Survey on Quality Assurance of Smart Contracts

ZHIYUAN WEI, School of Computer Science, Beijing Institute of Technology, Beijing, China
JING SUN, Faculty of Science, University of Auckland - City Campus, Auckland, New Zealand
ZIJIAN ZHANG, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China
XIANHAO ZHANG, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China
XIAOXUAN YANG, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China
LIEHUANG ZHU, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, China

As blockchain technology continues to advance, the secure deployment of smart contracts has become increasingly prevalent, underscoring the critical need for robust security measures. This surge in usage has led to a rise in security breaches, often resulting in substantial financial losses for users. This article presents a comprehensive survey of smart contract quality assurance, from understanding vulnerabilities to evaluating the effectiveness of detection tools. Our work is notable for its innovative classification of 40 smart contract vulnerabilities, mapping them to established attack patterns. We further examine nine defense mechanisms, assessing their efficacy in mitigating smart contract attacks. Furthermore, we develop a labeled dataset as a benchmark encompassing 10 common vulnerability types, which serves as a critical resource for future research. We also conduct comprehensive experiments to evaluate 14 vulnerability detection tools, providing a comparative analysis that highlights their strengths and limitations. In summary, this survey synthesizes state-of-the-art knowledge in smart contract security, offering practical recommendations to guide future research and foster the development of robust security practices in the field.

CCS Concepts: • **General and reference → Surveys and overviews**; • **Security and privacy → Domain-specific security and privacy architectures;**

Additional Key Words and Phrases: Smart contract, security, vulnerabilities, attacks, defenses

This work is supported by the National Key Research and Development Program of China (grant nos. 2021YFB2701200, 2021YFB2701202, and 2022YFB2702402), the National Natural Science Foundation of China (NSFC; grant nos. 62172040, 62002094, and U1836212), and the Anhui Provincial Natural Science Foundation (grant no. 2008085MF196).
Authors' Contact Information: Zhiyuan Wei, School of Computer Science, Beijing Institute of Technology, Beijing, Beijing, China; e-mail: weizhiyuan@bit.edu.cn; Jing Sun, Faculty of Science, University of Auckland —City Campus, Auckland, New Zealand; e-mail: jing.sun@auckland.ac.nz; Zijian Zhang (Corresponding author), School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, Beijing, China; e-mail: zhangzijian@bit.edu.cn; Xianhao Zhang, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, Beijing, China; e-mail: zhangxh@bit.edu.cn; Xiaoxuan Yang, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, Beijing, China; e-mail: yangxiaoxuan979@gmail.com; Liehuang Zhu, School of Cyberspace Science and Technology, Beijing Institute of Technology, Beijing, Beijing, China; e-mail: liehuangz@bit.edu.cn.

ACM 0360-0300/2024/10-ART32
https://doi.org/10.1145/3695864

ACM Reference Format:
Zhiyuan Wei, Jing Sun, Zijian Zhang, Xianhao Zhang, Xiaoxuan Yang, and Liehuang Zhu. 2024. Survey on Quality Assurance of Smart Contracts. *ACM Comput. Surv.* 57, 2, Article 32 (October 2024), 36 pages. https://doi.org/10.1145/3695864

1 Introduction

Blockchain technology has gained prominence in academia and industry as a secure and private solution for diverse applications [54, 123]. As a distributed ledger, blockchain technology is replicated and shared among a network of peer-to-peer nodes. It eliminates the need for intermediaries, thereby providing decentralization, transparency, immutability, security, and reliability. By maintaining a chronologically growing and immutable data record, blockchain systems have become ideal for a multitude of domains, ranging from business to healthcare. In the realm of business, blockchain significantly reduces operational costs [20]. Healthcare applications of blockchain range from securing cloud-based cyber-physical systems to enhancing privacy in monitoring systems [94]. Its role in managing healthcare challenges, particularly during the COVID-19 pandemic, further exemplifies its broad utility and adaptability [41]. Additionally, blockchain's integration with **artificial intelligence (AI)** in areas such as resource allocation for drone-terrestrial networks indicates its expanding scope in various innovative fields [86].

Smart contracts have emerged as a pivotal component of blockchain technology. Originally conceptualized in the early 1990s, they gained significant traction with the advent of blockchain platforms such as Ethereum. These self-executing contracts, implemented through lines of code [10, 72], have transformed traditional contract enforcement by automating it and embedding it within the blockchain network. They ensure transparency and immutability of contract rules, enforced by blockchain network participants. They allow **decentralized applications (DApps)** to be built on the blockchain networks, facilitating applications in diverse domains such as financial services [128], healthcare [91, 94], the **Internet of Things (IoT)** [46], crowdfunding [99], and supply chain [104, 108].

Typically, smart contracts are associated with the blockchain's native cryptocurrency, which compensates network participants for executing contracts. This incentive mechanism maintains the blockchain's security and decentralization. The innovative nature and transformative potential of smart contracts in business transactions have garnered considerable research attention. Despite increasing interest, numerous unresolved research questions persist in this nascent field.

Compared with traditional programs, smart contracts possess unique characteristics that render them more susceptible to software attacks. Primarily, smart contracts are immutable; once deployed on the blockchain, their code remains unalterable. This immutability, while ensuring trust and transparency, precludes easy rectification of vulnerabilities or errors without deploying a new contract version. Smart contracts often manage high-value digital assets, including cryptocurrencies and tokens. This concentration of valuable assets attracts malicious actors seeking to exploit contract code vulnerabilities. Additionally, public blockchains such as Ethereum are permissionless, enabling universal access and interaction with deployed smart contracts. This open architecture lowers the entry barrier for potential attackers, facilitating the identification and exploitation of smart contract vulnerabilities. The prevalent practice of publishing contract code on platforms such as Etherscan further exposes it to potential security risks.

The exploitation of these vulnerabilities has led to significant financial losses and broader implications in the blockchain world. For instance, the 2016 **Decentralized Autonomous Organization (DAO)** attack, in which attackers diverted over 3.6 million Ether, resulted in a loss of around 70 million USD and triggered a substantial drop in Ether's value [132]. The Parity

Table 1. Comparison of Surveys on Smart Contract Security

Research	Venue	Vulnerability Types	Attacks	Defense Methods	Tools	Datasets
Atzei et al. [10]	ETAPS'17	12	9	3	–	–
Zheng et al. [130]	FGCS Journal'20	✓	–	–	–	–
Chen et al. [24]	ACM Comput. Surv.	**40**	**29**	8	–	–
Di Angelo and Salzer [33]	DAPPCON'19	–	–	–	27	–
Durieux et al. [34]	ICSE'20	10	–	–	35	✓
Tolmach et al. [114]	ACM Comput. Surv.	–	–	5	34	–
Ivanov et al. [55]	ACM Comput. Surv.	37	–	8	38	–
Chu et al. [30]	IST Journal'23	12	–	2	20	✓
He et al. [49]	IEEE IoT Journal'23	8	–	6	6	–
Zhang et al. [129]	ICSE'23	14	–	6	5	✓
Chaliasos et al. [22]	ICSE'24	14	–	6	5	✓
This work	–	**40**	9	**9**	**169**	✓

Multisig Wallet incident led to a loss of about 30 million USD [85]. In 2020, the KuCoin hack resulted in the loss of over 280 million USD worth of cryptocurrencies. This incident underscored the risks inherent in smart contracts and catalyzed discussions on enhancing security measures and potential regulatory frameworks. The February 2020 reentrancy attack on the bZx DeFi (decentralized finance) platform, resulting in losses of nearly 350,000 USD, highlighted vulnerabilities in complex DeFi smart contracts and emphasized the necessity for rigorous testing and security audits [31]. Beyond the immediate financial losses, these incidents have raised concerns about the security and reliability of smart contracts, impacting user trust and investor confidence.

Due to the novelty and potential impact of smart contracts, there are some notable surveys involving vulnerable smart contracts from various perspectives. Table 1 underscores the similarities and differences between our study and existing surveys. Our study comprehensively examines multiple facets of smart contract security, encompassing vulnerability classifications, attack, defense methods, detection tools, and relevant datasets.

Atzeri et al. [10] pioneered the survey of smart contract security, categorizing 12 vulnerabilities into three domains: Solidity, EVM bytecode, and blockchain. This classification framework has been widely adopted in subsequent research and has practical applications in identifying and mitigating security risks in real-world smart contract deployments. Zheng et al. [130] conducted a comparative analysis of various smart contract platforms and developed a taxonomy of applications. Their study evaluated the features of diverse platforms, emphasizing the potential impact of vulnerabilities on sectors such as finance and healthcare, where transaction integrity and security are paramount.

Chen et al. [24] expanded the scope beyond vulnerability analysis to encompass defense mechanisms for blockchain security. Their research examined various defense techniques and strategies employed to bolster smart contract security. Di Angelo and Salzer [33] conducted a comprehensive survey on vulnerability detection tools tailored for Ethereum smart contracts. Their study encompassed tools from both academia and industry, offering insights into the array of available vulnerability identification tools. Durieux et al. [34] performed a comprehensive evaluation of nine smart contract detection tools. They assessed these tools using a labeled dataset and numerous real-world smart contracts, yielding a thorough analysis of their efficacy.

Tolmach et al. [114] concentrated on the formal verification of smart contracts across diverse applications. Their research explored the application of formal verification techniques for ensuring the correctness and security of smart contracts. Ivanov et al. [55] focused on security threat mitigation, developing a concise vulnerability map encompassing 37 known vulnerabilities. This map synthesized the vulnerability-addressing capabilities of 38 distinct classes of threat mitigation solutions. Chu et al. [30] examined vulnerability data sources, detection methods, and repair techniques. Through automated vulnerability injection into contracts, they constructed an objective vulnerability dataset, enhancing the analysis of existing security methods' performance.

He et al. [49] evaluated performance detection methods and their associated tools. They primarily tested six detection tools, analyzing them from multiple perspectives: detection accuracy, execution time, and Solidity version compatibility. Zhang et al. [129] conducted a systematic investigation of 462 defects reported in CodeArena audits and 54 exploits, assessing the detection capabilities of existing tools. Their survey indicates that existing tools can detect machine-auditable vulnerabilities, with more than 80% of exploitable bugs falling into this category. Some vulnerabilities are too complex or subtle and require the expertise of multiple human auditors. Chaliasos et al. [22] compared five **state-of-the-art (SOTA)** automated security tools with 49 developers and auditors from leading DeFi protocols. Their findings revealed that the tools could have prevented only 8% of the attacks in their dataset.

Although previous surveys offer valuable insights into specific aspects of smart contract security, they often lack a comprehensive analysis encompassing all perspectives of vulnerable smart contracts. It is crucial to develop a thorough understanding of vulnerabilities, attacks, defenses, and tool evaluation to gain a holistic view of the challenges and potential solutions related to smart contract security. Our article aims to bridge this gap by incorporating multiple perspectives, offering a more comprehensive analysis. Through a systematic examination of vulnerabilities, attacks, defenses, and tools, we strive to provide a comprehensive understanding of the challenges posed by vulnerable smart contracts and explore viable solutions.

The primary objective of our article is to contribute to the existing body of research on smart contract security by providing a comprehensive and up-to-date analysis. To achieve this objective, we present the following key contributions:

— **Novel Vulnerability Classification:** We propose a novel vulnerability classification that enhances the understanding of the underlying causes of vulnerabilities in smart contracts. This classification facilitates more effective categorization and analysis of vulnerabilities by researchers, establishing a robust foundation for security measures and enhanced vulnerability management.
— **In-depth Analysis of Real-World Attacks:** We present a comprehensive analysis of real-world attacks on smart contracts to gain valuable insights into the methods employed by attackers and the potential consequences of these attacks. By examining and breaking down these incidents, we aim to offer a clearer understanding of how vulnerabilities are exploited in practice, equipping developers and auditors with the knowledge to proactively mitigate potential threats.
— **Exploration of Defense Mechanisms:** We conduct a rigorous assessment of existing defense mechanisms employed to mitigate smart contract attacks. Through this exploration, we identify areas for improvement and potential new approaches to enhance the security of smart contracts. By analyzing the strengths and weaknesses of current defenses, we aim to contribute to the development of more robust and effective security practices.
— **Evaluation of Vulnerability-Detecting Tools:** We conduct a comprehensive evaluation of 14 representative vulnerability-detecting tools used in smart contract analysis. This evaluation encompasses the accuracy, performance, and effectiveness of each tool. By providing insights into the strengths and weaknesses of these tools, we assist researchers and practitioners in selecting the most suitable tools for identifying vulnerabilities in smart contracts.
— **Benchmark Dataset for Tool Evaluation:** We introduce a comprehensive benchmark dataset, encompassing 100 vulnerable smart contract cases and 10 secure contract instances,

to facilitate rigorous evaluation of vulnerability detection tools. This meticulously curated dataset serves as a standardized reference point for assessing the efficacy of vulnerability detection tools. It enables fair and objective comparisons, allowing researchers and practitioners to assess the capabilities of different tools in a consistent manner.

Through these key contributions, our article aims to serve as a comprehensive resource for researchers, developers, and auditors in the field of smart contract security. We aspire to advance secure smart contract development practices and promote the widespread adoption of secure smart contracts in real-world applications.

The rest of this article is structured as follows. Section 2 provides an overview of smart contract platforms and discusses the methodology used in this survey to ensure a thorough analysis. Section 3 offers a detailed analysis of 40 smart contract vulnerabilities, exploring their root causes and the factors that contribute to their existence. Section 4 examines 8 representative attacks, demonstrating how these vulnerabilities can be exploited in real-world scenarios to highlight the potential consequences. Section 5 discusses various defense methodologies and repair techniques for smart contracts, evaluating their effectiveness in mitigating vulnerabilities and preventing successful attacks. Section 6 focuses on evaluating 14 commonly used tools for detecting smart contract vulnerabilities, assessing their accuracy and performance using a carefully curated benchmarking dataset. Section 7 concludes the article by summarizing the key findings and contributions, and discusses future directions and potential research areas to further advance smart contract security.

2 Overview of Smart Contracts and Survey Methodology

2.1 Smart Contracts

The foundational concept of smart contracts is attributed to Nick Szabo, who pioneered the idea in 1996, predating the creation of blockchain technology. Szabo envisioned smart contracts as computer protocols that enable parties to engage in digitally verifiable and self-executing agreements. These contracts are written in code format and facilitate secure and efficient transactions, reducing costs and expediting execution compared with traditional contracts [111]. However, the development of smart contracts faced challenges in trustless systems until the emergence of Ethereum in 2015 [62]. Ethereum's introduction of a blockchain-based platform specifically designed for executing smart contracts revolutionized the field. Ethereum's success paved the way for the proliferation of other blockchain platforms that also support smart contract development. These platforms include Hyperledger [98], EOSIO [51], Tezos [15], NEO [80], and even Bitcoin, which implemented its own version of smart contracts [13].

