Security Threat Mitigation For Smart Contracts: A Survey

NIKOLAY IVANOV, Michigan State University, USA
CHENNING LI, Michigan State University, USA
QIBEN YAN*, Michigan State University, USA
ZHIYUAN SUN, King's College London, UK
ZHICHAO CAO, Michigan State University, USA
XIAPU LUO, The Hong Kong Polytechnic University, Hong Kong

The blockchain technology, initially created for cryptocurrency, has been re-purposed for recording state transitions of smart contracts — decentralized applications that can be invoked through external transactions. Smart contracts gained popularity and accrued hundreds of billions of dollars in market capitalization in recent years. Unfortunately, like all other programs, smart contracts are prone to security vulnerabilities that have incurred multimillion-dollar damages over the past decade. As a result, many automated threat mitigation solutions have been proposed to counter the security issues of smart contracts. These threat mitigation solutions include various tools and methods that are challenging to compare. This survey develops a comprehensive classification taxonomy of smart contract threat mitigation solutions within five orthogonal dimensions: defense modality, core method, targeted contracts, input-output data mapping, and threat model. We classify 133 existing threat mitigation solutions using our taxonomy and confirm that the proposed five dimensions allow us to concisely and accurately describe any smart contract threat mitigation solution. In addition to learning *what* the threat mitigation solutions do, we also show *how* these solutions work by synthesizing their actual designs into a set of uniform workflows corresponding to the eight existing defense core methods. We further create an integrated coverage map for the known smart contract vulnerabilities by the existing threat mitigation solutions. Finally, we perform the evidence-based evolutionary analysis, in which we identify trends and future perspectives of threat mitigation in smart contracts and pinpoint major weaknesses of the existing methodologies. For the convenience of smart contract security developers, auditors, users, and researchers, we deploy a regularly updated comprehensive open-source online registry of threat mitigation solutions.

CCS Concepts: • **Security and privacy → Distributed systems security**.

Additional Key Words and Phrases: smart contracts, blockchain, security

ACM Reference Format:
Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2022. Security Threat Mitigation For Smart Contracts: A Survey. *J. ACM* 37, 4, Article 0 (November 2022), 35 pages. https://doi.org/10.1145/1122445.1122456

*Corresponding Author: Qiben Yan (e-mail: qyan@msu.edu)

Authors' addresses: Nikolay Ivanov, ivanovn1@msu.edu, Michigan State University, 426 Auditorium Road, East Lansing, Michigan, USA, 48824; Chenning Li, lichenni@msu.edu, Michigan State University, 426 Auditorium Road, East Lansing, Michigan, USA, 48824; Qiben Yan, qyan@msu.edu, Michigan State University, 426 Auditorium Road, East Lansing, Michigan, USA, 48824; Zhiyuan Sun, zhiyuan.sun@kcl.ac.uk, King's College London, Strand, London WC2R 2LS, London, UK; Zhichao Cao, caozc@msu.edu, Michigan State University, 426 Auditorium Road, East Lansing, Michigan, USA, 48824; Xiapu Luo, daniel.xiapu.luo@polyu.edu.hk, The Hong Kong Polytechnic University, PQ828 Mong Man Wai Building, Kowloon, Hung Hom, Hong Kong.

0004-5411/2022/11-ART0 $15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Blockchain is a decentralized network that sustains distributed records stored in immutable blocks to form an ever-growing chain. In one decade, blockchain technology has evolved from the ledger of cryptocurrency (e.g., Bitcoin, Monero) to the decentralized computing platform (e.g., Ethereum, EOS) that allows the deployment and execution of smart contracts. *Smart contract* is a decentralized program deployed on a blockchain that enforces the execution of protocols and agreements without involving any third party or establishing a mutual trust [144]. A smart contract provides a set of functions to be called via transactions and executed by the blockchain's virtual machine (VM). Most smart contracts are written in high-level special-purpose programming languages, such as Solidity, JavaScript, or Vyper, and compiled into the blockchain VM bytecode. For example, the Ethereum Virtual Machine (EVM) is the blockchain VM for executing smart contracts on the Ethereum platform[1]. An important feature of smart contracts is their ability to perform financial operations with cryptocurrency and valuable custom tokens (e.g., ERC20, ERC721). As of March 2022, the total market capitalization of smart contracts exceeds 300 billion USD [4].

The large amounts of valued assets stored and transacted by smart contracts made them lucrative targets for attackers. Numerous security vulnerabilities and attacks on Ethereum smart contracts have been hampering their widespread adoption [73, 138]. In the past few years, exploitations of these vulnerabilities caused hundreds of millions of dollars in damages. For example, in June 2016, about $150 million were stolen from the popular DAO contract [69]. In July 2017, about $30 million were stolen from the Parity multi-signature wallet [41]. Not long after that, a bug in the same multi-signature wallet caused the freeze of about $280 million [45].

A large number of approaches and tools have been developed to address different types of smart contract security issues. In this work, we use the term *threat mitigation solutions* to describe the full spectrum of the active defense and passive preventative solutions aiming to reduce or eliminate the threat associated with the exploitation of security vulnerabilities in smart contracts. These solutions include both academic research efforts as well as commercial and open-source software products.

Some surveys have been published that summarize vulnerabilities and attacks in smart contracts [30, 99]. Furthermore, the Smart Contract Weakness Classification and Test Cases database, also known as the SWC Registry [18], identifies and describes 37 classes of known smart contract vulnerabilities (as of March 2022). However, all the existing ways of systematizing smart contract security knowledge focus primarily on vulnerabilities and attacks, paying very little or no attention to the broad swath of defense and prevention mechanisms developed in the past decade. In this work, we bridge the gap in the systematization of the threat mitigation solutions via the following four steps: developing classification taxonomy, synthesizing design workflows of core methods of threat mitigation, creating the map of vulnerability coverage, and conducting an evolutionary analysis.

Step I: Taxonomy. The smart contract threat mitigation constitutes a diverse set of efforts, so finding a uniform organizational methodology for all these solutions poses a major challenge. These solutions employ a variety of techniques, such as symbolic execution [119, 125], formal verification [47], static analysis [44, 175], to name a few. Some of these solutions target specific vulnerabilities, such as reentrancy [132] or integer overflow [141], while others are general-purpose [149]. Some threat mitigation solutions aim at detecting vulnerabilities [111], while others focus on verifying the safe property of a smart contract [129]. In other words, all these solutions vary within multiple dimensions. In this survey, we formalize these dimensions and create a comprehensive

[1]Although it is primarily associated with Ethereum, EVM has also been adopted by some other blockchain platforms, such as Polygon [10] and RSK [11].

taxonomy of smart contract threat mitigation based on five dimensions: defense modality, core method, targeted contracts, data mapping, and threat model.

Step II: Design Workflows. In addition to learning what the smart contract threat mitigation solutions *do*, we also explore *how* they achieve their aimed goals — which is challenging due to a wide variety of innovations and novel techniques employed by the existing solutions. In this work, we study the design workflows of all the 133 smart contract threat mitigation solutions under our investigation, and we subdivide them into eight core methods: *static analysis*, *symbolic execution*, *fuzzing*, *formal analysis*, *machine learning*, *execution tracing*, *code synthesis*, and *transaction interception*. Then, we synthesize the actual designs of the threat mitigation solutions corresponding to each of the eight core methods and build eight *uniform* workflows that summarize the whole variety of threat mitigation solutions for smart contracts.

Step III: Vulnerability Coverage. Next, we raise another important question: which known vulnerabilities are covered (i.e., prevented, detected, or unmasked) by the existing smart contract threat mitigation solutions? Answering this question requires overcoming two significant challenges: i) the lack of explicit and implicit declaration of addressed vulnerabilities by many threat mitigation solutions, and ii) the lack of uniform definitions of smart contract vulnerabilities. To overcome these challenges, we meticulously translate, group, or un-group the vulnerabilities referred to by the authors of the threat mitigation solutions to match the vulnerability classification proposed by the popular SWC Registry. Thus, we develop a unified vulnerability coverage map for these solutions based on the SWC registry.

Step IV: Evolutionary Analysis. We perform an evidence-based evolutionary analysis of existing smart contract threat mitigation solutions to identify trends and potential future research directions. Specifically, we identify the three most promising vectors of development of smart contract threat mitigation solutions: dynamic transaction interception, AI-driven security, and study of human-machine interaction in smart contracts. In addition, we identify two major deficiencies of the existing body of threat mitigation solutions: the under-representation of non-Ethereum smart contracts as targets and the lack of security-related large-scale measurements, especially related to off-chain data.

In summary, in this work, we make the following contributions:

- We develop a five-dimensional threat mitigation taxonomy tailored for smart contracts, and we use this taxonomy to classify 133 existing smart contract threat mitigation solutions.
- We pinpoint eight core methods adopted by the existing smart contract threat mitigation solutions, and we develop synthesized workflows of these methods to demonstrate the internal workings of smart contract threat mitigation.
- We identify the threat mitigation solutions that explicitly declare protection against specific vulnerabilities, and we create a smart contract vulnerability coverage map for these solutions.
- We identify trends and deficiencies of the existing smart contract mitigation solutions based on the findings of this survey and other solid evidence.
- Finally, in the spirit of open research, we develop and publish a constantly updated online registry of threat mitigation solutions, called the STM Registry[2].

