New Ideas Emerg. Results*, 2019, pp. 1–4.
[178] Z. Wang, W. Dai, K.-K. R. Choo, H. Jin, and D. Zou, “FSFC: An input filter-based secure framework for smart contract,” *J. Netw. Comput. Appl.*, vol. 154, 2020, Art. no. 102530, doi: 10.1016/j.jnca.2020.102530.
[179] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, “Erays: Reverse engineering ethereum’s opaque smart contracts,” in *Proc. USENIX Secur. Symp.*, 2018, pp. 1371–1385.
[180] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, “A survey of smart contract formal specification and verification,” 2020, *arXiv:2008.02712*.
[181] H. Jin, Z. Wang, M. Wen, W. Dai, Y. Zhu, and D. Zou, “Aroc: An automatic repair framework for on-chain smart contracts,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4611–4629, Nov. 2022, doi: 10.1109/TSE.2021.3123170.
[182] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, “Smart contract repair,” 2020, *arXiv:1912.05823*.
[183] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, “SMARTSHIELD: Automatic smart contract protection made easy,” in *Proc. IEEE Int. Conf. Softw. Anal. Evol. Reengineering*, 2020, pp. 23–34.
[184] T. D. Nguyen, L. H. Pham, and J. Sun, “SGUARD: Towards fixing vulnerable smart contracts automatically,” in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 1215–1229.
[185] M. Rodler, W. Li, G. O. Karame, and L. Davi, “EVMPatch: Timely and automated patching of ethereum smart contracts,” in *Proc. USENIX Secur. Symp.*, 2021, pp. 1289–1306.
[186] C. F. Torres, H. Jonker, and R. State, “Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts,” in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, 2022, pp. 115–128.
[187] X. Zhou, Y. Chen, H. Guo, X. Chen, and Y. Huang, “Security code recommendations for smart contract,” in *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reengineering*, 2023, pp. 190–200.
[188] S. So and H. Oh, “SmartFix: Fixing vulnerable smart contracts by accelerating generate-and-verify repair using statistical models,” in *Proc. ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 185–197.
[189] Q. Chen et al., “Tips: Towards automating patch suggestion for vulnerable smart contracts,” *Automat. Softw. Eng.*, vol. 30, no. 2, 2023, Art. no. 31, doi: 10.1007/s10515-023-00392-y.
[190] “EIPs/EIPS/eip-20.md at master · ethereum/EIPs,” GitHub. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

Huijuan Zhu received the PhD degree from the School of Computer and Control Engineering, University of Chinese Academy of Sciences, Beijing, China in 2017. Her research interests include malware detection and machine learning. She is an associate professor with the School of Computer Science and Communication Engineering, Jiangsu University.

Lei Yang is now studying with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. Her research interests include Ethereum security and deep learning.

Liangmin Wang received the BS degree in computational mathematics from Jilin University, Changchun, China, in 1999, and the PhD degree in cryptology from Xidian University, Xi’an, China in 2007. He is a full professor with the School of Cyber Science and Engineering, Southeast University, Nanjing, China. He has been honored as a “Wan-Jiang Scholar” of Anhui Province since Nov. 2013. Now his research interests include data security & privacy. He has published more than 70 technical papers at premium international journals and conferences, e.g., *IEEE/ACM Transactions on Networking* and IEEE International Conference on Computer Communications. He has severed as a TPC member of many IEEE conferences, such as IEEE ICC, IEEE HPCC, IEEE TrustCOM.

Victor S. Sheng (Senior Member, IEEE) received the master’s degree in computer science from the University of New Brunswick, Canada, in 2003, and the PhD degree in computer science from Western University, Ontario, Canada, in 2007. He is an associate professor of computer science with Texas Tech University, and the founding director of the Data Analyt-ics Lab (DAL). His research interests include data mining, machine learning, and related applications. He was an associate research scientist and NSERC postdoctoral fellow in information systems with Stern Business School, New York University, after he obtained his PhD. He is a lifetime member of the ACM. He received the test-of-time award for research from KDD’20, the best paper award runner-up from KDD’08, and the best paper award from ICDM’11. He is an area chair and SPC/PC member for several international top conferences and an associate editor for several international journals.