Our survey presents a comprehensive comparative analysis of four prominent blockchain platforms: Bitcoin, Ethereum, Hyperledger Fabric, and EOS. Table 2 presents this comparison, highlighting the similarities and differences in terms of smart contract implementation and security across these diverse platforms. The following sections analyze four blockchain platforms, highlighting their unique characteristics and approaches to smart contract technology.

— **Bitcoin**, the first blockchain system, primarily facilitates secure peer-to-peer currency transactions without the need for intermediaries. Although not originally designed to support extensive smart contract functionalities like Ethereum, Bitcoin offers limited capabilities for executing cryptographic protocols. These capabilities include basic operations such as sending/receiving messages, verifying signatures, and searching transactions. Bitcoin smart contracts, often called *script-based contracts*, are primarily written in Script, a simple, stack-based language designed specifically for transactions. Although Bitcoin smart contracts are relatively simple and have limited functionality, they still serve as a valuable

Table 2. Comparative Analysis of Smart Contract Ecosystems Across Major Blockchain Platforms

Feature	**Bitcoin**	**Ethereum**	**Hyperledger Fabric**	**EOSIO**
Language	Script	Solidity	Go, JavaScript, Java	C++, Python, JavaScript
Consensus	PoW	PoW/PoS	PBFT/Raft	DPoS+BFT
Environment	Script-based	EVM	Docker	EOS VM
Cryptocurrency	Bitcoin (BTC)	Ether (ETH)	None	EOS
Scalability	High	High	Low	High
Speed (TPS)	27	30	–	4,000
Confirmation Time	>1000s	<100s	<10s	<10s
Applications	Messaging, signatures, transactions	Finance, supply chain, IoT, medical	Enterprise licensing, finance	Finance, gambling
Security Issues	Informal protocols, poor docs	Code/EVM errors, attacks	Code errors, attacks	Code errors, weak verification
Solutions	Secure protocol design	Vulnerability tools, contract monitoring	Vulnerability tools, upgrades	Vulnerability tools, upgrades

tool for executing basic transactional functions in a trustless and decentralized manner. Researchers have studied Bitcoin's scripting language construction and capabilities, with notable examples including Ivy, BALZAC, SIMPLICIT, and BITML [11, 13, 83].

— **Ethereum** pioneered the deployment of smart contracts on blockchain platforms. It enables developers to create their own smart contracts using Solidity, a Turing-complete programming language. These smart contracts are executed on the **Ethereum Virtual Machine (EVM)**, which provides a runtime environment for their execution. Ethereum's smart contracts have significantly expanded blockchain technology applications, giving rise to DApps that enable trustless, decentralized interactions. Ethereum adopts an account-centered model for smart contracts, contrasting with Bitcoin's **Unspent Transaction Output (UTXO)** model. Ethereum's success with smart contracts has established it as a prominent second-generation blockchain platform, often referred to as *blockchain 2.0* [24]. Solidity is the most widely used programming language for Ethereum smart contracts on Ethereum, specifically designed to compile into EVM bytecode, the low-level representation of smart contracts.

— **Hyperledger Fabric (HF)** is an enterprise-focused blockchain platform operating on a permissioned network. Unlike public blockchains, HF allows participation of trusted organizations through a membership service provider [9]. This approach suits scenarios involving collaboration among multiple business organizations, offering enhanced privacy, scalability, and network control. HF smart contracts, called *chaincode*, offer flexibility in programming language support. Developers can choose from various programming languages such as Golang, Java, and JavaScript to implement their smart contracts, allowing them to leverage their existing expertise and use the language most suitable for their application [62]. Privacy is another key feature of HF smart contracts. Hyperledger Fabric supports private transactions, allowing designated participants to conduct confidential operations without disclosing sensitive information to all network members. HF smart contracts also feature modular design, facilitating easier maintenance and updates [67].

— **EOSIO** introduces innovative features to the field of blockchain technology, e.g., **delegated proof of stake consensus (DPOS)** and updatable smart contracts [51]. DPOS elects a limited number of trusted block producers to validate transactions and create blocks, resulting in high throughput and low latency. EOSIO supports multiple programming languages, including C++, Python, and JavaScript. This multi-language support enables developers to leverage their existing skills and expertise, facilitating smart contract creation and deployment. EOSIO smart contracts excel in handling high transaction throughput, with reported speeds of nearly 4,000 **transactions per second (TPS).** EOSIO emphasizes developer friendliness, providing a web-based **integrated development environment (IDE)** that simplifies smart contract creation and deployment. This makes it suitable for DApps requiring fast, scalable transaction processing, such as social media or gaming applications.

As the use of smart contracts becomes more widespread, ensuring their security has become paramount. In recent years, smart contracts have suffered from an increasing number of security issues, resulting in substantial financial losses and reputational damage. Consequently, it is crucial for both academics and industry professionals to focus on smart contract security and develop effective techniques and tools to safeguard these systems.

2.2 Survey Methodology

The objective of our survey is to provide a comprehensive analysis of different approaches to dealing with vulnerable smart contracts. To achieve this, we have formulated a set of **research questions (RQs)** outlining the scope of our review:

— RQ1: [Vulnerabilities] What are the common vulnerabilities that exist in smart contracts, and how can we classify them based on their characteristics?
— RQ2: [Attacks] How do attackers exploit these vulnerabilities, and what are the consequences of these attacks on smart contracts?
— RQ3: [Defenses] What defense methodologies are available to protect smart contracts against attacks, and how do these defenses mitigate the risks associated with vulnerabilities?
— RQ4: [Effectiveness] How effective are the existing vulnerability analysis tools in identifying and detecting vulnerabilities in Solidity smart contracts, and what are the strengths and limitations of these tools in terms of accuracy, performance, and coverage?
— RQ5: [Tools and Benchmark] How do we select representative vulnerability analysis tools for evaluation, and how can we create a standardized benchmark of vulnerable and correct smart contract cases to evaluate the effectiveness of these tools?

By addressing these research questions, our survey aims to provide comprehensive insights into the vulnerabilities, attacks, defense mechanisms, effectiveness of analysis tools, and the creation of a benchmark for smart contracts. To achieve this, we employed a rigorous methodology to ensure comprehensive coverage and relevance of the selected literature. The process began with defining clear inclusion and exclusion criteria to guide our literature selection. These criteria were based on the relevance to smart contracts, publication date (papers published between 2015 and July 2024 were considered), and the credibility of the publication source.

Our search strategy involved querying major academic databases and conference proceedings, using keywords related to smart contracts. We searched through databases such as IEEE Xplore, ACM Digital Library, Google Scholar, Springer Link, and Scopus. We employed the following keywords and search terms: *"blockchain" AND "smart contracts" AND "vulnerabilities" OR "security" OR "attacks".*

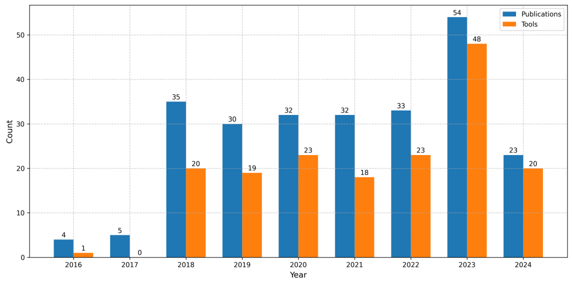

Fig. 1. Publications and tools from 2016 to July 2024.

The screening process generally consists of two stages: a title and abstract screening stage followed by a full-text screening stage [89]. Once we have identified the relevant papers, we assess their quality based on predefined inclusion criteria. This generally involves assessing the relevance, methodology, validity, reliability, credibility, clarity, and impact. The selected papers should address the topic of smart contracts in a meaningful way and provide valuable insights for our investigation. The methodology employed should be suitable for addressing our research questions, transparent, and capable of being replicated. The results should be accurate and consistent, and the conclusions drawn should be well supported by the data. The authors should present their ideas clearly and concisely, with the paper being free of errors and inconsistencies. Furthermore, the paper should have made a significant contribution to smart contracts.

After evaluating these quality attributes, we collected 309 related papers from 72 conference proceedings (including CCS, IEEE S&P, NDSS, USENIX, Euro S&P, Financial Crypto, ASIA CCS, ICSE, and ASE) and 11 journals (including IEEE TSE and IEEE COMST), and relevant preprints. Subsequently, we conducted a systematic data extraction process, focusing on the research questions, methodologies, findings, and conclusions of each paper. The extracted data and our analysis are documented in an online repository (https://github.com/WeiZ-boot/survey-on-smart-contract-vulnerability).

Figure 1 illustrates the trend in smart contract security publications and tools from 2016 to July 2024. The data shows a marked increase in both publications and tool development from 2016 to 2018. The numbers then stabilized between 2018 and 2022. A second notable increase occurred in 2023. This consistent growth in academic publications and tools is driven by advancements in technologies such as AI, the emergence of new vulnerabilities, and the increasing value of smart contracts. These factors present new challenges and opportunities for security research. As smart contracts become more crucial in managing valuable and sensitive operations, the demand for robust security measures intensifies, further fueling research and tool development.

3 Vulnerability in Smart Contracts

This section presents a methodology for identifying and categorizing vulnerabilities in blockchain smart contracts, focusing particularly on Ethereum. Our approach references two main sources: the **Decentralized Application Security Project (DASP)**[1] and the **Smart Contract Weakness Classification (SWC)**[2] Registry. The DASP highlights the top 10 smart contract vulnerabilities, while the SWC Registry details 37 specific vulnerabilities.

We also incorporate insights from related research papers [51, 52, 64, 73, 88, 95] to develop a comprehensive methodology for categorizing the root causes of smart contract vulnerabilities.

[1]https://dasp.co/
[2]https://swcregistry.io/

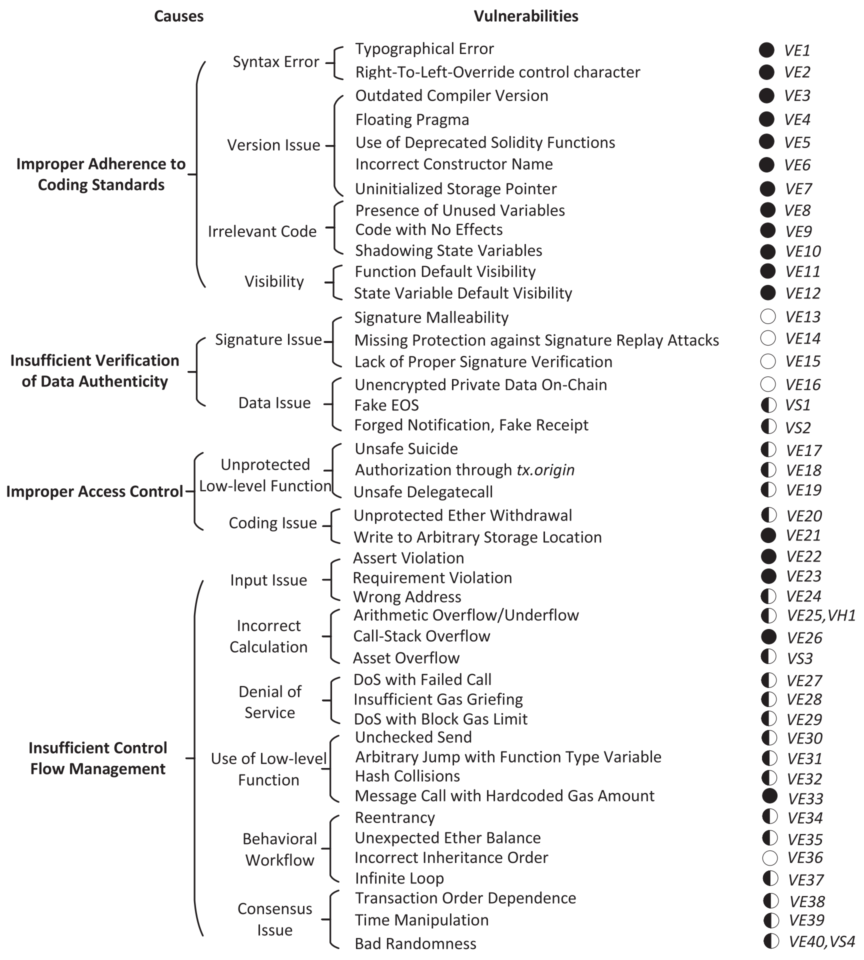
Causes
Vulnerabilities
Improper Adherence to Coding Standards
Syntax Error
Typographical Error ● VE1
Right-To-Left-Override control character ● VE2
Version Issue
Outdated Compiler Version ● VE3
Floating Pragma ● VE4
Use of Deprecated Solidity Functions ● VE5
Incorrect Constructor Name ● VE6
Uninitialized Storage Pointer ● VE7
Irrelevant Code
Presence of Unused Variables ● VE8
Code with No Effects ● VE9
Shadowing State Variables ● VE10
Visibility
Function Default Visibility ● VE11
State Variable Default Visibility ● VE12
Insufficient Verification of Data Authenticity
Signature Issue
Signature Malleability ○ VE13
Missing Protection against Signature Replay Attacks ○ VE14
Lack of Proper Signature Verification ○ VE15
Data Issue
Unencrypted Private Data On-Chain ○ VE16
Fake EOS ◐ VS1
Forged Notification, Fake Receipt ◐ VS2
Improper Access Control
Unprotected Low-level Function
Unsafe Suicide ◐ VE17
Authorization through tx.origin ◐ VE18
Unsafe Delegatecall ◐ VE19
Coding Issue
Unprotected Ether Withdrawal ◐ VE20
Write to Arbitrary Storage Location ● VE21
Insufficient Control Flow Management
Input Issue
Assert Violation ● VE22
Requirement Violation ● VE23
Wrong Address ◐ VE24
Incorrect Calculation
Arithmetic Overflow/Underflow ◐ VE25,VH1
Call-Stack Overflow ● VE26
Asset Overflow ◐ VS3
Denial of Service
DoS with Failed Call ◐ VE27
Insufficient Gas Griefing ◐ VE28
DoS with Block Gas Limit ◐ VE29
Use of Low-level Function
Unchecked Send ◐ VE30
Arbitrary Jump with Function Type Variable ◐ VE31
Hash Collisions ◐ VE32
Message Call with Hardcoded Gas Amount ● VE33
Behavioral Workflow
Reentrancy ◐ VE34
Unexpected Ether Balance ◐ VE35
Incorrect Inheritance Order ○ VE36
Infinite Loop ◐ VE37
Consensus Issue
Transaction Order Dependence ◐ VE38
Time Manipulation ◐ VE39
Bad Randomness ◐ VE40,VS4

Fig. 2. A classification of smart contract vulnerabilities and their causes. A filled circle means the vulnerability has already been solved, an empty circle means the vulnerability is widely discussed (not solved), and a half-filled circle means the vulnerability has been fixed by specific approaches or tools. *VE* means vulnerabilities of Ethereum, *VH* means vulnerabilities of HF, *VS* means vulnerabilities of ESOIO.

This methodology is based on **Common Weakness Enumeration (CWE)** rules and identifies four primary root causes: coding standards, data authenticity, access control, and control flow management.