Organization. The rest of this work is organized as follows. First, we compare our work with previous surveys related to smart contract security (§2). Then, we describe the methodology employed in this survey (§3). After that, we classify 133 threat mitigation solutions based on the developed five-dimensional taxonomy (§4), followed by a detailed comparative description of designs of the eight core methods of threat mitigation (§5). Next, we compare the threat mitigation methods by their ability to address specific known smart contract vulnerabilities (§6). Then, we

[2]https://stmregistry.io/

discuss trends and future perspectives of threat mitigation in smart contracts (§7), and finally, we conclude our work (§8).

2 PRIOR SURVEYS

A number of previous surveys aimed at smart contract security have been published, which, however, have different perspectives than this survey. Atzei et al. [30] propose the first systematic exposition of the Ethereum security vulnerabilities by organizing the vulnerabilities in three levels: Solidity[3], EVM[4] bytecode, and blockchain. They also illustrate six influential attacks in different application scenarios. In contrast, we primarily target vulnerability mitigation methods rather than the classification of programming pitfalls. Jiachi et al. [51] propose an empirical survey that provides a systematic study of smart contract defects on the Ethereum platform from five aspects: security, availability, performance, maintainability, and re-usability. They collect and analyze smart contract-related posts on *Ethereum.StackExchange*[5] as well as real-world smart contracts to define 20 kinds of contract flaws and 5 relevant impacts. Zou et al. [176] perform an exploratory research to illustrate the current state and potential challenges in smart contract development. Specifically, they conduct semi-structured interviews with 20 developers and professionals, followed by a survey of 232 practitioners to confirm the 5 conclusions from the interviews that focus primarily on smart contract development. In addition, Zhang et al. [169] present a new classification framework for smart contract bugs and construct a dataset of 176 buggy smart contracts. Wang et al. [159] conduct an analysis of the security of Ethereum smart contracts and categorize these security challenges into abnormal contracts, program vulnerabilities, and unsafe external data. Vacca et al. [150] provide a systematic review of techniques and tools used to address the software engineering-specific challenges of blockchain-based applications by analyzing 96 papers. The above surveys summarize smart contract security and development issues, while we focus on vulnerability mitigation solutions.

There are also a number of surveys that take the vulnerability mitigation solutions into consideration. Huashan et al. [50] present a comprehensive and systematic survey on Ethereum systems security which includes vulnerabilities, attacks, and defenses. The authors discuss 44 kinds of vulnerabilities based on the layers of the Ethereum architecture and describe the history, cause, tactic, and direct impact of 26 attacks. As for defenses, the authors enumerate 47 defense mechanisms and provide the best practices to guide contract development. Although they divide the defenses into proactive and reactive, they are lacking an explanation of how the different tools are designed. Another survey by Wang and He et al. [157] reviews 6 kinds of vulnerability detection methods and privacy protection techniques in 3 platforms (i.e., Ethereum, Hyperledger fabric and Corda), and summarizes several commonly used tools for each method. Di Angelo et al. [58] investigate 27 analysis tools of Ethereum smart contracts regarding availability, maturity level, methods employed, and detection of security issues. They examine the availability and functionality of the tools and compare their characteristics in a structured manner. In comparison, we carry out a multi-dimensional classification of 133 solutions and take into account different aspects of threat mitigation. Besides, we also analyze different defense mechanisms through their architecture. Furthermore, Samreen et al. [135] review some detection tools and discuss eight vulnerabilities by analyzing past exploitation cases. Ni et al. [124] propose a three-layered threat model for smart contract security and introduce 15 major vulnerabilities of Ethereum at three levels: programming language, virtual machine, and blockchain. They also summarize and compare the three most

[3]Solidity is an object-oriented programming language used mostly for writing Ethereum smart contracts.
[4]The Ethereum Virtual Machine (EVM) is a software platform for executing Ethereum smart contracts. All smart contracts are compiled into bytecode and run on the EVM of all Ethereum nodes.
[5]https://ethereum.stackexchange.com/

commonly used vulnerability mitigation techniques, viz., fuzzing, symbolic execution, and formal verification. Li et al. [99] survey the security threats of blockchain and enumerate 6 real attack cases. They also review the security enhancement solutions for blockchain by introducing 5 commonly used defense tools. In contrast, we categorize defenses in 5 orthogonal dimensions and compare 133 commonly used solutions. Praitheeshan et al. [130] review the security of Ethereum smart contracts through 16 types of security vulnerabilities, 19 software security issues, and 3 defense methods. For each defense method, they list several common tools but do not compare the different methods and tools. In contrast, we summarize 5 more vulnerability core methods and compare them through 5 dimensions. Moreover, we also construct a compact vulnerability map that contains 37 known vulnerabilities to summarize the vulnerability-addressing ability of 38 classes of threat mitigation solutions.

There are several studies that delve into a specific defense method (e.g., formal verification). Tolmach et al. [146] scrutinize formal models and specifications of smart contracts. They categorize the specifications of smart contracts in various application domains and propose a four-layered framework to classify smart contract analysis methods. After that, they summarize the tools for formal verification and group them based on the utilized techniques. In addition, the authors also discuss the difficulties in smart contract verification and development. Similarly, Singh et al. [139] conduct a systematic survey about current formalization research on all smart contract-enabled blockchain platforms by summarizing 35 studies between 2015 and 2019. However, these studies focus purely on formal verification without examining other types of threat mitigation. On the contrary, we provide eight commonly used vulnerability mitigation core methods and identify future research trends and directions in smart contract threat mitigation.

Unlike the above surveys, which have insufficient technical depth or only focus on a specific method, our survey comprehensively reviews the topic of eight commonly used core methods. Overall, we undertake **four** major steps to shed light on the ever-evolving threat mitigation landscape of smart contracts: 1) comprehensive 5-dimensional classification taxonomy; 2) synthesis of design workflows corresponding to the eight core methods; 3) vulnerability coverage map; and 4) evolutionary analysis with trends and perspectives. The combination of these four steps applied to 133 solutions makes our work the most comprehensive systematization of smart contract threat mitigation to date.

3 METHODOLOGY

In this section, we describe the details of the 4-step methodology that we use in this survey. Fig. 1 depicts these steps, which include: **Step I**: developing the classification taxonomy of smart contract threat mitigation solutions (§3.1); **Step II**: synthesizing the workflows of the core methods of threat mitigation solutions (§3.2); **Step III**: developing the vulnerability coverage map by threat mitigation solutions (§3.3); and **Step IV**: investigating the evolutionary trends and deficiencies of threat mitigation in smart contracts (§3.4). Next, we describe the approaches employed by these four steps in detail.

3.1 Classification Taxonomy

To classify the smart contract threat mitigation solutions, we build a comprehensive taxonomy of threat mitigation, which includes the following five orthogonal dimensions (see Table 1): 1) defense modality, 2) core method, 3) targeted contracts, 4) data mapping, and 5) threat model. We empirically verify that our taxonomy is not only concise but also allows to describe a threat mitigation solution with high accuracy. For example, using our taxonomy, the popular threat mitigation tool Oyente [111] can be accurately described via the following single sentence:

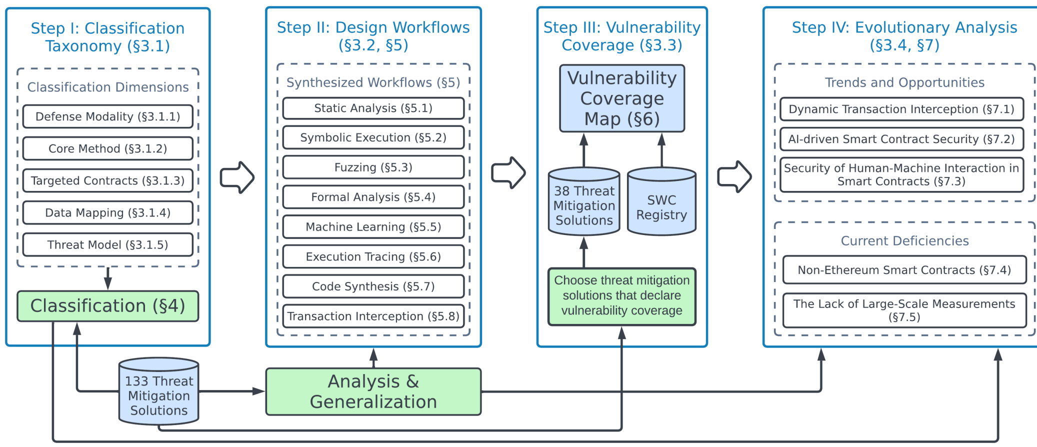


Fig. 1. Four-step methodology of this survey.

Table 1. Smart contract threat mitigation taxonomy

Classification Dimension	Possible Values		Short Notation	
Defense Modality	prevention		PR	
	detection		DET	
	exploration		EXP	
Core Method	static analysis		SA	
	symbolic execution		SE	
	fuzzing		F	
	formal analysis		FA	
	machine learning		ML	
	execution tracing		ET	
	code synthesis		CS	
	transaction interception		TI	
Targeted Contracts	Ethereum		ETH	
	EVM-compatible		EVMc	
	any contract		aC	
	non-Ethereum		nETH	
Data Mapping	**Input**	**Output**	**Input**	**Output**
	source code	report	S	R
	bytecode	source code	B	S
	ABI	bytecode	A	B
	specifications	action	Sp	Ac
	chain data	exploits	C	E
	assembly code	metadata	As	M
Threat Model	vulnerable contract only		VC	
	malicious contract only		MC	
	malicious *or* vulnerable contract		MVC	

> *"Oyente is a security tool based on symbolic execution that detects and reports vulnerabilities in the bytecode of malicious or buggy Ethereum smart contracts."*

Moreover, our taxonomy is cross-platform and general enough to be applied to the future developments of threat mitigation for smart contracts, even when new methods or platforms emerge. Next, we describe all these five dimensions of the threat mitigation taxonomy in detail (§3.1.1–§3.1.5).

3.1.1 Defense Modality. The *defense modality* is the essential philosophy used by a threat mitigation solution to achieve its goals, which is either *prevention*, *detection*, or *exploration*. The prevention methods aim at verifying or enforcing certain security properties of a smart contract. For example, the requirement that if a smart contract accepts cryptocurrency deposits, it must also provide the functionality for cryptocurrency withdrawal, can be used by a solution with the prevention modality as a property to enforce or verify security. The detection methods look for known vulnerabilities in smart contracts. For instance, defense tools that search for reentrancy vulnerabilities in smart contracts pertain to the detection defense modality. The exploration approaches enhance the transparency of a smart contract or associated transactions in order to facilitate security audits. For example, an auditing tool that allows demystifying the call stack of a complicated smart contract, thereby exposing the potential security problems, would belong to the exploration defense modality.

3.1.2 Core Method. The *core method* is the technical approach describing the implementation principles of a given threat mitigation solution. Unlike defense modality, which describes the general philosophy of a solution, the core method describes the implementation methodology utilized by the solution; in other words, the same defense philosophy can be implemented in a number of different core methods. Threat mitigation solutions belonging to the same core method, despite the diversity of implementations, share the same major workflow with possible minor additions. For example, all symbolic execution methods take a smart contract and a set of specifications as an input, utilize an SMT solver, and produce a human-readable report as an output; however, many symbolic execution solutions, in addition to the standard workflow items, add some additional modules and data units. In this work, we build workflows that demonstrate which items are essential and which of them provide an incremental augmentation.

3.1.3 Targeted Contracts. The dimension of *targeted contracts* describes the class of smart contracts that a threat mitigation solution applies to. This dimension is largely shaped by the practical circumstance, in which the vast majority of smart contract threat mitigation solutions target the popular Ethereum platform. Moreover, we notice that within the Ethereum platform, there is very little variety in terms of what kind of Ethereum smart contracts the threat mitigation solutions target. In other words, most solutions target Ethereum, and these Ethereum-based solutions are suitable for any Ethereum contract. Thus, to accurately represent the practical reality of the distribution of smart contract threat mitigation solutions in the dimension of targeted contract, we subdivide this dimension into four classes: *Ethereum smart contracts*, *EVM-compatible smart contracts*, *non-Ethereum smart contracts*, and *any smart contract* (i.e., platform-agnostic). Fig. 2 shows the Venn diagram of the relationships between these classes. Specifically, all Ethereum contracts are EVM-compatible, but there are non-Ethereum platforms that may or may not be EVM-compatible. At the same time, the "any contract" scope would embrace all the types of smart contracts mentioned above, without prioritizing any of them.

3.1.4 Data Mapping. The *data mapping* dimension describes what the input and output of a given threat mitigation solution are. As shown in Table 1, the input of a threat mitigation solution may be a combination of 1) source code; 2) bytecode; 3) application binary interface (ABI); 4) security

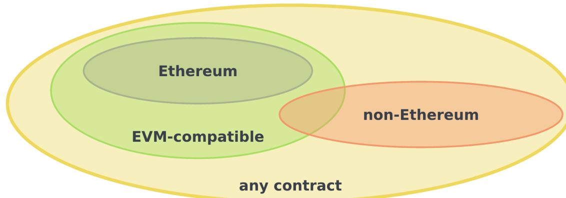


Fig. 2. Venn diagram of relationships between different scopes of smart contracts.

specifications; 5) chain data; or 6) assembly code. The output can be represented by any combination of the following six entities: 1) security report; 2) source code; 3) bytecode; 4) defense action; 5) set of exploits; or 6) metadata. In this work, we use the symbol $\mapsto$ as a convention for data mapping. For example, if the input of a threat mitigation solution is a set of specifications with the source code of the smart contract, and the output is a human-readable report, then we denote such a mapping as Sp,S$\mapsto$R. As we can see, the data mapping dimension allows to concisely and informatively describe the requirements for the input and expectations for the output for a smart contract threat mitigation solution.

3.1.5 Threat Model. The dimension of *threat model* describes the vector(s) of potential attacks that the threat mitigation solution aims to prevent, detect, or explore. We empirically observe that all the smart contract threat mitigation solutions belong to either of the three general threat models: 1) the one with the malicious smart contract; 2) the one in which the smart contract is the victim; and 3) the agnostic model, in which the contract may be either malicious or a victim. For example, the threat mitigation solutions capable of preventing exploitations of the reentrancy vulnerability, responsible for the infamous DAO hack [69], belong to the VC (victim contract) model. Conversely, a tool defending against honeypot smart contracts, which set unexpected traps for hackers attempting to exploit known smart contract vulnerabilities, is a typical example of a threat mitigation tool assuming the malicious contract (MC) threat model. However, some solutions defend against vulnerabilities that can be used both in a malicious or a victim smart contract; in this case, we assign to this solution the malicious-or-vulnerable contract (MVC) model. For example, the SWC-123 vulnerability [16], called *Requirement Violation*, can be both a bug in a vulnerable smart contract or an intentional malicious action of the smart contract developer.

3.2 Workflows of Core Methods

In this survey, not only do we explore *what* the smart contract threat mitigation solutions do, but we also explore, for the first time, *how* these solutions accomplish their goals. In order to do that, we adopt the following approach: for each of the eight core methods introduced in §3.1.2, we synthesize the workflows of all the existing solutions implementing these methods to showcase the mandatory (common for all solutions) and augmented (observed in some solutions) elements. Sections 5.1—5.8 describe the synthesized workflows of all the eight core methods of smart contract threat mitigation. In order to embrace the diverse variety of implementations, we use a uniform set of conventions in the eight workflows. Specifically, we use three types of elements connected with flows (arrows): modules (data processors), data entities, and environments (groups).

3.3 Vulnerability Coverage

The third step of our survey is scrutinizing the vulnerability coverage, i.e., to determine *which known vulnerabilities are detectable and/or preventable by the existing threat mitigation solutions.* To accomplish that, we create a uniform *vulnerability coverage* map using the popular SWC Registry. This task poses two major challenges: i) many threat mitigation solutions do not explicitly or even implicitly declare the set of addressed vulnerabilities; ii) the majority of threat mitigation solutions refer to the existing vulnerabilities using custom names and/or groupings, which often do not correspond to the SWC taxonomy. Here, we select the 38 threat mitigation solutions that explicitly specify the list of targeted vulnerabilities, and then we meticulously translate the declared vulnerability coverage provided by the selected 38 solutions into the SWC conventions.

3.4 Threat Mitigation Evolution

Our final step explores the evolution of the smart contract threat mitigation solutions, as well as the trends and obstacles observed in this area of computer security. Specifically, we explore the adoption and augmentation of new core methods over time. For each threat mitigation solution, we keep track of the publication date as well as the initial release or announcement date, whenever available. Additionally, we analyze the "blind spots" of the existing body of smart contract mitigation solutions — the potentially feasible yet unexplored combinations of approaches that can bring more benefits, especially if a similar combination of approaches has been successful in other more mature areas of computer security. As a result, we make five observations supported by data and evidence. First, we identify that dynamic transaction interception methods of smart contract threat mitigation are gaining momentum in the research community. Second, we show that the smart contract threat mitigation solutions utilizing AI and machine learning have started playing an important role in smart contract defense. Third, we identified the emerging trend for studying human-machine interaction in the domain of smart contracts. Fourth, we confirm that Ethereum smart contracts are over-represented by the threat mitigation solutions, and we discuss likely reasons explaining this phenomenon. Finally, we discuss the necessity for more exploration tools and large-scale measurements for gathering important data about smart contract security, such as the real market value of smart contracts and the traces of choices made by miners and crypto exchanges.