To illustrate our classification framework, Figure 2 provides a visual representation of smart contract vulnerabilities based on their root causes and corresponding secondary causes, which links 14 secondary causes to 40 specific vulnerabilities found in Ethereum, Hyperledger Fabric, and EOSIO. The figure also shows the status of each vulnerability, indicating whether it has been resolved, can be mitigated, or remains unsolved. Our study does not cover Bitcoin smart contract security, as Bitcoin's scripting language is limited and lacks a unified smart contract language, making typical vulnerabilities less relevant.

Recognizing the inconsistencies in the naming and definition of vulnerabilities across various studies, we have taken measures to ensure clarity and uniformity in our analysis. As part of our research, we have standardized the names of these vulnerabilities. Additionally, we provide comprehensive definitions for each vulnerability, aiming to enhance understanding and foster future research and analysis in this domain.

3.1 Improper Adherence to Coding Standards

This weakness occurs when a smart contract is not developed following established coding rules and best practices. It often results from the relative newness of smart contract programming languages, leading to a lack of experienced developers. Additionally, developers may lack sufficient knowledge of language-specific coding standards, leading to errors and vulnerabilities. Improper adherence to coding standards can manifest in various forms, as follows.

3.1.1 Syntax Errors (VE1, VE2). Syntax errors occur when the code breaks the syntax rules of the programming language, such as spelling or punctuation mistakes. Two examples are *typographical error (VE1)* and *Right-To-Left-Override control character (VE2). VE1* involves using an incorrect operator in the code. *VE2* involves the misuse of the U+202E unicode character. Both of these vulnerabilities can be mitigated by following best practices and employing preventive measures. For *VE1*, pre-condition checks and thorough code reviews can help identify and correct typographical errors, ensuring that the code functions correctly. The research of Khajwal et al. [61] supports the effectiveness of pre-conditions in spotting such errors. Regarding *VE2*, using reliable libraries such as SafeMath can help ensure secure mathematical operations, reducing the risk of these types of errors.

3.1.2 Version Issues (VE3, VE4, VE5, VE6, VE7). Version issues in smart contracts can occur due to rapid updates in smart contract technology, including changes in compiler versions. Writing code with outdated or deprecated functions in a new compiler version can lead to unexpected behaviors and vulnerabilities. This category includes five specific flaws: *outdated compiler version (VE3)*, *Floating pragma (VE4)*, *use of deprecated Solidity functions (VE5)*, *incorrect constructor name (VE6)*, and *uninitialized storage pointer (VE7)*. To avoid these issues, it is essential to stay updated with the latest compiler version to ensure compatibility with new features, bug fixes, and security improvements. Developers should regularly review and update their code to use the correct functions, constructors, and storage initialization techniques as per the latest compiler guidelines.

3.1.3 Irrelevant Code (VE8, VE9, VE10). Irrelevant code in smart contracts refers to code that is not necessary for the contract's execution or functionality. While it might not directly affect the contract's correctness, it can introduce security vulnerabilities or make them harder to detect. This category includes three specific flaws: *presence of unused variables (VE8)*, *code with no effects (VE9)*, and *shadowing state variables (VE10)*. To mitigate these issues, developers should thoroughly test and review the contract's functionality and behavior before deployment. This includes ensuring that the contract works as intended and does not contain irrelevant or unused code.

3.1.4 Visibility (VE11, VE12). In Solidity, visibility labels such as 'public', 'external', 'private', and 'internal' control access to functions and variables in smart contracts. By default, Solidity sets visibility to public, meaning if no visibility is specified, functions and variables are accessible to everyone. Failing to set the correct visibility can lead to two vulnerabilities: *function default visibility (VE11)* and *state variable default visibility (VE12)*. To avoid these risks, developers should carefully assign the appropriate visibility to each function and variable. Implementing pre-condition checks is also essential to ensure that only authorized parties can access critical functions or variables.

3.2 Insufficient Verification of Data Authenticity

This weakness occurs when systems fail to verify the origin or authenticity of data properly, allowing attackers to manipulate or access sensitive information. This can lead to various security issues, including *cryptographic signatures* and *cryptographic data.*

3.2.1 Cryptographic Signatures (VE13, VE14, VE15). Cryptographic signatures are essential for validating data authenticity and integrity in blockchain systems. Ethereum and Bitcoin commonly use the **Elliptic Curve Digital Signature Algorithm (ECDSA)** for this purpose. However, vulnerabilities can arise, including *signature malleability (VE13)*, *missing protection against signature replay attacks (VE14)*, and *lack of proper signature verification (VE15)*. To mitigate these vulnerabilities, robust signature verification mechanisms are critical. In Ethereum, the built-in function 'ecrecover()' is useful for verifying ECDSA signatures.

3.2.2 Cryptographic Data (VE16, VS1, VS2). Even when data is marked as 'private', it can still be accessed by unauthorized parties due to the transparency of blockchain transactions. This can lead to significant financial losses. Vulnerabilities in cryptographic data include *unencrypted private data on-chain (VE16)*, *fake EOS (VS1)*, and *forged notification, fake receipt (VS2)*. To address these vulnerabilities, sensitive data should be encrypted before being stored on-chain to ensure its security. Developers should also be cautious when dealing with private data in contracts, conducting thorough testing to minimize the risk of exploitation by malicious actors.

3.3 Improper Access Control

Improper access control occurs when unauthorized users gain access to a contract and can perform actions they should not be allowed to. These vulnerabilities can lead to significant issues, including financial losses and other negative consequences. Improper access control manifests in two main forms: *unprotected low-level function* and *coding issues*.

3.3.1 Unprotected Low-Level Function (VE17, VE18, VE19). Solidity's low-level functions, such as 'SELFDESTRUCT', 'tx.origin', and 'DELEGATECALL', provide powerful capabilities but can be risky if not used carefully. The following vulnerabilities are associated with unprotected low-level functions:

— *Unsafe suicide (VE17)*: The 'SELFDESTRUCT' function allows a contract to be removed from the blockchain, sending any remaining Ether to a designated address. However, if Ether is sent to a contract that has self-destructed, the funds are permanently lost. Developers should use 'SELFDESTRUCT' with caution, carefully considering the variables and conditions.

— *Authorization through tx.origin (VE18)*: The 'tx.origin' variable represents the address that initiated a transaction, while msg.sender represents the immediate function invoker. Relying on 'tx.origin' for authorization can be dangerous because it can result in funds being sent to the wrong address when one contract calls another. To prevent this, it is recommended to use 'msg.sender' instead of 'tx.origin' for authorization checks.

— *Unsafe delegatecall (VE19)*: The 'DELEGATECALL' instruction allows third-party code to execute within the context of the current contract. This can be exploited, especially in proxy contracts, if an attacker manipulates the address used in 'DELEGATECALL'. This could lead to unauthorized actions such as fund theft or contract destruction. Developers should be cautious when using 'DELEGATECALL' to prevent such exploits.

3.3.2 Coding Issues (VE20, VE21). Exposing certain functions unintentionally can allow malicious parties to withdraw Ether from the contract or manipulate its storage. Two common coding flaws are *unprotected Ether withdrawal (VE20)* and *write to arbitrary storage location (VE21)*. *VE20* occurs when a function intended to be a constructor, is wrongly named. This mistake allows anyone to re-initialize the contract, potentially leading to unauthorized Ether withdrawals. In *VE21*, attackers can write to sensitive storage locations, overwriting the original content and altering the contract's behavior. These flaws can be avoided by carefully designing and structuring the code.

3.4 Insufficient Control Flow Management

This weakness arises when attackers exploit the openness of public blockchains to manipulate the execution of a program in unexpected ways. This can manifest in several forms:

3.4.1 Improper Input (VE22, VE23, VE24). Improper input handling in the EVM can lead to several issues: *assert violation (VE22)*, *requirement violation (VE23)*, and *wrong address (VE24)*. *VE22* occurs when the 'assert()' function fails due to improper input. *VE23* happens when the require() function is used with incorrect conditions. In *VE24*, if a contract address is not 40 hexadecimal characters long, Ethereum may automatically register a new, unowned address, making any Ether sent to it inaccessible. The 'assert()' and 'require()' functions are intended to enhance code readability, but they require strong logical conditions. Improper use can cause errors. Additionally, the length and format of contract addresses should be carefully validated.

3.4.2 Incorrect Calculation (VE25, VH1, VE26, VS3). Incorrect calculations in smart contracts can lead to serious security issues, including arbitrary code execution. Common vulnerabilities include:

— *Arithmetic Overflow/Underflow (VE25, VH1)*: This is a widespread error in both Ethereum and Hyperledger Fabric (HF). It occurs when an arithmetic operation exceeds the maximum or minimum value of an integer, causing the result to wrap around unexpectedly. In Ethereum, this issue arises from the EVM's lack of automatic checks for arithmetic correctness. For instance, when an addition operation surpasses the maximum value representable by an integer type, it wraps around to a lower value without triggering an error. Research has shown that over 42,000 contracts, especially ERC-20 Token contracts, are vulnerable to this issue [118]. To prevent this, using libraries such as SafeMath can reduce overflow and underflow issues.
— *Call-Stack Overflow (VE26)*: This vulnerability occurs when the EVM's call stack, which allows up to 1,024 nested function calls, is exceeded. An attacker can exploit this by repeatedly invoking functions, causing the call stack to overflow. Once the stack reaches its maximum depth, subsequent operations, such as sending Ether, will fail.
— *Asset Overflow (VS3)*: Specific to the EOSIO blockchain, this occurs when there's an overflow in the asset type, which represents token balances and other asset values on EOSIO.

3.4.3 Denial of Service (VE27, VE28, VE29). **Denial of Service (DoS)** vulnerabilities in smart contracts can lead to severe consequences, such as contract lock-ups or freezing of funds. DoS attacks can be carried out primarily through failed calls and excessive gas consumption.

— *DoS with Failed Call (VE27)*: This vulnerability occurs when an external call within a contract fails, whether accidentally or deliberately. It is particularly problematic in payment scenarios in which multiple calls are executed in a single transaction. A failed call can disrupt the contract's flow, leading to unintended consequences.
— *Insufficient Gas Griefing (VE28)*: This issue arises when there is not enough gas to complete an external call, causing the transaction to revert.
— *DoS with Block Gas Limit (VE29)*: Also known as *DoS with Unbounded Operations*, this vulnerability occurs when operations within a contract consume more gas than the block gas limit, preventing the contract from completing its intended tasks.

3.4.4 Use of Low-Level Function (VE30, VE31, VE32, VE33). Solidity's low-level functions, such as 'call', 'transfer', 'send', 'mstore', and 'abi.encodePacked', offer flexibility but can introduce vulnerabilities if not used carefully.

- *Unchecked Send (VE30)*: This vulnerability, also known as 'unhandled exceptions' or 'unchecked low-level call', occurs when developers assume a function will never fail. If an attacker forces the function to fail, it can leave the contract in an unexpected state. Often, developers include checks for success but fail to handle exceptions properly, potentially preventing funds from reaching their intended recipient. This issue arises from inconsistent exception handling in Solidity.
- *Arbitrary Jump with Function Type Variable (VE31)*: The 'mstore' function can be manipulated by an attacker to point a function type variable to any code instruction, allowing arbitrary code execution and compromising the contract's integrity.
- *Hash Collisions (VE32)*: Incorrect use of 'abi.encodePacked' with multiple variable-length arguments can lead to hash collisions, in which different inputs produce the same hash, potentially leading to vulnerabilities.
- *Message Call with Hardcoded Gas Amount (VE33)*: Using 'transfer' and 'send' functions with hardcoded gas amounts can lead to issues when gas costs change, such as during network upgrades or hard forks.

3.4.5 Improper Behavioral Workflow (VE34, VE35, VE36, VE37). These vulnerabilities occur when the expected sequence of operations within a smart contract is manipulated by malicious users, leading to unexpected states or undesired behaviors. These vulnerabilities include:

- *Reentrancy (VE34)*: A malicious contract can repeatedly call back into the calling contract before the initial function execution is complete, creating a loop. This allows the attacker to manipulate the contract's execution flow and exploit vulnerabilities. It is essential to secure contract interactions to prevent reentrancy attacks and maintain the contract's integrity.
- *Unexpected Ether Balance (VE35)*: Attackers can intentionally send funds to a contract in a way that disrupts its intended behavior, potentially causing a DoS condition or making the contract unusable by affecting its Ether balance.
- *Incorrect Inheritance Order (VE36)*: In Solidity, the order in which contracts inherit from one another can impact their behavior. Improper ordering can be exploited by attackers to achieve unexpected outcomes, potentially leading to vulnerabilities.
- *Infinite Loop (VE37)*: This occurs when a contract enters a loop that never terminates, often due to incorrect invocation of the fallback function. The loop can exhaust the contract's gas, disrupting its functionality and leading to non-termination of execution.

3.4.6 Consensus Issues (VE38, VE39, VE40, VS4). In blockchain systems, synchronization of blocks relies on consensus protocols like Proof of Work or Proof of Stake. While these protocols are essential for agreeing on which transactions to include in blocks, they can introduce vulnerabilities that attackers may exploit. Key vulnerabilities include:

- *Transaction Order Dependency (TOD) (VE38)*: Also known as frontrunning, this vulnerability arises from the prioritization of transactions in blockchain blocks. Miners can choose which transactions to include and in what order, often based on gas price. A malicious miner who sees and reacts to transactions before they are mined can manipulate the order to their advantage, causing undesirable outcomes or financial losses for users.
- *Time Manipulation (VE39)*: This occurs when smart contracts rely on block timestamps for certain functions. In Solidity, timestamps can be accessed using block.timestamp or now, but these values can be manipulated by miners. If a contract's functionality depends on the timestamp, miners can choose a timestamp that manipulates the contract's behavior for profit.
- *Bad Randomness (VE40, VS4)*: This vulnerability involves flaws in generating random numbers within smart contracts. Random numbers are often used for decision-making or

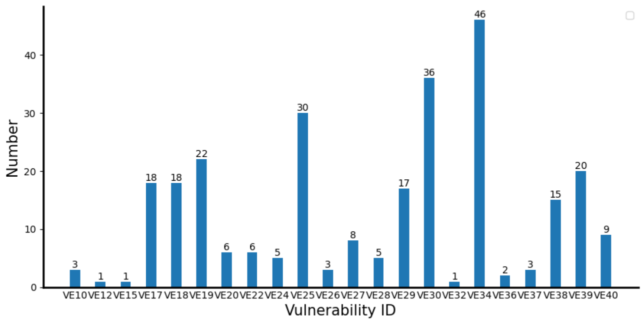

Fig. 3. Vulnerability frequency statistics.

determining outcomes. If the random number generation process is predictable, attackers can exploit it. For example, if a predictable seed value is used, an attacker could determine the seed, predict the random numbers, and manipulate the contract's outcome.