4 THREAT MITIGATION CLASSIFICATION

In this section, we apply the taxonomy developed earlier (§3.1) to describe each of the threat mitigation solutions via the five orthogonal dimensions: threat mitigation modality (§4.1), core method (§4.2), the scope of targeted contracts (§4.3), the input-output data mapping of the solution (§4.4), and the assumed threat model (§4.5). The results of our classification are given in Table 2. Furthermore, we perform a frequency analysis of the results along the five dimensions, and create a visual representation of the distributions of defense modalities, core methods, targeted contracts, and threat models in Fig. 3. In the first column of the table, we assign to each of the threat mitigation solutions a permanent Security Threat Mitigation (STM) registry identifier in the `STM-XXX` format. The second column provides the name of the tool implementing the solution along with its reference; if a solution does not have a common name, we refer to the solution by its authors (e.g., Ivanov et al.). In columns 3–7, we provide the values along the five classification dimensions for each of the 133 threat mitigation solutions. Furthermore, to keep the data in this table up to date and handy, we deploy the Smart Contract Threat Mitigation Registry (STM Registry) at https://stmregistry.io/.
Selection Method of the Threat Mitigation Solutions. For this survey, we select 133 threat mitigation solutions, encompassing both academic research projects (e.g., Securify [149], Oyente [111] and commercial non-academic efforts (e.g., OpenZeppelin Contracts [9], MythX [7]). To assure the quality of our study, we use the following four criteria for selecting threat mitigation solutions:

(1) *Implementation.* We select only solutions that are implemented and evaluated, either as a proof-of-concept (PoC) prototype or in the form of a final product.

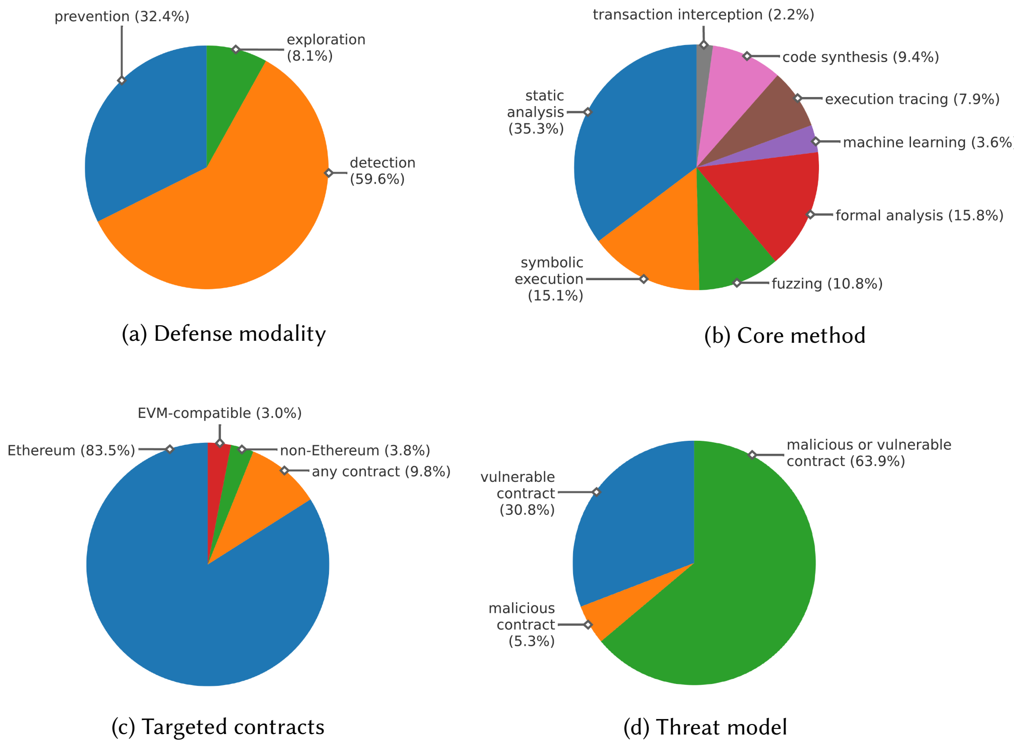
Table 2. Classification of threat mitigation solutions based on the proposed taxonomy.

STM registry code	Threat mitigation solution	*Classification Criteria (Dimensions)*[†] Defense Modality	Core Method	Targeted Contracts	Data Mapping	Threat Model
STM-001	Oyente [111]	DET	SE	ETH	B ↦ R	MVC
STM-002	Mythril [120]	DET	SE	ETH	B ↦ R	MVC
STM-003	Securify [149]	DET+PR	SA	ETH	B, S ↦ R, M	MVC
STM-004	Maian [125]	DET	SE	ETH	B ↦ R	MVC
STM-005	Manticore [119]	DET	SE	ETH	B ↦ R	MVC
STM-006	KEVM [83]	EXP	FA	ETH	B ↦ R	MVC
STM-007	ZEUS [92]	PR	SE	ETH	B ↦ R	MVC
STM-008	Sereum [132]	DET	ET, TI	ETH	C ↦ R	VC
STM-009	ECFChecker [78]	PR	ET, TI	ETH	C ↦ R	VC
STM-010	teEther [96]	DET	SE	ETH	B ↦ E	VC
STM-011	Hydra [42]	PR	CS	ETH	S ↦ B	VC
STM-012	Erays [175]	EXP	SA	ETH	B ↦ M	MVC
STM-013	TokenScope [55]	DET	ET	ETH	C ↦ R	MVC
STM-014	Osiris [147]	DET	SE	ETH	B ↦ R	VC
STM-015	Vandal [44]	DET	SA	ETH	B ↦ R	MVC
STM-016	FSolidM [115]	PR	CS	ETH	S, Sp ↦ S	VC
STM-017	ContractFuzzer [89]	DET	F	ETH	A, B ↦ R	VC
STM-018	S-GRAM/Ether* [106]	DET	SA	ETH	S ↦ R	MVC
STM-019	MadMax [74]	DET	SA	ETH	B ↦ R	MVC
STM-020	SmartCheck [145]	DET	SA	ETH	S ↦ R	MVC
STM-021	ReGuard [105]	DET	F	ETH	S ↦ R, E	VC
STM-022	GASPER [54]	DET	SA	ETH	B ↦ R	MC
STM-023	Grishchenko et al. [77]	EXP	FA	ETH	B ↦ M	MVC
STM-024	Lolisa [164]	PR	FA	ETH	S ↦ R	MVC
STM-025	SASC [173]	EXP	SA	ETH	S ↦ R	MVC
STM-026	Chen et al. [56]	DET	ET, SA	ETH	C, S ↦ R	MC
STM-027	Solidity*/EVM* [36]	PR	SA, FA	ETH	S ↦ R	MVC
STM-028	Amani et al. [26]	PR	SA, FA	ETH	B, Sp ↦ R	MVC
STM-029	Model-Checking [121]	PR	FA	ETH	Sp, S ↦ R	MVC
STM-030	EtherTrust [76]	DET	SA	ETH	B ↦ R	MVC
STM-031	Flint [137]	PR	CS	ETH	Sp ↦ S	VC
STM-032	HoneyBadger [148]	DET	SA, SE	ETH	B ↦ R	MC
STM-033	ILF [81]	DET	F, ML	ETH	B, S ↦ R	MVC
STM-034	VeriSolid [116]	PR	FA, CS	ETH	S, Sp ↦ S	VC
STM-035	solc-verify [79]	PR	SA	ETH	S, Sp ↦ R	MVC
STM-036	Slither [62]	DET	SA	ETH	S ↦ R	MVC
STM-037	sCompile [49]	DET	SE	ETH	B ↦ R	MVC
STM-038	NPChecker [155]	DET	SA	ETH	B ↦ R	MVC
STM-039	BitML [31]	PR	SA	nETH	C, Sp ↦ R, E	VC
STM-040	CESC [101]	DET	SA	ETH	B ↦ R	VC

[†] DET— detection; PR — prevention; EXP — exploration; SA — static analysis; SE — symbolic execution; F — fuzzing; FA — formal analysis; ML — machine learning; ET — execution tracing; CS — code synthesis; TI — transaction interception; S — source code; B — bytecode; A — ABI; Sp — specifications; C — chain data; As — assemb. code; R — report; Ac — action; E — exploits; M — metadata; ETH — Ethereum; nETH — non-Ethereum; EVMc — EVM-comp.; aC — any contract; VC — vuln. contract; MC — mal. contract; MVC — mal. or vuln. contract.

STM registry code	Threat mitigation solution	*Classification criteria* Defense Modality	Core Method	Targeted Contracts	Data Mapping	Threat Model
STM-041	EasyFlow [72]	DET	SA	ETH	C ↦ R	VC
STM-042	Vultron [154]	DET	CS	ETH	S ↦ R	VC
STM-043	SAFEVM [23]	PR	SA	ETH	S,B ↦ R	MVC
STM-044	EthRacer [94]	DET	F	ETH	B,C ↦ R	MVC
STM-045	SolidityCheck [168]	DET	SA	ETH	S ↦ R	MVC
STM-046	EVMFuzz [70]	DET	F	ETH	S ↦ R	MVC
STM-047	EVulHunter [131]	DET	SA	ETH	As ↦ R	VC
STM-048	GasFuzz [112]	DET	F	ETH	B ↦ R	MVC
STM-049	NeuCheck [109]	DET	SA	ETH	S ↦ R	MVC
STM-050	SolAnalyser [22]	DET	SA	ETH	S ↦ R	MVC
STM-051	SoliAudit [103]	DET	F	ETH	S ↦ R	MVC
STM-052	MPro [171]	DET	SA,SE	ETH	S ↦ R	MVC
STM-053	Li et al. [100]	PR	FA	ETH	S ↦ R	VC
STM-054	Gastap [24]	PR	SA	ETH	S,B,As ↦ R	MVC
STM-055	Momeni et al. [118]	DET	ML	ETH	S ↦ M	MVC
STM-056	KSolidity [90]	EXP	FA	ETH	S ↦ M	MVC
STM-057	VerX [129]	PR	SE,CS	ETH	S ↦ R	MVC
STM-058	VeriSmart [141]	DET	SA	ETH	S ↦ R	MVC
STM-059	TxSpector [167]	EXP	SA,ET	ETH	C ↦ R	MVC
STM-060	Zhou et al. [174]	EXP	SA	ETH	C ↦ R	MVC
STM-061	ETHBMC [68]	DET	SE	ETH	B ↦ R	MVC
STM-062	SODA [53]	DET	TI	EVMc	C ↦ R,Ac	VC
STM-063	Ethor [136]	PR	SA,FA	ETH	B ↦ R	MVC
STM-064	ÆGIS [65]	DET	TI	ETH	C ↦ R,Ac	VC
STM-065	SafePay [102]	DET	SE	ETH	S,B ↦ R	VC
STM-066	Solar [64]	DET	CS,SE	ETH	Sp ↦ R	VC
STM-067	EVMFuzzer [71]	DET	F	EVMc	Sp ↦ R	MVC
STM-068	ModCon [107]	DET+PR	F	aC	S ↦ R	MVC
STM-069	Harvey [162]	DET	F	ETH	S ↦ R	MVC
STM-070	Solythesis [98]	PR	CS	ETH	S ↦ S	VC
STM-071	Ethainter [43]	DET	SA	ETH	S ↦ R	MVC
STM-072	sFuzz [123]	DET	F	ETH	B,A ↦ R	MVC
STM-073	Seraph [165]	DET	SE	aC	S ↦ R	MVC
STM-074	Clairvoyance [166]	DET	SA	ETH	S ↦ R	VC
STM-075	Artemis [151]	DET	SE	ETH	B ↦ R	MVC
STM-076	Echidna [75]	DET	F	ETH	B,Sc ↦ R,M	MVC
STM-077	EShield [163]	PR	CS	ETH	B ↦ B	VC
STM-078	SMARTSHIELD [172]	DET	CS	ETH	B ↦ B,R	VC
STM-079	ETHPLOIT [170]	DET	F	ETH	S ↦ E	MVC
STM-080	Cecchetti et al. [48]	PR	CS	aC	S ↦ R	VC
STM-081	EthScope [160]	DET	ET	ETH	C ↦ R	MC
STM-082	ContractWard [156]	DET	ML	ETH	S ↦ R,M	MVC
STM-083	RA [57]	DET	SA,SE	ETH	B ↦ R	VC
STM-084	Camino et al. [46]	DET	SA	ETH	B ↦ R	MC
STM-085	OpenBalthazar [29]	DET	SA	ETH	S ↦ R	MVC
STM-086	sGUARD [122]	PR	SA,FA	ETH	B ↦ R	VC
STM-087	SmartPulse [142]	PR	SA	ETH	S,Sp ↦ R	MVC

STM registry code	Threat mitigation solution	*Classification criteria* Defense Modality	Core Method	Targeted Contracts	Data Mapping	Threat Model
STM-088	SeRIF [47]	DET	FA	aC	S ↦ R	VC
STM-089	EVMPatch [133]	DET	CS	ETH	B, C ↦ B	VC
STM-090	Perez et al. [128]	EXP	ET	ETH	C ↦ R	VC
STM-091	DEFIER [143]	DET	ET	ETH	C ↦ R	MVC
STM-092	SmarTest [140]	DET	SE, SA	ETH	B ↦ R, E	MVC
STM-093	EOSAFE [82]	DET	SA	nETH	B ↦ R	MVC
STM-094	Ivanov et al. [88]	PR+DET	SA	ETH	S ↦ R	MC
STM-095	ConFuzzius [66]	DET	F	ETH	B, A ↦ R	MVC
STM-096	Huang et al. [86]	DET	SA	ETH	B ↦ R	VC
STM-097	STC/STV [85]	PR	SA	ETH	S ↦ R	VC
STM-098	Horus [67]	DET	ET	ETH	C ↦ R	MVC
STM-099	BlockEye [152]	DET	ET, TI	ETH	C ↦ R	MVC
STM-100	Sailfish [39]	DET	SE, SA	ETH	B ↦ R	MVC
STM-101	DeFiRanger [161]	DET	SA	ETH	C ↦ R	VC
STM-102	ESCORT [110]	DET	ML	ETH	B, Sp ↦ R	MVC
STM-103	DefectChecker [52]	DET	SE	ETH	B ↦ R	MVC
STM-104	Hu et al. [84]	DET	SA, ML	ETH	B ↦ R	MVC
STM-105	HFContractFuzzer [59]	DET	F	nETH	S ↦ R	VC
STM-106	Solidifier [28]	PR	FA	ETH	S ↦ R	VC
STM-107	SafelyAdministrated [87]	PR	CS, ML	ETH	S ↦ S	MC
STM-108	EXGEN [91]	DET	SE	aC	S ↦ R	VC
STM-109	EtherProv [104]	DET	SA	ETH	S ↦ B, M	MVC
STM-110	Abdellatif et al. [20]	PR	FA	aC	C ↦ R	MVC
STM-111	Bai et al. [32]	PR	FA	aC	Sp ↦ R	MVC
STM-112	Bigi et al. [37]	PR	FA	aC	Sp ↦ R	MVC
STM-113	Findel [38]	PR	CS	aC	Sp ↦ S	VC
STM-114	ContractLarva [61]	PR	CS	ETH	S, Sp ↦ S	MVC
STM-115	Le et al. [97]	PR	FA	aC	S ↦ R	MVC
STM-116	Solicitous [114]	PR	SA	ETH	S ↦ B	MVC
STM-117	VeriSol [158]	PR	FA	EVMc	S ↦ R	VC
STM-118	SmartCopy [63]	DET	F	ETH	B, A ↦ R	VC
STM-119	WANA [153]	DET	SE	aC	B ↦ R	MVC
STM-120	E-EVM [126]	EXP	ET	ETH	C ↦ R	MVC
STM-121	AMEVulDetector [108]	DET	ML	aC	S ↦ R	MVC
STM-122	Javadity [21]	PR	CS	EVMc	S ↦ S	MVC
STM-123	Alqahtani et al. [25]	PR	SA	aC	S ↦ B	MVC
STM-124	Bartoletti et al. [33]	PR	FA	nETH	S ↦ R	MVC
STM-125	Beckert et al. [34]	PR	FA	nETH	S ↦ R	VC
STM-126	SmartInspect [40]	EXP	SA	ETH	S, C ↦ R	MVC
STM-127	CPN [60]	EXP	SA	ETH	S, B ↦ R	MVC
STM-128	Hajdu et al. [80]	PR	FA, SA	ETH	S ↦ R	MVC
STM-129	Kongmanee et al. [95]	PR	FA	ETH	Sp ↦ M	MVC
STM-130	EVM* [113]	PR	TI	ETH	B, C ↦ R, Ac	MVC
STM-131	OpenZeppelin Contracts [9]	PR	CS	ETH	S ↦ S	VC
STM-132	MythX [7]	DET	many	ETH	B ↦ R	MVC
STM-133	Contract Library [2]	DET+PR	CS	ETH	B ↦ R	MVC



(a) Defense modality (b) Core method

(c) Targeted contracts (d) Threat model

Fig. 3. Distribution of threat mitigation methods by four criteria: defense modality, core method, targeted contracts, and threat model.

(2) *Publication.* For academic research projects, we search for the papers published or accepted at a reputable peer-reviewed venue.
(3) *Impact.* We select solutions that deliver specific improvements or other unique qualities compared to the state-of-the-art solutions.
(4) *Novelty.* Not only do we consider the fact of improvement or impact, but we also consider the presence of technical novelty, i.e., a specific innovation that leads to the improvement.