3.5 Common Vulnerability Ranking

Beyond the DASP list, we have created a new list of the top 10 smart contract vulnerabilities that pose significant risks to contract security and functionality. This list is based on the frequency of occurrence across various analysis tools. Figure 3 shows statistics on 22 vulnerability categories. The top 10 vulnerabilities are *reentrancy (VE34)*, *arithmetic overflow/underflow (VE25)*, *DoS with block gas limit (VE29)*, *unsafe suicidal (VE17)*, *unsafe delegatecall (VE19)*, *unchecked send (VE30)*, *TOD (VE38)*, *time manipulation (VE39)*, and *authorization through tx.origin (VE18)*. These vulnerabilities are the most common and high-risk issues that smart contract auditors should prioritize.

In this section, we conducted a comprehensive analysis of the root causes of smart contract vulnerabilities, introduced a novel classification system, and ranked the most frequently encountered vulnerabilities based on existing research, providing clear and conclusive answers to our research question, **RQ1**, as outlined in Section 2.2. By understanding these vulnerabilities and their rankings, developers can prioritize their efforts on the most critical security concerns. Additionally, exploring the common types of attacks that can exploit these vulnerabilities helps identify potential attack vectors, enabling developers to proactively implement robust measures to mitigate risks and enhance the overall security of smart contracts.

4 Attacks on Smart Contracts

A recent survey [132] reveals a rapid increase in the number of Solidity contracts over the past 5 years. This growth reflects the expanding range of smart contract applications across sectors such as DeFi [14], insurance, and lending platforms. Unfortunately, this growth has also led to an increase in the number of attackers exploiting vulnerabilities in smart contracts. Consequently, several high-profile attacks have occurred, resulting in substantial financial losses.

The DAO attack in 2016 stands out as a significant incident, resulting in the loss of millions of dollars worth of Ether from the organization. Figure 4 illustrates a series of high-profile attacks on smart contracts from 2016 to 2024. The data reveals an increasing frequency of high-value attacks over time, coupled with a rise in contract values. To better understand these attacks, we analyzed and identified eight key attack patterns, as highlighted in Figure 5. This figure delineates the major application domains of smart contracts and their associated attack patterns. We also investigated the vulnerabilities underlying these attacks. Classifying vulnerabilities according to established attack patterns enables the identification of critical weaknesses requiring attention.

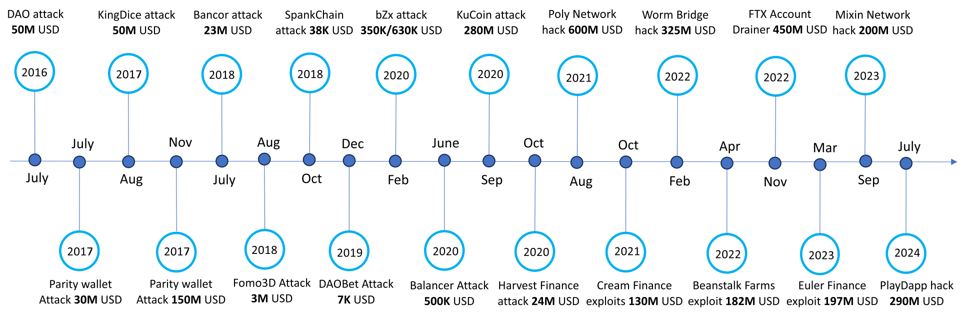

Fig. 4. Several high-profile attacks from 2016 to 2024.

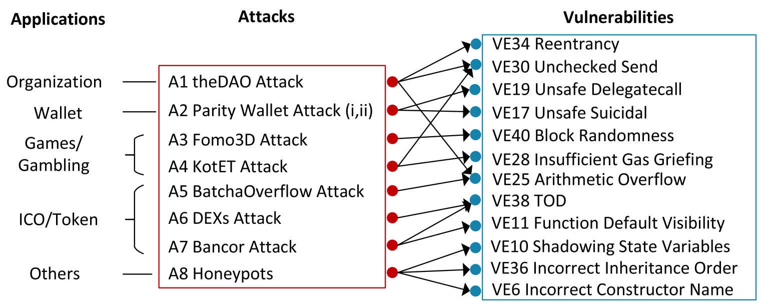

Fig. 5. The relationships between attacks and vulnerabilities.

4.1 The DAO Attack (A1)

The DAO (Decentralized Autonomous Organization) was a pioneering project launched on the Ethereum blockchain in 2016, aimed at creating a decentralized venture capital fund in which participants could invest and vote via smart contracts. The DAO quickly gained attention, raising over $150 million USD in funding. However, in June 2016, the project was compromised due to a critical security vulnerability in its smart contract code. An attacker exploited this flaw to drain approximately $60 million USD worth of Ether from the organization [10].

This incident, known as the DAO attack, was a major setback for the Ethereum community. It sparked a significant debate about the immutability of blockchain transactions and whether a hard fork should be implemented to recover the stolen funds. Ultimately, the Ethereum community decided to implement a hard fork, creating a new version of the blockchain that reversed the effects of the hack and returned the stolen funds to their rightful owners.

Following the DAO hack, other smart contracts, such as Spankchain and Lendf.me, also suffered losses due to similar security vulnerabilities. Atzei et al. [10] provided examples of attack contracts named Mallory and Mallory2. In Mallory, the attacker manipulates the control flow and exploits reentrancy (VE34). In Mallory2, the attacker efficiently exploits arithmetic overflow/underflow (VE25) and unchecked send (VE30) vulnerabilities using only two calls.

4.2 Parity Wallet Attack (A2)

The Parity Wallet attack refers to two separate incidents in 2017, in which vulnerabilities in the Parity Multisig Wallet led to significant financial losses. The Parity Wallet consists of two components: a library contract containing essential wallet functions and wallet contracts that act as proxies, delegating calls to the library contract via the delegatecall mechanism. In both incidents, vulnerabilities allowed attackers to gain unauthorized control over the wallets.

- First Incident: The attacker exploited the unsafe delegatecall (VE19) vulnerability, draining over $30 million USD worth of Ether [85]. The attacker executed two transactions–the first to gain ownership of the victim's wallet contract and the second to drain the funds.
- Second Incident: The attacker exploited the suicide (VE17) vulnerability, locking more than $280 million USD worth of funds. In this case, the attacker used the delegatecall mechanism to self-initialize as the owner of the wallet contract, similar to the first incident.

4.3 Fomo3D Attacks (A3)

The Fomo3D contract was an Ethereum game in which participants could purchase keys using Ether and refer others to the game to earn more Ether. The objective of the game was to be the last participant to purchase a key before the timer expired, thereby winning the entire pot of Ether. The attacker purchased a ticket and then sent multiple transactions with high gas prices in rapid succession, effectively consuming a significant portion of the block's gas limit. This action caused other transactions related to Fomo3D, including the key purchases made by other participants, to be delayed or stuck in a pending state until the timer ran out. The attackers gained an advantage by exploiting two vulnerabilities: bad randomness (VE40) and DoS with block gas limit (V29).

4.4 KotET Attack (A4)

The King of the Ether Throne (KotET) was a game on the Ethereum blockchain in which participants competed to win the throne and claim all the Ether held in the contract. In February 2016, the KotET contract was exploited through two vulnerabilities: unchecked send (VE30) and insufficient gas griefing (VE28).

In the attack, when the KotET contract attempted to transfer funds to another wallet contract, both contracts required sufficient gas to complete the transaction [84]. If the wallet contract had insufficient gas, the payment would fail, and the funds would be returned to the KotET contract. However, the KotET contract did not recognize the payment failure. As a result, the latest player was still crowned as King, whereas the previous player did not receive the intended compensation.

4.5 BatchOverflow Attack (A5)

The BatchOverflow attack, which occurred in April 2018, targeted the **Beauty Ecosystem Coin (BEC)** token. The attacker exploited an arithmetic overflow/underflow (VE25) vulnerability to increase digital assets without authorization. This led to the theft of BEC tokens and a temporary shutdown of the exchange platform.

According to the blockchain security firm PeckShield, this vulnerability was not limited to the BEC token. They identified similar integer overflow vulnerabilities in about 12 other token smart contracts, including SMT (proxyOverflow), UET (transferFlow), SCA (multiOverflow), HXG (burnOverflow), and others.

4.6 Frontrunning Attack (A6, A7)

The DEXs and Bancor attacks are examples of Frontrunning attacks, a practice also seen in traditional financial markets in which transactions are manipulated based on prior knowledge [115]. In blockchain, every transaction is visible in the pending pool before being included in a block. Miners, who prioritize transactions with higher gas prices, enable attackers to manipulate transaction orders for profit. Frontrunning attacks exploit the TOD (VE38) vulnerability. Eskandari et al. [36] categorize these attacks into three types: displacement, insertion, and suppression.

4.6.1 DEXs Attack (A6). **Decentralized exchanges (DEXs)** are platforms built on smart contracts that allow users to exchange ERC-20 tokens for Ether or other tokens. Frontrunning attacks have become a significant concern in DEXs, exploiting the TOD vulnerability to manipulate transactions for unfair trading advantages.

In a typical attack, a bot detects a profitable transaction and submits its own transaction with higher gas fees to ensure thatit executes before the victim's transaction. The attacker's transaction is designed to capitalize on the expected price movement caused by the victim's trade.

For example, on the PancakeSwap DEX, an attacker's bot might observe a buyer trying to purchase 1 Corgicoin token. The bot quickly buys the token itself, then sells it immediately after the victim's transaction, maximizing profits from the price movement triggered by the victim's purchase. These attacks are carried out by highly competitive bots aiming to extract **Miner's Extractable Value (MEV)**. According to Flashbots [39], searchers extracted approximately $691 million USD worth of value from Ethereum in January 2023 alone.

4.6.2 Bancor Event (A7). Bancor is a decentralized exchange platform that allows users to create and trade their own tokens. During an audit of its exchange smart contract, two vulnerabilities were discovered: TOD (VE38) and Function Default Visibility (VE11). An attacker could have exploited the TOD vulnerability to execute transactions ahead of others, potentially obtaining $135,229 USD worth of Ether [126]. Fortunately, no real-world attack occurred, but the vulnerability was identified and addressed before any damage was done.

4.7 Honeypots (A8)

In the context of smart contracts, honeypots are fraudulent schemes that exploit security vulnerabilities while using deceptive tactics. The idea is to create a smart contract that appears to have an obvious flaw or vulnerability, enticing potential victims to exploit it for financial gain. However, the contract is intentionally designed so that anyone attempting to take advantage of the apparent flaw ends up losing their funds instead.

Honeypots prey on human greed and the desire for quick profits, luring unsuspecting users into a trap set by malicious actors. According to Torres et al. [119], honeypots often exploit specific vulnerabilities such as incorrect constructor name (VE6), shadowing state variables (VE10), and incorrect inheritance order (VE36).

In this section, we systematically examined eight common attack patterns and their associated vulnerabilities, addressing **RQ2**, as outlined in Section 2.2. This question focuses on how these attacks exploit vulnerabilities and the consequences of such attacks.

Our analysis of past attacks offers valuable insights into common vulnerabilities, attack patterns, and emerging trends in smart contract security, as follows.

— **Increasing Complexity of Smart Contract Attacks**: As smart contracts become more sophisticated, attackers have developed increasingly complex techniques, often exploiting multiple vulnerabilities or the interactions between different smart contracts simultaneously.
— **Rise in Attacks Targeting DeFi Applications**: DeFi platforms, with their high liquidity and rapid growth, have become prime targets for attackers. Vulnerabilities in these platforms can lead to significant financial losses, as seen in several high-profile incidents in recent years.
— **Potential Advent of AI-Enabled Attackers**: AI-enabled attacks could rapidly identify and exploit vulnerabilities in smart contracts before they are detected by developers or security teams. AI systems have the potential to analyze large amounts of code at unprecedented speeds, identifying patterns and weaknesses that human analysis might miss and adapting to changing security environments.

These insights are crucial for smart contract developers aiming to enhance contract security. While completely eliminating all attacks may be challenging, implementing best practices and

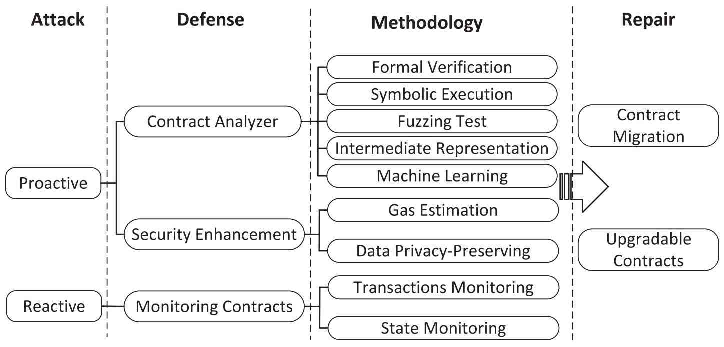

Fig. 6. Research ideas for the defense of smart contracts.

security measures can significantly reduce the risks of exploitation. Building on our understanding of attack patterns, we will explore various defense methodologies in the next section.

5 Defense Methodologies

Defense technologies evolve alongside the advancement of attacks, and a substantial body of work has been dedicated to smart contract security measures [24, 115, 133]. Figure 6 presents an overview of a comprehensive set of research solutions for smart contract defense. Defense strategies for smart contracts generally fall into two main categories: proactive and reactive.

- **Proactive Defense**: These strategies focus on preventing attacks before they occur by addressing known vulnerabilities. This includes using contract analyzers and implementing security enhancements to prevent or mitigate attacks that have been previously identified and studied.
- **Reactive Defense**: These strategies respond to attacks after they have happened, particularly those that were not previously identified or studied, and for which no specific defense mechanisms exist. Monitoring contracts plays a crucial role in detecting and responding to such attacks.

However, identifying vulnerabilities and deploying monitoring contracts alone is not enough for effective smart contract defense. Repairing vulnerable smart contracts is also a critical component of the defense process [97, 117]. While defense strategies encompass a wide range of measures, repairing vulnerabilities is a specific action to fix and address them.

5.1 Contract Analyzer

Contract analyzers play a crucial role in reducing the risk of vulnerabilities in smart contracts before their deployment. Researchers employ various methodologies to analyze smart contracts, many of which are publicly available under open-source licenses. There are five common methodologies for smart contract analysis: formal verification, symbolic execution, fuzzing, intermediate representation, and machine learning.

5.1.1 Formal Verification. Formal verification is a mathematical technique used to build and check formal proofs that ensure that software behaves according to its specifications and requirements, even in large and complex state spaces. This technique is particularly useful for smart contracts, which are often written in languages such as Solidity, designed to be amenable to formal verification [48, 57]. There are two main methods of formal verification in smart contracts: model checking and theorem proving.

- **Model Checking**: This method systematically examines all possible states of a smart contract to confirm whether it meets certain characteristics. Model checking is especially effective for smart contracts because they often have a finite number of states and are designed to be deterministic. This allows model checkers to accurately capture and verify the behavior of smart contracts early in the design process [79]. Tools such as SIPN [12], FDR [93], and NuSMV [79] are commonly used for this purpose. Model checking is particularly suited for detecting specific vulnerabilities, such as buffer overflows and integer overflows.
- **Theorem Proving**: This technique uses mathematical logic to describe desired properties of a system and then employs a theorem prover to generate proofs that verify these properties. Theorem proving is used to ensure that a smart contract meets specific requirements, such as correctness, safety, and liveness. Unlike model checking, theorem proving can handle the verification of systems with infinite states, making it ideal for analyzing complex smart contracts. Tools such as Coq [15] and FEther [125] support theorem proving for smart contracts. For example, Amani et al. [7] extended the EVM definition into Isabelle/HOL, allowing for formal verification of EVM-based contracts. However, theorem proving is a semi-automated process that often requires manual interaction and is typically used to detect broader classes of vulnerabilities, such as logic errors and design flaws.

5.1.2 Symbolic Execution. It systematically explores more possible execution paths simultaneously to trigger deep program errors. This approach does not require a specified input value; rather, it abstracts the input values into symbols.

From the vulnerability detection perspective, symbolic execution offers developers specific input to the triggered vulnerability, which can be used to confirm or debug. Symbolic execution has the advantage of achieving high test coverage with as few test cases as possible, thereby digging out deep program errors. Symbolic execution is often combined with constraint solving to reason whether an execution path is reachable. However, when the program is large and complex, symbolic execution will generate too many paths, which may lead to path explosion. A smart contract can be a maximum of 24 KB or it will run out of gas [19]. Therefore, symbolic execution is perhaps the most popular approach for smart contracts. Z3 SMT solvers are used to check which paths are flexible.

Oyente [72] was the first attempt for smart contract verification, using a **Control Flow Graph (CFG)** representation of the bytecode to identify vulnerabilities such as *transaction-ordering dependence*, *timestamp dependence*, *mishandled exceptions*, and *reentrancy*. Other tools, such as Manticore and Maian, have extended the capabilities of symbolic execution to detect additional vulnerabilities such as *arithmetic overflow/underflow*, *unsafe suicide*, and *unchecked send*. However, scalability is a concern with symbolic execution, especially in complex application scenarios. The exploration of deep program paths can be slow and resource intensive.

5.1.3 Fuzzing. Fuzzing is a software testing technique that involves executing target programs with a large number of abnormal or random test cases to detect vulnerabilities. It has gained significant attention in both industry and academia [50, 68] due to its simplicity and practical effectiveness in identifying software vulnerabilities. Major software vendors, such as Google [101] and Microsoft [76], employ fuzzing techniques to uncover vulnerabilities in their products.

In the context of smart contracts, fuzzing has been utilized as a means of vulnerability detection, although there are relatively fewer works specifically focused on smart contract fuzzing in recent years. ContractFuzzer [56], for example, uses the **Application Binary Interface (ABI)** specification of contracts as input for fuzzing to detect vulnerabilities. It relies on user-provided input seeds. Echidna [44], on the other hand, uses falsified user-defined predicates or Solidity assertions as input seeds, which are then subjected to grammar-based fuzzing to detect vulnerabilities.

Harvey [124] incorporates a prediction component to generate new input seeds for gray box fuzzing. sFUZZ [81] adopts an adaptive strategy to select input seeds, which are then fed into the prominent fuzzer **American Fuzzy Lop (AFL)**. However, these methods are more effective in finding shallow bugs and less effective in finding bugs that lie deep in the execution flow.

This limitation is due to the heavy reliance on input seeds in fuzzing. An alternative approach that has shown promising results in traditional programs is hybrid fuzzing, which combines fuzzing with symbolic execution. **The Imitation Learning Fuzzer (ILF)** [50] addresses this limitation by using a symbolic execution expert to generate a large number of training sequences, which are then fed into imitation learning prior to fuzzing. The ILF achieves improved coverage and performs well on both large and small contracts. However, the ILF is limited to the contracts used for imitation learning in its training phase. ConFuzzius [116], on the other hand, leverages lightweight symbolic execution to analyze execution traces and employs a constraint solver to obtain input seeds for the fuzzer. This approach enhances the effectiveness of fuzzing by incorporating symbolic execution-based analysis. By combining fuzzing with symbolic execution or other complementary techniques, researchers aim to enhance the effectiveness of vulnerability detection in smart contracts, addressing both shallow and deep vulnerabilities.

5.1.4 Intermediate Representation. To accurately analyze smart contracts, some researchers are also exploring converting contracts into an **intermediate representation (IR)** with highly semantic information, which is more suitable for the analysis and detection of common security issues. Different from formal verification, IR relies on semantic-based transformation. The analysis process can be divided into four stages: lexical analysis, syntax analysis, semantic analysis, and transformation. The lexical analysis uses Scanner to check whether the input code is a combination of several legitimate words; the syntax analyzer checks whether the combination of these legitimate words meets grammatical rules; semantic analysis checks whether semantics are reasonable; and the transformer converts source code or bytecode into machine code, such as XML. Then, the analyzer detects vulnerabilities through specific methods.

Slither [37] transfers contracts to its internal representation language (SlithIR) which uses the **Static Single Assignment (SSA)** form to facilitate the computation of code analyses. EthIR [6], based on Oyente, translates CFGs to a **rule-based representation (RBR)** of the bytecode. Smartcheck [113] directly translates source code into an XML-based IR and then checks it against XPath patterns. MadMax [43], based on the Vandal [18] decompiler, translates EVM bytecode to a structured IR to check gas-related vulnerabilities. NeuCheck [71] employs the Solidity parser ANTLR to complete the transformation from source code to an IR (XML parse tree).

However, IR analysis faces two challenges: (1) Because of semantic heterogeneity, it is unavoidable to produce semantic missing during the security analysis; and (2) IR analysis requires more processing time compared with other methodologies.

5.1.5 Machine Learning. Machine learning demonstrated significant potential in program security, often outperforming traditional methods in various aspects [21, 47, 105]. Unlike traditional methods, machine learning combines static analysis and dynamic detection. This combination addresses the high false-negative rate of static analysis and the low code coverage of dynamic analysis. Furthermore, machine learning exhibits excellent scalability and adaptability to novel vulnerabilities.

Several prior works have employed machine learning techniques to analyze smart contracts. Tann et al. [112] introduced a **long short-term memory (LSTM)** [105] model to handle the semantic representations of smart contract opcode to detect contract security threats. Their model can achieve higher detection accuracy than symbolic execution analyzer Maian [82], when both are based on the same vulnerabilities taxonomy. Qian et al. [92] applied a **bi-directional long**

short-term memory with attention mechanism (BLSTM-ATT) in their sequential model to *reentrancy* detection. This framework converts source code into contract snippets and feeds the sequential model with feature vector representations parsed from these snippets.

Zhuang et al. [134] proposed a **degree-free graph convolutional neural network (DR-GCN)** and a novel **temporal message propagation network (TMP)** for vulnerability detection. In their approach, the source code of a contract is converted to a contracted graph, which is then normalized through a node elimination process. The normalized graphs are fed to a DR-GCN and TMP for vulnerability modeling and detection. ContractWard [121] trains its machine learning model using bigram features extracted from the opcodes of the compiled smart contract.

SmartMixModel [102] extracts high-level syntactic features from source code as well as low-level bytecode features from the smart contract. Then, these features are trained on a machine learning model and deep learning model to detect vulnerabilities. ESCORT [100] introduces a deep learning-based method for vulnerability detection in smart contracts, combined with distinct branches for learning specific features of various vulnerability types. When a new vulnerability type is identified, ESCORT seamlessly integrates a new branch into the existing feature extractor and trains it with minimal data.

In addition to these methods, some researchers suggest that combining machine learning with fuzzing could enhance detection efficiency. SoliAudit [69] employs machine learning to detect known vulnerabilities without the need for expert knowledge or predefined patterns, whereas fuzzing is used to identify potential weaknesses. Although there is no direct correlation between the two methods, fuzzing complements machine learning in vulnerability detection. ILF [50] utilizes an imitation learning model to develop a fuzzer from training sequences, demonstrating another innovative approach to enhancing vulnerability detection through machine learning and fuzzing.

Recent studies have explored the potential of **large language models (LLMs)** in enhancing software security, leveraging their capabilities in code comprehension, generation, and analysis. Studies have demonstrated the effectiveness of LLMs in identifying vulnerabilities, verifying compliance, and assessing logical correctness. Chen et al. [28] demonstrated that LLMs (GPT-2, T5) trained on a high-quality dataset of 18,945 vulnerable C/C++ functions outperformed other machine learning methods, including Graph Neural Networks, in vulnerability prediction.

The application of LLMs to smart contract security has gained significant attention. David et al. [32] investigated the utility of LLMs, such as GPT-4 and Claude, in conducting security audits of DeFi smart contracts. Chen et al. [23] conducted a comparative analysis of GPT's performance in identifying smart contract vulnerabilities against other established tools, utilizing a publicly accessible dataset. Sun et al. [110] tested GPT's ability to match candidate vulnerabilities using a binary response format, in which GPT responds with 'Yes' or 'No' to potential matches with predefined scenarios. They also highlighted potential false positives due to GPT's inherent limitations. Shou et al. [103] integrated the Llama-2 model into the fuzzing process to detect vulnerabilities in smart contracts, aiming to address inefficiencies in traditional fuzzing methods. Sun et al. [109] compared open-source language models such as Mixtral and CodeLlama against GPT-4 for smart contract vulnerability detection. They found that GPT-4, leveraging its advanced Assistants' functionalities, significantly outperformed open-source alternatives.

5.2 Security Enhancement

Before deployment, certain measures can be taken to strengthen the security of smart contracts. Gas costs and data privacy are critical factors. By thoroughly reviewing these aspects prior to deployment, the security of the contract can be improved. Various methods and tools are available for assessing gas costs and ensuring data privacy.

5.2.1 Gas Estimation and Optimization. Gas is one of the most important mechanisms in EVMs for assigning a cost to the execution of an instruction. This mechanism can effectively prevent resource abuse (especially DoS attacks) and avoid "infinite" loops [27]. When issuing a transaction, the sender needs to specify a gas limit and a gas price before submitting it to the network.

Gas represents much more than just the cost of processing transactions on the Ethereum network. It enables smart contracts to run various applications, forming a decentralized web. Thus, while gas could technically be described as "transaction fees," this term should be used with caution. Several gas-related vulnerabilities exist, as discussed in Section 3. If a transaction runs out of gas during execution, the EVM throws an exception, immediately reverting to the state before execution and consuming all the gas provided by the sender.

To prevent running out of gas, many Ethereum wallets, such as Metamask [75], can statically estimate the cost of a transaction before it is executed. However, there are many operations that are difficult to estimate during executions. In order to address this problem, some researchers proposed some methods that estimate gas and optimize smart contracts.

Albert et al. [5] proposed an automatic gas analyzer Gastap that infers gas upper bounds for all its public functions. Gastap requires complex transformation and analysis of the source code that includes several key techniques, including Oyente [72], EthIR [6], Saco [2] and Pubs [3]. Gasol [4], an extension of Gastap, introduces an automatic optimization of the selected function that reduces inefficient gas consumption.

Li et al. [26] developed a GasChecker tool that identifies the gas-inefficient code of smart contracts. They summarized ten gas-inefficient programming patterns that assisted users in better tailoring contracts to avoid gas waste. Gas Gauge [78] can automatically estimate the gas cost for the target function and the loop-bound threshold. In addition, Gas Gauge can find all the loops and furnish the gas-related instances to help developers with suggestions. Li et al. [66] estimated the gas for new transactions by learning the relationship between historical transaction traces and their gas costs.

5.2.2 Data Privacy-Preserving. In addition to gas concerns, another major concern for the smart contract is private data [60]. Since the openness and transparency of public blockchains, the privacy overlay feature on the chain is absent. This not only leads to some security issues but also prevents their wider adoption. It is problematic for applications that handle sensitive data such as voting schemes [74], electronic medical records [70], and crowdsensing [87].

A promising approach to handling private data involves designing new blockchain infrastructures with built-in privacy support [63]. Several proposed blockchain infrastructures, including Hawk [63], Arbitrum [60], Ekiden [29], and Town Crier [127], support private data through trusted third-party mechanisms. Hawk [63] enables programmers to write private smart contracts without specialized knowledge and generates efficient cryptographic protocols for inter-party interactions. Arbitrum [60] facilitates private smart contract creation by emulating a **virtual machine (VM)** and enables honest parties to update the VM state on-chain. Both Hawk and Arbitrum utilize trusted managers, implementable through trusted computing hardware or multi-party computation among users. Ekiden [29] processes smart contracts with private data off-chain in **trusted execution environments (TEEs)**, while ensuring secure on-chain interactions between contracts.

An alternative approach involves using cryptographic primitives, particularly zero-knowledge proofs. Hawk [63] employs zero-knowledge proofs to ensure correct contract execution and maintain money conservation on-chain. Steffen et al. introduced zkay contracts, which incorporate private data protected by **non-interactive zero-knowledge (NIZK)** proofs [106]. For enhanced blockchain executability, zkay contracts are transformed into equivalent contracts that maintain privacy and functionality. Zapper [107] utilizes NIZK proofs for private state updates and employs an oblivious Merkle tree to conceal sender and receiver identities.

5.3 Runtime Monitoring

The ability to deploy smart contracts is a key feature of blockchains. However, once deployed, smart contracts become immutable, meaning any vulnerabilities they contain cannot be changed. While pre-deployment tools exist to scan for vulnerabilities, they often have limited scopes and may miss some issues, leaving smart contracts vulnerable to runtime attacks. To address this, runtime monitoring techniques have been developed to enhance the security of smart contracts after deployment. Runtime monitoring analyzes and monitors the behavior of smart contracts during execution, offering greater coverage and precision in detecting attacks compared with pre-deployment analysis alone [35]. This approach can be categorized into two main types: transaction monitoring and state monitoring.

5.3.1 Transaction Monitoring. As Ethereum can be seen as a transaction-based state machine, a transaction contains much information aiming to change the blockchain state. This transaction information can also be used to detect and prevent attacks.

ECFChecker [45] is the first dynamic detection tool designed to identify transactions that include reentrancy vulnerabilities. It examines transactions to determine whether they exhibit the characteristics of a reentrancy attack, in which a contract can be called recursively before previous invocations have been completed. Sereum [96], on the other hand, aims to prevent reentrancy attacks by employing taint tracking techniques. It tracks the flow of data from storage variables to control-flow decisions, helping identify potential vulnerabilities [25, 53]. Both ECFChecker and Sereum rely on modified versions of the EVM and primarily focus on detecting reentrancy attacks.

In contrast, ÆGIS [38] takes a broader approach by providing an extensible framework for detecting new vulnerabilities in smart contracts. It maintains attack patterns and reverts transactions that match these patterns, thereby enhancing security. The framework allows for the storage and voting of new attack patterns through a smart contract, enabling the community to actively contribute to the detection and prevention of novel attacks.

SODA [25] is based on a modified EVM-based client and provides a platform for developing various applications to detect malicious transactions in real time. It offers flexibility and extensibility in creating online apps for monitoring and identifying potentially harmful transactions. SODA has been integrated into popular blockchains that support the EVM, increasing its accessibility and usability. Horus [131] leverages transaction graph-based analysis to identify the flow of stolen assets. By examining the transaction graph, Horus can trace the movement of assets and detect potential theft or unauthorized transfers.