In some cases, we include threat mitigation solutions that do not meet all the four above criteria, such as the academic project Vandal [44], which has never been published at a peer-reviewed venue. However, we include this work in our survey because it is widely adopted and cited.

Lessons learned: There are more than 200 claims of smart contract threat mitigation solutions. Yet, our thorough manual examination reveals various problems associated with some of them. For example, we observed that sometimes two research papers refer to the same implementation (e.g., poster or journal extension articles). In the end, 133 instances have been selected to represent the body of smart contract threat mitigation solutions. Therefore, manual scrutiny of each work is required.

4.1 Threat Mitigation Modalities

A threat mitigation modality is a philosophy that a smart contract threat mitigation method employs to address security issues of a smart contract. The threat mitigation solutions that employ the detection modality are designed to identify vulnerabilities in smart contracts. Some of them (e.g., Oyente [111], Securify [149], Vandal [44], and Mythril [120]) target several groups of vulnerabilities. Other detection-based threat mitigation solutions focus on specific classes of vulnerabilities, such as Sereum [132], which detects only reentrancy vulnerabilities (SWC-107 [13]). Another narrow-focused detection tool is VeriSmart [141], which detects arithmetic bugs only. Overall, we note that the detection solutions that focus on specific vulnerabilities tend to deliver improved detection rates compared to the solutions targeting multiple vulnerabilities.

The solutions belonging to the prevention modality validate some safety properties or rules. ZEUS [92] provides eight semantic rules that are used as part of an abstract assertion language for specifying safety properties for ensuring that a smart contract is free of certain vulnerabilities (e.g., reentrancy, unchecked send, integer overflow, etc.). Another salient representation of a prevention solution is SmartPulse [142], which creates a linear temporal logic (LTL) language, called SmartLTL, for expressing temporal safety properties in smart contracts and enforcing them with the SmartPulse verifier.

The exploration modality solutions do not detect vulnerabilities or enforce safety properties; instead, they reveal previously concealed data that facilitates human-based or automated auditing of a smart contract. Erays [175] is a tool for reverse-engineering of smart contracts that converts a bytecode of a smart contract into pseudocode-like metadata. TxSpector [167] is another exploration solution, which is a transaction processing framework that identifies the executed attacks in smart contract execution traces.

Some threat mitigation solutions adhere to a hybrid detection+prevention modality, which means that they can detect existing vulnerabilities, as well as enforce security properties. Securify [149] not only checks the compliance with security patterns but also detects violations of patterns associated with specific vulnerabilities, such as reentrancy and restricted transfer. Another threat mitigation solution with a hybrid detection+prevention modality is ModCon [107], which is a smart contract testing tool that generates a list of states and transitions between these states, thereby enabling further identification of vulnerabilities and confirmation of security properties.

Fig. 3a shows the breakdown of the three defense modalities among the 133 threat mitigation solutions. As we can see, 81 (59.6%) of all the threat mitigation solutions employ the detection modality, 44 (32.4%) use the verification modality, and the remaining 11 (8.1%) belong to the exploration modality. Some threat mitigation solutions exhibit a hybrid modality (e.g., `DET+PR` — detection combined with prevention), in which case we identify and assume the predominant modality for the statistical analysis, or we count both modalities in cases when it is impossible to detect the predominant one — which explains the 136 total modalities considered, despite the fact that they correspond to 133 threat mitigation solutions.

4.2 Core Methods

The core method describes *how* a threat mitigation solution addresses the security issues of a smart contract. In other words, the core method defines the implementation approach, choice of algorithms, and internal data processing model of a threat mitigation solution. By scrutinizing all the 133 smart contract threat mitigation solutions, we identify eight distinct core methods: 1) static analysis; 2) symbolic execution; 3) fuzzing; 4) formal analysis; 5) machine learning; 6) execution tracing; 7) code synthesis; and 8) transaction interception.

Static analysis solutions extract data from smart contracts in order to detect vulnerabilities or confirm safety properties. Most static analysis solutions adhere to the detection modality (e.g., Security [149], S-GRAM [106], MadMax [74], SmartCheck [145]). However, some static analysis solutions enforce policies instead of detecting vulnerabilities (e.g., solc-verify [79], BitML [31], GasTap [24], Solicitious [114]). Moreover, we notice that the static analysis core method is often coupled with some other methods. Solidity* [36], Amani et al. [26], Ethor [136], and sGUARD [122] use static analysis together with formal analysis. Also, static analysis is often used together with the symbolic execution core method, as we can see in HoneyBadger [148], MPro [171], SmarTest [140], and Sailfish [39].

Symbolic execution methods execute a smart contract with symbolic parameters instead of real ones — in order to make conclusions regarding some security properties of smart contracts (e.g., the range of values that make a certain condition true). Oyente [111], Mythril [120], Maian [125], Manticore [119], ZEUS [92], Osiris [147], teEther [96] are popular solutions employing the symbolic execution core method. Similar to static analysis, symbolic execution is also often coupled with other core methods. VerX [129] and Solar [64] use symbolic execution to guide code synthesis. The solution by Hu et al. [84] takes advantage of both symbolic execution and machine learning for detecting smart contract vulnerabilities.

Fuzzing methods perform smart contract testing by iteratively generating test cases that are likely to reveal vulnerabilities. ContractFuzzer [89] uses the abstract binary interface (ABI) of the smart contract to facilitate the generation of fuzzing inputs. Harvey [162] is a smart contract tester based on greybox fuzzing, which is a middle-ground solution between the absence of code analysis (blackbox fuzzing) and full code execution (whitebox fuzzing); specifically, greybox fuzzing assumes a lightweight (compared to symbolic execution) analysis of the code execution paths. Confuzzius [66] is a smart contract fuzzer that uses a combination of genetic algorithms and constraint solving. Overall, fuzzing threat mitigation solutions utilize a diverse variety of predictive methods for balancing accuracy and performance.

Formal analysis methods convert a smart contract into a formal representation and run a solver over this representation to prove or disprove some security properties. Most solutions employing the formal analysis core method belong to either the prevention defense modality (e.g., Lolisa [164], Model-Checking [121], Li et al. [100], Solidifier [28], VeriSol [158]) or the exploration modality (e.g., KEVM [83], Grishchenko et al. [77]). However, SeRIF [47], which primary purpose is defense against reentrancy, demonstrates that the formal analysis can also be used for targeting vulnerabilities.

Machine learning methods extract features from smart contracts and train models for detecting vulnerabilities. The smart contract threat mitigation solutions utilizing the machine learning core method are ContractWard [156], ESCORT [110], AMEVulDetector [108], and the solution by Momeni et al. [118]. In §7.2, we conduct an in-depth discussion about the evolutionary perspective of machine learning in smart contract security.

Execution tracing and transaction interception core methods constitute the transaction-based methods of smart contract threat mitigation. The execution tracing methods examine the runtime traces of the actual transactions submitted to a smart contract in order to detect vulnerabilities, verify safety properties, or facilitate manual auditing. TokenScope [55], EthScope [160], DEFIER [143], Horus [67], BlockEye [152], E-EVM [126] are instances of "pure" execution tracing methods (i.e., not combined with other methods).

Code synthesis threat mitigation solutions aim at generating vulnerability-free smart contract code resistant to attacks. Hydra [42] is a framework that generates bug bounties for smart contracts using the N-of-N version programming (NNVP) principle. FSolidM [115] is a framework for designing secure smart contracts as finite state machines (FSMs) and converting them into Solidity code. Solythesis [98] is a source-to-source Solidity compiler that instruments the input source code with additional instructions for validation of security-sensitive invariants.

Transaction interception solutions dynamically observe the transaction pool of a blockchain node in order to prevent the execution of malicious or unsafe transactions. These solutions are represented by SODA [53], and EVM* [113]. However, we observe that execution tracing is often combined with other core methods. Sereum [132] and ECFChecker [78] combine execution tracing with transaction interception, while TxSpector [167] and the Ponzi scheme detection solution by Chen et al. [56] utilize trace execution combined with static analysis.

Fig. 3b shows the distribution of the eight core methods among the 133 threat mitigation solutions. Specifically we found 49 (35.3%) static analysis tools, 21 (15.1%) symbolic execution methods, 15 (10.8%) fuzzing tools, 22 (15.8%) formal analysis tools, 5 (3.6%) machine learning solutions, 11 (7.9%) execution tracing tools, 13 (9.4%) code synthesis tools, and 3 (2.2%) transaction interceptors. Notably, some threat mitigation solutions employ a combination of the aforementioned core methods; in this case, we recognize all the methods evolved in Table 2, yet for the purpose of counting and frequency analysis, we reduce the combination of core methods to the predominant core method, if there is one. If it is impossible to identify the predominant core method, we count all of them, which explains that the total count of instances of core methods slightly exceeds the number of the threat mitigation solutions surveyed in this work.

4.3 Targeted Contracts

Each of the threat mitigation solutions assumes a type of targeted smart contract. Some solutions target general groups of smart contracts, such as Ethereum or even all possible contracts, while some other solutions may target a single specific smart contract instance. Oyente [111], Mythril [120], Securify [149], Sereum [132], Vandal [44], OpenZeppelin Contracts [9], MythX [7], Contract Library [2], and many other popular threat mitigation solutions are strictly Ethereum-based. Some solutions are EVM-compatible, which means that they are compatible with *but not limited by* the Ethereum smart contracts. SODA [53], VeriSol [158], and Javadity [21] are EVM-compatible solutions. Some solutions are universal in terms of the scope of targeted contracts; although they might not support *any* type of smart contracts (e.g., the ones that are not Turing-complete), they do not limit their scope to a specific group either. Such solutions are ModCon [107], Seraph [165], SeRIF [47], EXGEN [91], and the information flow control solution by Cecchetti et al. [48]. Some threat mitigation solutions target a specific non-Ethereum platform. BitML [31] targets Bitcoin smart contract overlays, EOSAFE [82] targets the smart contracts on the EOS blockchain [3], and HFContractFuzzer [59] targets the Hyperledger Fabric platform [27].

To make sense of this diverse spectrum, we group the targeted smart contracts into four types, as described in § 3.1.3. Fig. 3c shows the distribution of different groups of targeted contracts among the threat mitigation methods. Specifically, we discover that as many as 111 (83.5%) solutions target Ethereum contracts, 13 (9.8%) are suitable for any contract (including Ethereum, but not specifying it), 5 (3.8%) aim for some non-Ethereum contracts (e.g., Hyperledger Fabric), and 4 (3.0%) target EVM-compatible contracts (e.g., Polygon [10], RSK [11]).

4.4 Data Mapping

Next, we explore the design-specified inputs and outputs of each of the threat mitigation solutions. Most smart contract threat mitigation solutions assume a smart contract as an input, either as bytecode, source code, or as part of the chain data. Oyente [111], Mythril [120], Vandal [44], ZEUS [92], teEther [96], and Osiris [147] are solutions that take bytecode as a smart contract input. Hydra [42], S-GRAM [106], SmartCheck [145], VerX [129], VeriSmart [141], and SeRIF [47] are solutions that assume source code as the input. Sereum [132], ECFChecker [78], TokenScope [55], EasyFlow [72], TxSpector [167], and EthScope [160] are the threat mitigation solutions that read smart contract information from the chain data, i.e., stored copy of the blockchain.

Some threat mitigation solutions use a combination of bytecode and source code as an input, e.g., Securify[6] [149], SAFEVM [23], Gastap [24], SafePay [102], and CPN [60]. Other solutions, in addition to a smart contract, also take a set of manual specifications as an input, as we see it in FSolidM [115], Model-Checking [121], VeriSolid [116], solc-verify [79], BitML [31], SmartPulse [142], ESCORT [110], and ContractLarva [61]. Moreover, a smart contract is not always used as an input of a threat mitigation solution. For instance, Flint [137], Solar [64], EVMFuzzer [71], Findel [38], and the solution by Kongmanee et al. [95] assume a set of specifications as the only input.

Most threat mitigation solutions produce a human-readable report as an output, e.g., Oyente [111], Mythril [120], Maian [125], Manticore [119], ZEUS [92], and Sereum [132]. However, some solutions produce machine-readable metadata (e.g., a formal model) in lieu of a human-readable report, which can be observed in Erays [175], the solution by Grishchenko et al. [77], the solution by Momeni et al. [118], KSolidity [90], and the solution by Kongmanee et al. [95].

Table 2 shows that the majority of the threat mitigation solutions (82.7%) produce a human-readable report as an output, and for 78.19% of the solutions, the security report is the only output. Notably, only 4 (3.0%) of all the threat mitigation solutions result in an action (e.g., stopping a malicious transaction), which is indicative of the predominance of the static methodology in the smart contract defense, which is further discussed in §7.1.

One important property of data mapping is that it often provides fine-tuned information that cannot be inferred from the workflow of the corresponding core method. For example, the workflows of smart contract threat mitigation solutions often specify "smart contract" as one of the inputs. However, a smart contract can have several representations: source code, bytecode, deployed address, etc. In this work, we extract the specific meaning of the "smart contract" and represent it accordingly in the data mapping.

4.5 Threat Model

Finally, we describe all the threat mitigation solutions through the general description of their assumed threat models. In other words, the threat model specifies the source of the threat, identifies the victim(s), and defines the intent. We generalize all the threat models by subdividing them into three major groups: victim contract, malicious contract, and hybrid malicious *or* victim contract. Sereum [132], teEther [96], Hydra [42], Osiris [147], SODA [53], ÆGIS [65], EVMPatch [133], SeRIF [47], and OpenZeppelin Contracts [9] are threat mitigation solutions with the vulnerable contract threat model. Solutions with malicious contract threat models are the Ethereum honeypot detector HoneyBadger [148], GASPER [54], and the social engineering attack detector by Ivanov et al. [88]. Most threat mitigation solutions, however, are threat vector agnostic, i.e., they are capable of defending against malicious smart contracts, as well as protecting vulnerable contracts. Securify [149], Oyente [111], ZEUS [92], SmartCheck [145], SmartPulse [142], SmarTest [140], and MythX [7] are solutions with a bidirectional vector (malicious *or* victim contract) threat model.

Fig. 3d shows the breakdown of different threat models among the threat mitigation methods. We find that 41 (30.8%) methods assume vulnerable contracts, 7 (5.3%) imply the malicious contract model, and 85 (63.9%) assume both these vectors. As we can see, the pure malicious smart contract threat model is underrepresented among the threat mitigation solutions, which suggests that attacks on smart contracts are generally perceived as more important than the cases of malicious contracts attacking users. This finding is corroborated by the study by Zhou et al. [174], which confirms that the popularity of the honeypot vulnerability, associated with the malicious smart contract

[6]Source code is optional in Securify.

modality, is fourth after call injection, call-after-destruct, and airdrop-hunting vulnerabilities, which all assume the victim smart contract threat model.

5 DESIGN WORKFLOWS OF THREAT MITIGATION METHODS

In this section, we scrutinize the designs of the threat mitigation solutions by synthesizing the uniform workflows for all the eight core methods, i.e., static analysis (§5.1), symbolic execution (§5.2), fuzzing (§5.3), formal analysis (§5.4), machine learning (§5.5), execution tracing (§5.6), code synthesis (§5.7), and transaction interception (§5.8). Figs. 4—11 depict the workflows of the eight core methods. Each of these eight workflows utilizes a set of uniform elements: modules, data entities, flows (arrows), and environments. This set of elements allows us to concisely summarize and demystify the wide variety of implementations of smart contract threat mitigation solutions using the aforementioned set of uniform conventions.

The modules (green rectangles) represent items that *do* something, i.e., algorithms, data filters, etc. Modules can be mandatory, i.e., pertaining to any solution with the given core method (solid borders) or optional/augmenting, i.e., implemented by some solutions employing the given core method (dashed borders). The data entities (blue rectangles) represent pieces of data or abstract data structures. The flows, depicted as arrows, show data or execution transitions. Environments (red rectangles) allow grouping of certain elements into single logical modules.

Lessons learned: By manually examining the workflows of all the 133 threat mitigation solutions, we learned that every component exhibits a certain degree of generalization. For example, an element called “smart contract” is a more general form of what could also be denoted as “source code” or “bytecode”. Thus, one of the challenges we face when synthesizing the workflows is to equate the generalizations of similar workflow elements.

5.1 Static Analysis Workflow

The static analysis methods apply automated data filtering and syntax analysis techniques to the input. Static analysis methods detect vulnerabilities by extracting information (facts) from the source code or bytecode of a smart contract. Fig. 4 shows the general workflow of static analysis methods.

The static analysis methods take bytecode (e.g., Erays [175], Vandal [44], MadMax [74]) or source code (e.g., S-GRAM [106], SmartCheck [145], Slither [62]) of a smart contract as an input, while some solutions also analyze previously executed transactions gathered from the chain data (e.g., EasyFlow [72], Zhou et al. [174]). A large part of the static analysis process is devoted to constructing a model in the form of one or a set of abstract data structures (ADS) that constitute a suitable (and efficient) input for the static analyzer. Control flow graph (CFG) is a popular type of such an ADS, which is utilized by Securify [149], Erays [175], and Vandal [44], to name a few. The built model, data (in the form of some intermediate representation, e.g., a graph), and a set of pre-defined or user-specified specifications are then directed to the static analyzer, which produces a human-readable security assessment report.