5.3.2 State Monitoring. In addition to transactions, blockchain state variables can provide valuable information about the real-time status of a smart contract. Monitoring and analyzing the state variables can be an effective approach to detect and prevent attacks.

Solythesis [65] allows users to instrument user-specified invariants into smart contract code. These invariants represent specific conditions that should always hold true during the execution of the contract. By tracking the transactions that violate these invariants, Solythesis can efficiently enforce powerful monitoring of the contract's state. This approach helps identify abnormal or unexpected changes in the state variables and allows for proactive detection of potential vulnerabilities.

ContractGuard [122] takes a similar approach but introduces the concept of an **intrusion detection system (IDS)** to monitor the behavior of a deployed smart contract. The IDS continuously monitors the state variables and looks for abnormal or suspicious behaviors that deviate from expected patterns. If an abnormal state is detected, ContractGuard can roll back all the changes to the contract state and notify the relevant users about the potential intrusion.

5.4 Post-deployment Repair

Ideally, smart contracts should be deployed with the highest possible level of security. However, this presents a challenge not only for smart contracts but also for all software programs. Furthermore, the underlying blockchain technology ensures immutability, meaning that the past cannot be altered. Consequently, updating the code of a deployed contract or addressing vulnerabilities becomes unfeasible. Apart from fixing vulnerabilities, there are numerous other reasons to modify contract code, including adapting business logic or enhancing functionality.

5.4.1 Contract Migration. Contract migration is a process in which a new instance of a contract is deployed and the data from the old contract is transferred to it. This approach has been successfully used in various token migrations, such as those for Augur, VeChainThor, and TRON.

A typical contract migration process involves two main steps: data recovery and data writing [59]. During data recovery, data from the old contract is extracted, including both public and private variables. While retrieving public variables is relatively straightforward, handling private variables and mappings can be more complex, requiring special measures to ensure data integrity and consistency.

To address the challenges of data migration, some developers use separate contracts to store and manage data [8]. This approach simplifies the migration process by separating the logic from data storage. By migrating the logic to a new contract while keeping the data in a separate contract, the migration becomes more manageable. It is important to note that contract migration procedures can be costly, especially for token-based contracts with a large number of accounts [59]. Migrating data for many accounts requires careful planning and execution to avoid potential issues and ensure the accuracy and security of the migrated data.

5.4.2 Upgradable Smart Contracts. Upgradable contracts offer a promising approach to modifying smart contract code by incorporating an upgradability mechanism that allows for seamless upgrades without requiring data migration or updating external references [58].

These contracts are typically implemented using a proxy pattern architecture in which the contract is divided into two parts: one for logic execution and the other for data storage. The logic contract contains all the business logic but does not store any state, whereas the proxy contract holds all funds and internal states without implementing any logic. This separation allows for quick and cost-effective upgrades by simply modifying or replacing the logic contract while keeping the proxy contract intact, preserving all existing data and external references.

Various approaches have been proposed for implementing upgradable contracts. Zeppelin [8] introduced three proxy pattern architectures: inherited storage, eternal storage, and unstructured storage, collectively known as delegate-proxy patterns because they rely on the 'DELEGATECALL' instruction. Building on this concept, EVMPatch [97] uses a proxy pattern to facilitate quick and cost-effective smart contract upgrades, allowing for patching vulnerabilities or adding new features to deployed contracts without requiring data migration or disrupting functionality.

While upgradable smart contracts offer significant advantages, they also introduce notable security challenges. Their secure implementation requires a deep understanding of security principles, and they can be vulnerable to exploitation for malicious purposes. To address these issues, Bodell et al. [16] developed a comprehensive taxonomy that characterizes the unique behaviors of upgradable smart contracts. Their research also identified 2,546 real-world security and safety issues related to upgradable smart contracts, classified into six major categories.

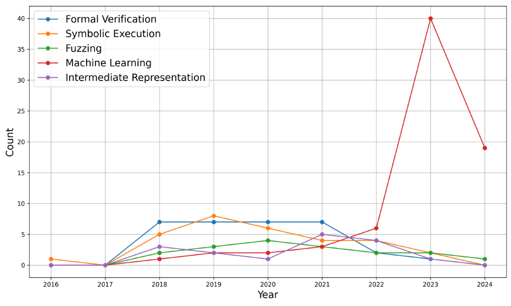

Fig. 7. Trends in method usage from 2016 to July 2024.

5.5 Evolving Trends in Contract Analyzer Usage

Smart contract analyzers, as the primary strategy for identifying vulnerabilities in smart contracts, have been extensively studied by researchers. As discussed in Section 5.1, there are five common methodologies for smart contract analysis. The utilization of these techniques has varied significantly over time.

Figure 7 illustrates a significant shift in method usage trends after 2021. Before 2021, methods such as Formal Verification, Symbolic Execution, and Fuzzing showed steady or increasing usage, peaking between 2018 and 2020. Post-2021, however, these methods experienced a notable decline, indicating a decrease in their adoption or relevance. Formal Verification and Symbolic Execution, once widely adopted, showed a marked decrease, suggesting a shift in research focus or industry practices. Fuzzing exhibited a similar trend, peaking around 2020 before declining. Intermediate Representation (IR), despite its relatively stable usage, also declined post-2021, reinforcing the general downward trend across most methods.

On the other hand, Machine Learning stands out as an exception to this trend. While Machine Learning had fluctuating usage in earlier years, its relevance surged dramatically in 2023, reflecting its growing importance and widespread application in contemporary fields. The boost of Machine Learning also reveals that an increasing number of researchers are paying attention to smart contract security, leveraging advanced techniques to address emerging challenges.

This section has provided a comprehensive examination and categorization of defense methodologies designed to prevent or mitigate smart contract attacks. These findings offer a comprehensive response to RQ3, as defined in Section 2.2. However, there is a notable lack of empirical evidence to evaluate the effectiveness of these methodologies and their associated tools. Despite claims of superiority, empirical studies are necessary to determine which methods are practical and beneficial in real-world applications.

6 Evaluation

This study provides a comprehensive overview of SOTA automated analysis tools for smart contracts. Since the field of smart contract analysis is relatively new and rapidly evolving, keeping up with the latest developments and understanding the strengths and limitations of existing tools can be challenging. Therefore, we conducted an extensive review of the literature and relevant websites to compile a list of the most promising analysis tools for smart contracts.

6.1 Experimental Settings

6.1.1 Tool Selection. Vulnerability detection tools are widely employed to assist developers in identifying vulnerabilities within smart contracts. Numerous analysis tools have been developed for this purpose. We have compiled a list of 169 such tools based on academic literature and online sources. This list includes key properties such as publication venue, methodology, input type, open-source availability, and the specific vulnerabilities they target. The full list is available in a comprehensive table.[3]

For our comparative analysis, we selected tools based on four criteria:

— Available: The tool must be publicly available and accessible for download or running with a **command line interface (CLI)**.
— Functionality: The tool must be designed for smart contracts and capable of detecting vulnerabilities. This excludes tools that only construct artifacts, such as control flow graphs.
— Compatibility: The tool must operate on the source code of the smart contract, excluding those that only consider EVM bytecode.
— Documentation: The tool must provide comprehensive documentation and user guides.

Application of these criteria to our comprehensive tool list resulted in the identification of 14 tools meeting all requirements: ConFuzzius [116], Conkas [40], Maian [82], Manticore [77], Mythril [34], Osiris [118], Oyente [72], Securify [17], sfuzz [81], Slither [37], Smartcheck [113], solhint [90], GPT-4o [1], and Meta-Llama-3.1-8b [120].

6.1.2 Benchmarking Dataset. A key issue when evaluating analysis tools is obtaining a sufficient number of vulnerable smart contracts. Despite the availability of numerous open-source analysis tools, comparing and reproducing results can be challenging due to the lack of publicly available datasets. Most analysis tools primarily check for only some of the well-known Ethereum smart contract vulnerabilities. To evaluate the effectiveness of any analysis tool, establishing a standard benchmark is crucial. While several researchers have published their datasets [34, 42, 116], these often have limitations such as small sample sizes or an uneven distribution of vulnerable contracts.

To address these limitations, we have compiled an extensive, annotated dataset comprising 110 distinct smart contract test cases. These cases are categorized into 11 sub-datasets: 10 sub-datasets contain known vulnerabilities corresponding to the top 10 vulnerability categories mentioned in Section 3 and one sub-dataset represents safe contracts.

6.1.3 Evaluation Criteria. To systematically and objectively assess the quality of selected smart contract analysis tools, we developed a set of evaluation criteria based on the internationally recognized ISO/IEC 25010 standard. This standard offers a comprehensive framework for software product quality assessment, focusing on eight key characteristics. For our study, we tailored these characteristics to the specific requirements of smart contract analysis, concentrating on four of the most relevant aspects: detection suitability (functional suitability), resource efficiency (performance efficiency), version compatibility (compatibility), and category coverage (usability).

— **Detection Suitability**: This criterion evaluates how well the analysis tools meet user needs, particularly in detecting vulnerabilities in smart contracts. It considers aspects such as accuracy, precision, recall, and adaptability to assess the tools' ability to identify potential security issues within contract code.
— **Resource Efficiency**: This aspect measures how effectively a tool utilizes resources, specifically by examining the execution time required for analyzing smart contracts. Efficient performance is crucial for practical use, especially when dealing with large volumes of contracts.
— **Version Compatibility**: This criterion assesses the tool's ability to function consistently across different versions of the Solidity programming language. Given the frequent updates to Solidity, high compatibility ensures the tool's broader applicability and longevity.

[3]https://github.com/WeiZ-boot/survey-on-smart-contract-vulnerability

Table 3. Comparative Analysis of Vulnerability Detection Accuracy Across Smart Contract Detection Tools

Category	**ConFuzzius**	**Conkas**	**Maian**	**Manticore**	**Mythril**	**Osiris**	**Oyente**	**Securify**	**sFuzz**	**Slither**	**Smartcheck**	**solhint**	**GPT-4o**	**Llama-3.1-8b**
Reentrancy	9	10	0	8	9	7	7	8	6	9	8	0	10	9
Arithmetic	7	9	0	7	7	9	9	0	6	0	1	0	10	10
Gasless Send	0	0	0	0	0	0	0	0	3	0	7	0	5	2
Unsafe Suicide	4	0	4	0	3	0	0	0	0	6	0	4	9	9
Unsafe Delegatecall	1	0	0	4	6	0	0	0	5	7	0	0	9	10
Unchecked Send	9	10	0	2	9	0	0	0	0	0	8	6	8	6
TOD	2	7	0	0	0	0	2	2	0	0	9	0	3	3
Timestamp Manipulation	8	8	0	7	6	2	0	0	1	8	0	10	10	7
Authorization through tx.origin	0	0	0	3	6	0	0	0	0	0	0	9	10	6
Bad Randomness	2	0	0	0	8	0	0	0	0	0	0	5	10	2
Total	42	44	40	31	54	18	18	10	21	30	33	34	**84**	64

— **Category Coverage**: This aspect evaluates the tool's ability to detect a wide range of vulnerability types. A tool with extensive category coverage is more valuable to users, as it can identify various vulnerabilities within a contract, enhancing overall usability.

To further elaborate, detection suitability can be seen as a binary classification problem. This binary classification method simplifies the evaluation methodology and provides an effective measure of the tool's precision in vulnerability identification. The classification outcomes are categorized into four distinct groups:

— **True Positive (TP)**: Tool correctly identifies a vulnerability in a contract when it actually exists.
— **False Positive (FP)**: Tool incorrectly identifies a vulnerability in a contract when none exists.
— **False Negative (FN)**: Tool fails to identify a vulnerability when one actually exists.
— **True Negative (TN)**: Tool correctly identifies that a contract does not have a vulnerability when it does not.

To evaluate a tool's detection effectiveness, we employ three key metrics: Precision, which is the ratio of true positive results to all positive results predicted by the tool (Precision = TP / (TP + FP)); Recall rates, which is the ratio of true positive results to all actual positive cases (Recall = TP / (TP + FN)); and F1-score, which is the harmonic mean of Precision and Recall (F1 = (2 * Precision * Recall) / (Precision + Recall)).

6.2 Experimental Results

6.2.1 Detection Suitability. We first measure the detection effectiveness of the selected tools in identifying vulnerabilities. We test the selected tools on our benchmark, and the results are summarized in Table 3. This table presents an overview of the strengths and weaknesses of the selected tools across the common 10 categories discussed in Section 3. The numbers in each cell show the number of TPs identified by each tool for each vulnerability category.

Table 4. Accuracy and F1-Score Across Smart Contract Detection Tools

Metric	ConFuzzius	Conkas	Maian	Manticore	Mythril	Osiris	Oyente	Securify	sFuzz	Slither	Smartcheck	solhint	GPT-4o	Llama-3.1-8b
TP	42	44	4	31	54	18	18	10	21	30	33	34	84	64
FN	58	56	96	69	46	82	82	90	79	70	67	66	16	36
FP	3	4	0	0	0	2	2	0	1	0	0	0	7	10
TN	7	6	10	10	10	8	8	10	9	10	10	10	3	0
Accuracy	45%	46%	13%	37%	58%	24%	24%	18%	27%	36%	39%	40%	**79%**	58%
Recall	0.42	0.44	0.04	0.31	0.54	0.18	0.18	0.1	0.21	0.3	0.33	0.34	0.84	0.64
Precision	0.93	0.92	1	1	1	0.9	0.9	1	0.95	1	1	1	0.92	0.86
F1-score	0.58	0.59	0.08	0.47	0.70	0.3	0.3	0.18	0.34	0.46	0.49	0.51	**0.88**	0.74

Table 5. Execution Time, Compatibility Version (S_v), and Category Coverage (S_c) Scores for Each Tool

Metric	ConFuzzius	Conkas	Maian	Manticore	Mythril	Osiris	Oyente	Securify	sFuzz	Slither	Smartcheck	solhint	GPT-4o	Llama-3.1-8b
Comp. ver.	0.8.x	0.5.x	0.8.x	0.8.x	0.8.x	0.4.21	0.4.19	0.5.11	0.4.24	0.6.x	0.8.x	0.8.x	0.8.x	0.8.x
S_v	5	2	5	5	5	1	1	2	1	3	5	5	5	5
S_c	8	5	1	6	7	3	3	2	5	4	5	5	10	10
Avg. Time	18m9s	48m	57s	11m2s	4m17s	1m32s	4s	27s	18m	**1s**	3s	**1s**	2.2s	47s

GPT-4o shows the highest overall performance, detecting 84 out of 100 vulnerabilities across all categories. Llama-3.1-8b is the second-best performer, detecting 64 vulnerabilities. Among traditional tools, Mythril performs best, with 54 TPs. The results also reveal that different tools perform better at identifying certain vulnerability categories. For example, GPT-4o, Conkas, and Llama-3.1-8b perform best at reentrancy vulnerabilities, whereas Smartcheck performs best at Gasless Send vulnerabilities. LLM tools generally outperform traditional tools across most categories.

To further study the tools' effectiveness, we add safe contracts to our experiments. The results are presented in Table 4. GPT-4o shows the best overall performance with the highest Accuracy (79%) and F1-score (0.88). Llama-3.1-8b and Mythril tie for second place in Accuracy (58%), but Llama-3.1-8b has a higher F1-score (0.74 vs. 0.70). Interestingly, Llama-3.1-8b has the highest FP count (10), followed by GPT-4o (7). While LLMs (especially GPT-4o) show promising results in overall vulnerability detection, they may exhibit high FP rates. Traditional tools, despite lower overall performance, might be valuable for their high precision in specific scenarios.

6.2.2 Resource Efficiency. Execution time is a crucial factor in evaluating a tool's effectiveness, as it directly impacts efficiency. We assessed the resource efficiency of each tool by measuring its execution time, as indicated in Table 5.

Our analysis revealed significant variations in execution time across different tools. Slither and solhint emerged as the fastest tools, with an average execution time of 1 second. GPT-4o also demonstrated high efficiency, with an average execution time of 2.2 seconds. On the other end of the spectrum, Conkas was the slowest, taking 48 minutes on average to complete its analysis.

Table 6. Overall Scores for Each Tool with Different Weight Configurations

Weights: α, β, γ, δ	**ConFuzz**	**Conkas**	**Maian**	**Mantic**	**Mythril**	**Osiris**	**Oyente**	**Securify**	**sFuzz**	**Slither**	**SmartCh**	**solhint**	**GPT-4o**	**Llama-3.1-8b**
0.25, 0.25, 0.25, 0.25	4.37	2.89	1.86	3.69	4.46	1.62	2.22	1.55	2.18	5.16	4.31	6.00	**6.86**	5.26
0.7, 0.1, 0.1, 0.1	4.42	3.88	1.51	3.71	5.28	2.07	2.30	1.71	2.51	4.25	4.07	4.80	**7.49**	5.59
0.1, 0.7, 0.1, 0.1	1.75	1.16	0.85	1.48	1.81	0.71	2.39	0.84	0.88	8.06	3.72	**8.40**	5.47	2.23
0.1, 0.1, 0.7, 0.1	4.75	2.35	3.74	4.47	4.79	1.25	1.49	1.82	1.47	3.86	4.72	5.40	**5.75**	5.10
0.1, 0.1, 0.1, 0.7	6.55	4.15	1.34	5.07	5.99	2.45	2.69	1.82	3.87	4.46	4.72	5.40	**8.75**	8.10

ConFuzzius and sFuzz also exhibited long execution time, which could be attributed to their utilization of fuzzing methods.

6.2.3 Version Compatibility and Category Coverage. Given Solidity's continuous updates, it is essential for analysis tools to keep pace with the latest versions to ensure accurate vulnerability detection. We assessed each tool's compatibility with Solidity versions up to 0.8.19 (the latest version as of January 2023). We assigned a rating value (S_v) to different Solidity versions based on their compatibility with each tool. The rating value starts at 1 for version 0.4.x and increases to 5 for version 0.8.x.

The range of vulnerability categories a tool can detect is crucial in evaluating its effectiveness. Tools with wider category coverage may be more versatile and effective in comprehensive smart contract audits. We assigned a score (S_c) to each tool based on the number of vulnerability categories it can detect. For instance, a tool capable of detecting 5 categories receives an S_c score of 5.

Table 5 summarizes the S_v and S_c scores for each evaluated tool. Most tools, including ConFuzzius, Maian, Manticore, Mythril, Smartcheck, solhint, GPT-4o, and Llama-3.1-8b, support Solidity version 0.8.x, achieving the highest S_v score of 5. GPT-4o and Llama-3.1-8b demonstrated the highest S_c score of 10, indicating that they cover all identified vulnerability categories.

6.2.4 Overall Effectiveness. To provide a comprehensive assessment of the smart contract analysis tools, we developed a weighted sum method that balances the key factors evaluated in the previous sections: detection effectiveness, resource efficiency, version compatibility, and category coverage. We assign weights to each factor based on its relative importance: α to accuracy, β to average execution time, γ to compatibility version, and δ to category coverage, where $\alpha+\beta+\gamma+\delta = 1$.

The overall score for each tool is calculated using the following formula:

$$Score = \alpha \times A * 10 + \beta \times (1/AEX) * 10 + \gamma \times S_v + \delta \times S_c,$$

where A is the accuracy value and AEX is the average execution time.

Table 6 presents the scores for each tool under five different weight configurations, emphasizing different aspects of tool performance. GPT-4o consistently achieved the highest scores across most weight configurations, indicating its well-rounded performance. Llama-3.1-8b generally performed well, often scoring second or third highest. Among traditional tools, solhint, Slither, and Mythril tended to score higher than others. When emphasis was placed on resource efficiency, solhint led with a score of 8.40, followed closely by Slither (8.06).

The choice of tool may depend on specific project priorities. For rapid, preliminary scans, tools such as solhint or Slither might be preferred. For in-depth security audits, GPT-4o or a combination of tools could be most effective. The flexibility provided by different weight configurations in our scoring system allows for tool selection based on project-specific requirements.

In addition, a combined approach using multiple tools could potentially yield the most comprehensive results. For example, solhint could be used for grammatical checks and ensuring code adherence to standards. LLMs such as GPT-4o or Llama-3.1-8b could identify known vulnerabilities and provide suggestions to prevent attacks. Slither could offer deeper analysis of the code and detect semantic-level issues. These findings provide clear and comprehensive responses to **RQ4** and **RQ5**, highlighting the relative strengths and weaknesses of different tool types and suggesting strategies for their effective use in smart contract vulnerability detection.

6.3 Threat to Validity

In our research, we identified two specific threats that could impact the results and findings of our experiment: smart contract vulnerability types and the generality of the evaluation datasets.

One potential threat is the subjectivity and variation among researchers in evaluating and categorizing vulnerabilities. Different perspectives, criteria, and interpretations can introduce bias, affecting the validity of our comparisons. To mitigate this, we adopted a systematic approach based on industry standards and best practices, as detailed in Section 3. We thoroughly reviewed and discussed each vulnerability category to ensure consistent and objective classification, consulting experts and existing literature. By providing clear definitions and criteria, we aim to minimize ambiguity and ensure consistency across researchers and readers.

Another threat to validity is the generality of the evaluation datasets, which refers to how well these datasets reflect real-world scenarios and usage patterns of smart contracts. If the datasets are too narrow, our findings may lack generalizability. To address this, we collected a diverse set of 110 contract test cases from various sources, including publicly available datasets and our own developed test cases, covering a wide range of applications and code sizes. This diversity aims to provide a more representative evaluation of smart contract vulnerabilities and improve the generalizability of our findings.

7 Conclusions

As the adoption of smart contract technology continues to surge, its security becomes increasingly critical for the robustness of blockchain ecosystems. Our comprehensive survey, encompassing vulnerabilities, attacks, defenses, and tool support, not only advances academic understanding but also offers practical implications for developers and stakeholders in blockchain technology.

Our novel classifications of vulnerability types and attack patterns provide developers with a clearer understanding of potential security risks, which is crucial for building more secure smart contracts. By being aware of common vulnerabilities and how attacks exploit them, developers can proactively incorporate security measures during development. Additionally, our evaluation of 14 vulnerability-detection tools helps developers choose the most effective tools for their needs, enhancing smart contract security.

However, the smart contract landscape is rapidly evolving, with new functionalities and protocols introducing new security vulnerabilities. Ongoing investment in research and development is crucial to make smart contract languages more robust. There is growing interest in using programming languages other than Solidity, such as Go and Rust, which offer stronger syntax and logical soundness, potentially addressing some of Solidity's security issues. Additionally, there is a need for more powerful analysis tools capable of detecting dynamic or logic errors, as existing tools primarily focus on known vulnerabilities and attacks. Addressing unknown attacks remains a significant challenge for future research, and developing automated approaches for repairing vulnerable smart contracts after deployment could be a fruitful direction.

References

[1] Josh Achiam, Steven Adler, and Sandhini Agarwal. 2023. GPT-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. 2014. SACO: Static analyzer for concurrent objects. In *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference (TACAS 2014), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2014), Grenoble, France, April 5–13, 2014. Proceedings 20*. Springer, 562–567.

[3] Elvira Albert, Puri Arenas, and Samir Genaim. 2008. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Static Analysis: 15th International Symposium (SAS 2008), Valencia, Spain, July 16–18, 2008. Proceedings 15*. Springer, 221–237.

[4] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2020. GASOL: Gas analysis and optimization for Ethereum smart contracts. In *Tools and Algorithms for the Construction and Analysis of Systems — 26th International Conference (TACAS 2020), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2020), Dublin, Ireland, April 25–30, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12079)*. Springer, 118–125.

[5] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2019. Running on fumes — preventing out-of-gas vulnerabilities in Ethereum smart contracts using static resource analysis. In *Verification and Evaluation of Computer and Communication Systems — 13th International Conference (VECoS 2019), Porto, Portugal, October 9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11847)*. Springer, 63–78.

[6] Elvira Albert, Pablo Gordillo, and Benjamin Livshits. 2018. Ethir: A framework for high-level analysis of Ethereum bytecode. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 513–520.

[7] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018), Los Angeles, CA, January 8–9, 2018*. ACM, 66–77.

[8] Shaima AL Amri, Leonardo Aniello, and Vladimiro Sassone. 2023. A review of upgradeable smart contract patterns based on OpenZeppelin technique. *The Journal of The British Blockchain Association* (2023).

[9] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, and Yacov Manevich. 2018. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the 13th EuroSys Conference (EuroSys 2018), Porto, Portugal, April 23–26, 2018*. ACM, 30:1–30:15.

[10] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on Ethereum smart contracts (SoK). In *Principles of Security and Trust: 6th International Conference (POST 2017), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2017), Uppsala, Sweden, April 22–29, 2017, Proceedings 6*. Springer, 164–186.

[11] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. 2018. A formal model of Bitcoin transactions. In *Financial Cryptography and Data Security: 22nd International Conference (FC'18), Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*. Springer, 541–560.

[12] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. 2018. Formal modeling and verification of smart contracts. In *Proceedings of the 7th International Conference on Software and Computer Applications (ICSCA'18), Kuantan, Malaysia, February 08–10, 2018*. ACM, 322–326.

[13] Massimo Bartoletti and Roberto Zunino. 2019. Formal models of bitcoin contracts: A survey. *Frontiers Blockchain* 2 (2019), 8.

[14] Jan Arvid Berg, Robin Fritsch, and Lioba Heimbach. 2022. An empirical study of market inefficiencies in Uniswap and SushiSwap. In *International Conference on Financial Cryptography and Data Security*. Springer, 238–249.

[15] Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. 2020. Making Tezos smart contracts more reliable with Coq. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 60–72.

[16] William E. Bodell III, Sajad Meisami, and Yue Duan. 2023. Proxy hunting: Understanding and characterizing proxy-based upgradeable smart contracts in blockchains. In *32nd USENIX Security Symposium (USENIX Security'23)*. 1829–1846.

[17] Lexi Brent, Neville Grech, and Sifis Lagouvardos. 2020. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.

[18] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981* (2018).

[19] Vitalik Buterin. 2013. Ethereum white paper. *GitHub Repository* 1 (2013), 22–23.

[20] Fran Casino, Thomas K. Dasaklis, and Constantinos Patsakis. 2019. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics* 36 (2019), 55–81.

[21] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.

[22] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. 2024. Smart contract and DeFi security tools: Do they meet the needs of practitioners? In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.

[23] Chong Chen, Jianzhong Su, and Jiachi Chen. 2023. When ChatGPT meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520* (2023).

[24] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)* 53, 3 (2020), 1–43.

[25] Ting Chen, Rong Cao, and Ting Li. 2020. SODA: A generic online detection framework for smart contracts. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society.

[26] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2021. GasChecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Trans. Emerg. Top. Comput.* 9, 3 (2021), 1433–1448.

[27] Ting Chen, Xiaoqi Li, and Ying Wang. 2017. An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks. In *Information Security Practice and Experience: 13th International Conference ISPEC'17, Melbourne, VIC, Australia, December 13–15, 2017, Proceedings 13*. Springer, 3–24.

[28] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 654–668.

[29] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *IEEE European Symposium on Security and Privacy (EuroS&P;19), Stockholm, Sweden, June 17–19, 2019*. IEEE, 185–200.

[30] Hanting Chu, Pengcheng Zhang, Hai Dong, Yan Xiao, Shunhui Ji, and Wenrui Li. 2023. A survey on smart contract vulnerabilities: Data sources, detection and repair. *Information and Software Technology* 159 (2023), 107221.

[31] Tim Copeland. 2020. *How a Genius Hacker Made $350,000 Exploiting DeFi*. Retrieved from https://decrypt.co/19612/how-a-genius-hacker-made-350000-exploiting-defi

[32] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023).

[33] Monika Di Angelo and Gernot Salzer. 2019. A survey of tools for analyzing Ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON'19)*. IEEE, 69–78.

[34] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *ICSE'20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June–19 July, 2020*. ACM, 530–541.

[35] Joshua Ellul and Gordon J. Pace. 2018. Runtime verification of Ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC'18)*. IEEE, 158–163.

[36] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. 2019. SoK: Transparent dishonesty: Front-running attacks on blockchain. In *Financial Cryptography and Data Security —FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11599)*. Springer, 170–189.

[37] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE/ACM, 8–15.

[38] Christof Ferreira Torres, Mathis Baden, and Robert Norvill. 2019. ÆGIS: Smart shielding of smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2589–2591.

[39] flashbots. 2023. *MVE-Explore v1*. Retrieved from https://explore.flashbots.net/

[40] Terrell Ford. 2022. *Benchmarking Ethereum Smart Contract Static Analysis Tools*. Ph.D. Dissertation.

[41] J. M. G. Martínez, P. Carracedo, and D. G. Comas. 2022. An analysis of the blockchain and COVID-19 research landscape using a bibliometric study. *Sustainable Technology and Entrepreneurship* 1, 1 (2022), 100006.

[42] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20), Virtual Event, USA, July 18–22, 2020*. ACM, 415–427.

[43] Neville Grech, Michael Kong, and Anton Jurisevic. 2018. Madmax: Surviving out-of-gas conditions in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

[44] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 557–560.

[45] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *Proc. ACM Program. Lang.* 2, POPL (2018), 48:1–48:28.

[46] Sunil Gupta, Hitesh Kumar Sharma, and Monit Kapoor. 2023. *Blockchain for Secure Healthcare Using Internet of Medical Things (IoMT)*. Springer.

[47] Hazim Hanif, Mohd Hairul Nizam Md Nasir, and Mohd Faizal Ab Razak. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications* 179 (2021), 103009.

[48] Dominik Harz and William Knottenbelt. 2018. Towards safer smart contracts: A survey of languages and verification methods. *arXiv preprint arXiv:1809.09805* (2018).

[49] Daojing He, Rui Wu, Xinji Li, Sammy Chan, and Mohsen Guizani. 2023. Detection of vulnerabilities of blockchain smart contracts. *IEEE Internet of Things Journal* 10, 14 (2023), 12178–12185.

[50] Jingxuan He, Mislav Balunović, and Nodar Ambroladze. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 531–548.

[51] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. {EOSAFE}: Security analysis of {EOSIO} smart contracts. In *30th USENIX Security Symposium (USENIX Security'21)*. 1271–1288.

[52] Bin Hu, Zongyang Zhang, and Jianwei Liu. 2021. A comprehensive survey on smart contract construction and execution: Paradigms, tools, and systems. *Patterns* 2, 2 (2021), 100179.

[53] Teng Hu, Xiaolei Liu, Ting Chen, Xiaosong Zhang, Xiaoming Huang, Weina Niu, Jiazhong Lu, Kun Zhou, and Yuan Liu. 2021. Transaction-based classification and detection approach for Ethereum smart contract. *Information Processing & Management* 58, 2 (2021), 102462.

[54] Laurie Hughes, Yogesh K. Dwivedi, Santosh K. Misra, Nripendra P. Rana, Vishnupriya Raghavan, and Viswanadh Akella. 2019. Blockchain research, practice and policy: Applications, benefits, limitations, emerging research themes and research agenda. *International Journal of Information Management* 49 (2019), 114–129.

[55] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2023. Security threat mitigation for smart contracts: A comprehensive survey. *Comput. Surveys* (2023).

[56] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3–7, 2018*. ACM, 259–269.

[57] Jiao Jiao, Shuanglong Kan, and Shang-Wei Lin. 2020. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy (SP'20)*. IEEE, 1695–1712.

[58] josselinfeist. 2018. *Contract Upgrade Anti-patterns*. Retrieved from https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/

[59] josselinfeist. 2018. *How Contract Migration Works*. Retrieved from https://blog.trailofbits.com/2018/10/29/how-contract-migration-works/

[60] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S. Matthew Weinberg, and Edward W. Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018*. USENIX Association, 1353–1370.

[61] Basim Khajwal, C.-H. Luke Ong, and Dominik Wagner. 2023. Fast and correct gradient-based optimisation for probabilistic programming via smoothing. In *European Symposium on Programming*. Springer Nature Switzerland Cham, 479–506.

[62] Shafaq Naheed Khan, Faiza Loukil, Chirine Ghedira Guegan, Elhadj Benkhelifa, and Anoud Bani-Hani. 2021. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Netw. Appl.* 14, 5 (2021), 2901–2925.

[63] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22–26, 2016*. IEEE Computer Society, 839–858.

[64] Stefano Lande and Roberto Zunino. 2018. SoK: Unraveling Bitcoin smart contracts. *Principles of Security and Trust LNCS* 10804 (2018), 217.

[65] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 438–453.

[66] Chunmiao Li, Shijie Nie, Yang Cao, Yijun Yu, and Zhenjiang Hu. 2020. Trace-based dynamic gas estimation of loops in smart contracts. *IEEE Open J. Comput. Soc.* 1 (2020), 295–306.

[67] Peiru Li, Shanshan Li, Mengjie Ding, Jiapeng Yu, He Zhang, Xin Zhou, and Jingyue Li. 2022. A vulnerability detection framework for hyperledger fabric smart contracts based on dynamic and static analysis. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. 366–374.

[68] Yuwei Li, Shouling Ji, Yuan Chen, and Liang. 2021. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security'21)*. 2777–2794.

[69] Jian-Wei Liao, Tsung-Ta Tsai, and Chia-Kang He. 2019. SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS'19)*. IEEE, 458–465.

[70] Jingwei Liu, Xiaolu Li, Lin Ye, Hongli Zhang, Xiaojiang Du, and Mohsen Guizani. 2018. BPDS: A blockchain based privacy-preserving data sharing for electronic medical records. In *IEEE Global Communications Conference, GLOBECOM 2018, Abu Dhabi, United Arab Emirates, December 9–13, 2018*. IEEE, 1–6.

[71] Ning Lu, Bin Wang, Yongxin Zhang, Wenbo Shi, and Christian Esposito. 2021. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Softw. Pract. Exp.* 51, 10 (2021), 2065–2084.

[72] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016*. ACM, 254–269.

[73] Penghui Lv, Yu Wang, Yazhe Wang, and Qihui Zhou. 2021. Potential risk detection system of hyperledger fabric smart contract based on static analysis. In *IEEE Symposium on Computers and Communications, ISCC 2021, Athens, Greece, September 5–8, 2021*. IEEE, 1–7.

[74] Patrick McCorry, Siamak F. Shahandashti, and Feng Hao. 2017. A smart contract for boardroom voting with maximum voter privacy. In *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3–7, 2017, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10322)*. Springer, 357–375.

[75] METAMASK. 2023. *A Crypto Wallet & Gateway to Blockchain Apps*. Retrieved from https://metamask.io/

[76] Microsoft. 2019. *Onefuzz*. Retrieved from https://github.com/microsoft/onefuzz

[77] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. IEEE, 1186–1189.

[78] Behkish Nassirzadeh, Huaiying Sun, and Sebastian Banescu. 2022. Gas gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. In *The International Conference on Mathematical Research for Blockchain Economy*. Springer, 143–167.

[79] Zeinab Nehai, Pierre-Yves Piriou, and Frederic Daumas. 2018. Model-checking of smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 980–987.

[80] T. Q. Nguyen, A. K. Das, and L. T. Tran. 2019. NEO smart contract for drought-based insurance. In *2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE'19)*. IEEE, 1–4.

[81] Tai D. Nguyen, Long H. Pham, and Jun Sun. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[82] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663.

[83] Russell O'Connor. 2017. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. 107–120.

[84] King of the Ether Throne. 2016. *KotET – Post-Mortem Investigation*. Retrieved from https://www.kingoftheether.com/postmortem.html

[85] Santiago Palladino. 2017. *The Parity Wallet Hack Explained*. Retrieved from https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[86] Qianqian Pan, Jun Wu, Ali Kashif Bashir, Jianhua Li, Sahil Vashisht, and Raheel Nawaz. 2022. Blockchain and AI enabled configurable reflection resource allocation for IRS-aided coexisting drone-terrestrial networks. *IEEE Wireless Communications* 29, 6 (2022), 46–54.

[87] Alfredo J. Perez and Sherali Zeadally. 2022. Secure and privacy-preserving crowdsensing using smart contracts: Issues and solutions. *Comput. Sci. Rev.* 43 (2022), 100450.

[88] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *USENIX Security Symposium*. 1325–1341.

[89] Joshua R. Polanin, Terri D. Pigott, Dorothy L. Espelage, and Jennifer K. Grotpeter. 2019. Best practice guidelines for abstract screening large-evidence systematic reviews and meta-analyses. *Research Synthesis Methods* 10, 3 (2019), 330–342.

[90] Purathani Praitheeshan, Lei Pan, Xi Zheng, Alireza Jolfaei, and Robin Doss. 2021. SolGuard: Preventing external call issues in smart contract-based multi-agent robotic systems. *Information Sciences* 579 (2021), 150–166.

[91] Sarah Qahtan, Khaironi Yatim, Hazura Zulzalil, Mohd Hafeez Osman, A. A. Zaidan, and Hassan A. Alsattar. 2023. Review of healthcare industry 4.0 application-based blockchain in terms of security and privacy development attributes: Comprehensive taxonomy, open issues and challenges and recommended solution. *Journal of Network and Computer Applications* 209 (2023), 103529.

[92] Peng Qian, Zhenguang Liu, and Qinming He. 2020. Towards automated reentrancy detection for smart contracts based on sequential models. *IEEE Access* 8 (2020), 19685–19695.

[93] Meixun Qu, Xin Huang, Xu Chen, Yi Wang, Xiaofeng Ma, and Dawei Liu. 2018. Formal verification of smart contracts from the perspective of concurrency. In *Smart Blockchain - First International Conference, SmartBlock 2018, Tokyo, Japan, December 10–12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11373)*. Springer, 32–43.

[94] Anu Raj and Shiva Prakash. 2022. A privacy-preserving authentic healthcare monitoring system using blockchain. *International Journal of Software Science and Computational Intelligence (IJSSCI)* 14, 1 (2022), 1–23.

[95] Heidelinde Rameder, Monika Di Angelo, and Gernot Salzer. 2022. Review of automated vulnerability analysis of smart contracts on Ethereum. *Front. Blockchain* 5 (2022).

[96] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, CA, February 24–27, 2019*. The Internet Society.

[97] Michael Rodler, Wenting Li, and Ghassan O. Karame. 2021. {EVMPatch}: Timely and automated patching of Ethereum smart contracts. In *30th USENIX Security Symposium (USENIX Security'21)*. 1289–1306.

[98] Muhammad Saad, Jeffrey Spaulding, and Laurent Njilla. 2020. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys & Tutorials* 22, 3 (2020), 1977–2008.

[99] Md Nazmus Saadat, Syed Abdul Halim Syed Abdul Rahman, and Rasheed Mohammad Nassr. 2019. Blockchain based crowdfunding systems in Malaysian Perspective. In *Proceedings of the 2019 11th International Conference on Computer and Automation Engineering*. 57–61.

[100] Christoph Sendner, Huili Chen, Hossein Fereidooni, Lukas Petzi, Jan König, Jasper Stang, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2023. Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society.

[101] Kostya Serebryany. 2017. {OSS-Fuzz}-Google's continuous fuzzing service for open source software. (2017).

[102] Supriya Shakya, Arnab Mukherjee, and Raju Halder. 2022. SmartMixModel: Machine learning-based vulnerability detection of solidity smart contracts. In *2022 IEEE International Conference on Blockchain (Blockchain'22)*. IEEE, 37–44.

[103] Chaofan Shou, Jing Liu, Doudou Lu, and Koushik Sen. 2024. LLM4Fuzz: Guided fuzzing of smart contracts with large language models. *arXiv preprint arXiv:2401.11108* (2024).

[104] Rajesh Kumar Singh, Ruchi Mishra, Shivam Gupta, and Archana A. Mukherjee. 2023. Blockchain applications for secured and resilient supply chains: A systematic literature review and future research agenda. *Comput. Ind. Eng.* 175 (2023), 108854.

[105] Camila Sitonio and Alberto Nucciarelli. 2018. The impact of blockchain on the music industry. (2018).

[106] Samuel Steffen, Benjamin Bichsel, Mario Gersbach, Noa Melchior, Petar Tsankov, and Martin Vechev. 2019. zkay: Specifying and enforcing data privacy in smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1759–1776.

[107] Samuel Steffen, Benjamin Bichsel, and Martin Vechev. 2022. Zapper: Smart contracts with data and identity privacy. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2735–2749.

[108] Jakia Sultana, Say Yen Teoh, and Stan Karanasios. 2022. The impact of blockchain on supply chains: A systematic review. *Australasian Journal of Information Systems* 26 (2022).

[109] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Wei Ma, Lyuye Zhang, Miaolei Shi, and Yang Liu. 2024. LLM4Vuln: A unified evaluation framework for decoupling and enhancing LLMs' vulnerability reasoning. *arXiv preprint arXiv:2401.16185* (2024).

[110] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24)*. 1–13.

[111] Nick Szabo. 1996. Smart contracts: Building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought (16)* 18, 2 (1996), 28.

[112] Wesley Joon-Wie Tann, Xing Jie Han, and Sourav Sen Gupta. 2018. Towards safer smart contracts: A sequence learning approach to detecting security threats. *arXiv preprint arXiv:1811.06632* (2018).

[113] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static analysis of Ethereum smart contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27–June 3, 2018*. ACM, 9–16.

[114] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2022. A survey of smart contract formal specification and verification. *ACM Comput. Surv.* 54, 7 (2022), 148:1–148:38.

[115] Christof Ferreira Torres and Ramiro Camino. 2021. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *30th USENIX Security Symposium (USENIX Security'21)*. 1343–1359.

[116] Christof Ferreira Torres, Antonio Ken Iannillo, and Arthur Gervais. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P'21)*. IEEE, 103–119.

[117] Christof Ferreira Torres, Hugo Jonker, and Radu State. 2022. Elysium: Context-aware bytecode-level patching to automatically heal vulnerable smart contracts. In *25th International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2022, Limassol, Cyprus, October 26–28, 2022*. ACM, 115–128.

[118] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in Ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676.

[119] Christof Ferreira Torres, Mathis Steichen, and Radu State. 2019. The art of the scam: Demystifying honeypots in Ethereum smart contracts. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, August 14–16, 2019*. USENIX Association, 1591–1607.

[120] Hugo Touvron, Thibaut Lavril, and Gautier Izacard. 2023. LLaMA: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[121] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2021. ContractWard: Automated vulnerability detection models for Ethereum smart contracts. *IEEE Trans. Netw. Sci. Eng.* 8, 2 (2021), 1133–1144.

[122] Xinming Wang, Jiahao He, and Zhijian Xie. 2019. ContractGuard: Defend Ethereum smart contracts with embedded intrusion detection. *IEEE Transactions on Services Computing* 13, 2 (2019), 314–328.

[123] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151, 2014 (2014), 1–32.

[124] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A greybox fuzzer for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1398–1409.

[125] Zheng Yang and Hang Lei. 2019. FEther: An extensible definitional interpreter for smart-contract verifications in Coq. *IEEE Access* 7 (2019), 37770–37791.

[126] ZenGo. 2020. *Bancor Smart Contracts Vulnerability: It's Not Over*. https://zengo.com/bancor-smart-contracts-vulnerability-its-not-over/

[127] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 270–282.

[128] Luyao Zhang, Tianyu Wu, Saad Lahrichi, Carlos-Gustavo Salas-Flores, and Jiayi Li. 2022. A data science pipeline for algorithmic trading: A comparative study of applications for finance and cryptoeconomics. In *2022 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 298–303.

[129] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. 2023. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)*. IEEE, 615–627.

[130] Zibin Zheng, Shaoan Xie, and Hong-Ning Dai. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Generation Computer Systems* 105 (2020), 475–491.

[131] Liyi Zhou, Kaihua Qin, and Christof Ferreira Torres. 2021. High-frequency trading on decentralized on-chain exchanges. In *2021 IEEE Symposium on Security and Privacy (S&P'21)*. IEEE, 428–445.

[132] Liyi Zhou, Xihan Xiong, and Jens Ernstberger. 2023. SoK: Decentralized finance (DeFi) attacks. In *2023 IEEE Symposium on Security and Privacy (S&P'23)*. IEEE, 2444–2461.

[133] Shunfan Zhou, Malte Möser, and Zhemin Yang. 2020. An ever-evolving game: Evaluation of real-world attacks and defenses in Ethereum ecosystem. In *29th USENIX Security Symposium (USENIX Security'20)*. 2793–2810.

[134] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 3283–3290.

Received 5 June 2023; revised 29 August 2024; accepted 2 September 2024