5.2 Symbolic Execution Workflow

Symbolic execution methods [93] simulate the execution of a smart contract in a way that the actual inputs are replaced with special traceable symbolic parameters. Fig. 5 depicts the general workflow of symbolic execution methodology. These methods use smart contract bytecode and a set of specifications as an input. In some cases, the specifications are part of the tool (e.g., Oyente [111], Mythril [120], teEther [96], Osiris [147]), in other cases, the specifications are expected to be

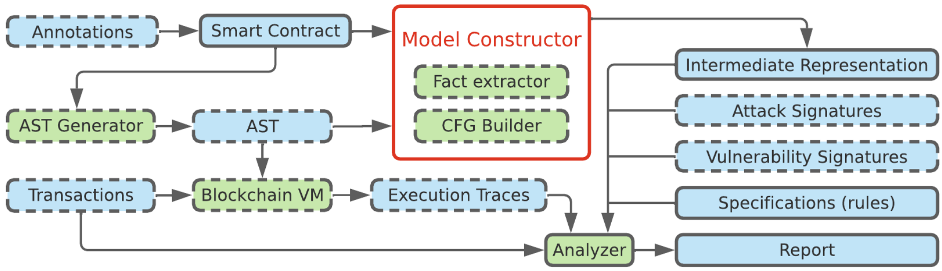


Fig. 4. Workflow of the static analysis core method.

provided by the user (e.g., Maian [125]). Symbolic execution methods execute smart contracts with traceable (symbolic) parameters in lieu of actual inputs, which allows to prove or disprove some presumptions about smart contracts. Specifically, symbolic execution can answer questions about the possibility of execution of a certain block of code (reachability), the ability to invoke a certain execution path, or the ability to satisfy certain constraints. Similar to static analysis, symbolic execution often involves building a search-efficient data structure, such as CFG, as well as extracting facts and features from the input. However, unlike static analysis, the symbolic execution methods run the code instead of analyzing its syntax. All the existing symbolic execution solutions surveyed in this work employ the Z3 [19] SMT solver.

Some symbolic execution solutions use certain augmentations to the basic design by adding additional features. Oyente [111], teEther [96], SafePay [102], Artemis [151], and DEFECTCHECKER [52] process the smart contract to build a CFG. Another augmentation observed in symbolic execution solutions is the production of exploits (sample inputs revealing vulnerabilities), as can be seen in teEther [96] and EthBMC [68]. Moreover, some symbolic execution methods perform a preliminary analysis (preprocessing) for generating guidance data facilitating the symbolic execution. SmarTest [140] guides symbolic execution with a language-based model in order to achieve higher accuracy and reduce the rate of timeouts.

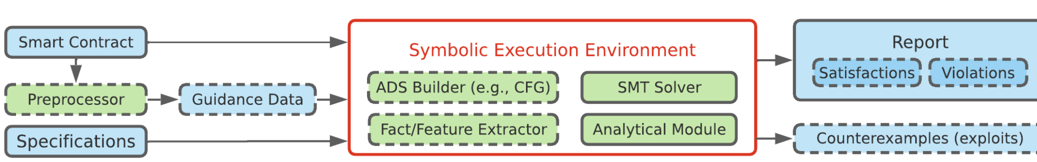


Fig. 5. Workflow of the symbolic execution core method.

5.3 Fuzzing Workflow

Fuzzing methods use various techniques for generating subsets of test inputs that could reveal vulnerable execution paths in smart contracts. Fig. 6 shows how the fuzzing core method works in smart contracts. Fuzzing tools perform iterative testing of a smart contract by generating test cases and adjusting these cases via a feedback loop. The execution of smart contracts is performed by the fuzzing engine, which is either a stand-alone code interpreter or an instrumented (i.e., modified with a custom code) blockchain virtual machine. Fuzzing techniques allow to address the two notorious problems associated with software testing — input ranges and path explosion. Even a single parameter of a smart contract function might exhibit a virtually endless range of actual values, e.g., the 256-bit integer in Ethereum; so the goal of a fuzzing method is to pick input samples that are likely to reveal vulnerabilities. The path explosion problem occurs when the user needs to call a sequence of transactions. Even if the exact arguments are known in advance (which is not always the case), the number of possible orders of transactions and other variable scenarios "explodes" as the number of transactions in the sequence increases, which necessitates the use of special techniques, such as pruning, by the fuzzing threat mitigation methods.

Similar to symbolic execution, some fuzzing methods also utilize guidance data for facilitating test case generation. Confuzzius [66] performs a preprocessing in the form of taint analysis in order to guide the fuzzing engine. Also, in addition to identifying a problem in a smart contract, it is common for a fuzzing solution to deliver proof of a vulnerability in the form of a sample malicious transaction or a series thereof, as we see in ReGuard [105], SoliAudit [103], and EthPloit [170].

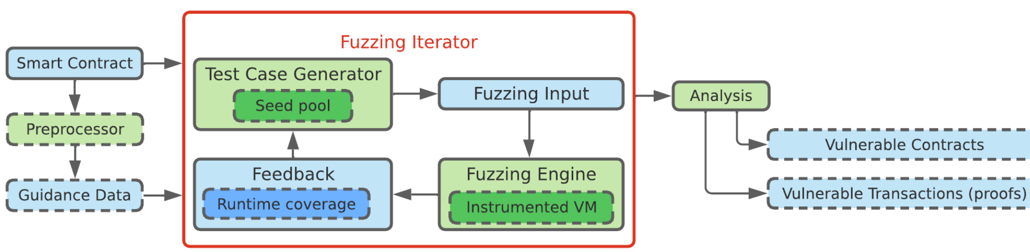


Fig. 6. Workflow of the fuzzing core method.

5.4 Formal Analysis Workflow

Formal analysis methods convert smart contracts into formal representations and use automated provers for deriving deterministic conclusions about the security properties of these smart contracts. Fig. 7 depicts the workflow of the smart contract formal analysis core method. One important component of a formal analysis solution is the fact extractor, which converts a smart contract into a formal representation, usually in a form of a domain-specific language (DSL). The formal representation is then delivered to an automated prover, such as Tamarin [117], along with some specifications representing vulnerabilities or security properties. The prover then juxtaposes the extracted facts with the provided properties to deliver a set of conclusions, which include compliance and violation statements. The output of a formal analysis solution may be supplemented with additional outputs. Specifically, some formal analysis solutions include the intermediate results in the report, e.g., extracted semantics, as seen in KEVM [83]. Also, some solutions not only prove existing theorems, but they also produce theorems based on certain specifications, such as theorems, as we can see in Lolisa [164].

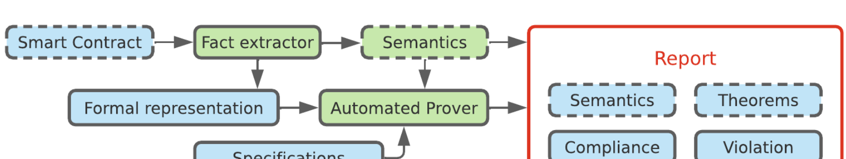


Fig. 7. Workflow of the formal analysis core method.

5.5 Machine Learning Workflow

Machine learning methods extract features from smart contracts or smart contract transactions and train models for classifying smart contracts based on the types of vulnerabilities discovered in them. Fig. 8 shows the general workflow of smart contract machine learning-based threat mitigation solutions. We discover that all the existing machine learning methods of smart contract threat mitigation use supervised models, requiring a subset of labeled smart contract samples. The workflow of a machine learning approach requires the data preprocessing (preparation) step, which includes building a "clean" (uniform) dataset, creating training and testing samples, and performing manual labeling (or using an existing one). The primary goal of the training step is to determine the parameters of a chosen model. The goal of the testing step is to verify the robustness of the model candidate. Once the model is trained and properly tested (e.g., using a K-fold method, as observed in the evaluation part of SafelyAdministrated [87]), the model can detect vulnerabilities or confirm the safety of the unlabeled contracts or smart contract transactions.

Feature extraction and model building are two major characteristics that describe machine learning threat mitigation solutions. Momeni et al. [118] deliver an ML model for detecting vulnerability patterns in smart contracts, using an abstract syntax tree (AST) and control flow graph (CFG) for feature extraction. ContractWard [156] approaches an ML-based detection of vulnerabilities in smart contracts based on bigram features. ESCORT [110] is a machine learning smart contract threat mitigation solution based on a deep neural network (DNN) with a semantic-based feature extractor. AMEVulDetector [108] builds a semantic graph from the source code and applies deep learning to building the vulnerability detection model.

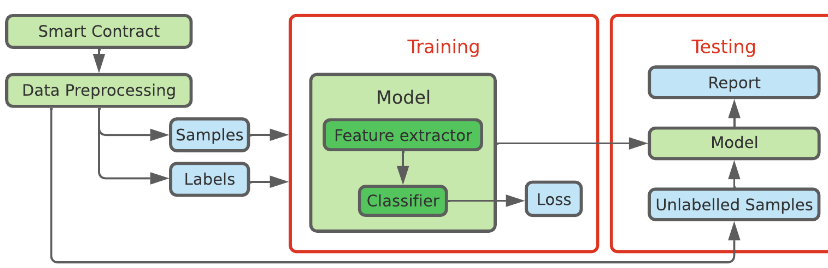


Fig. 8. Workflow of the machine learning core method.

5.6 Execution Tracing Workflow

Execution tracing methods assess the security properties of smart contracts by exploring the execution of transactions sent to a given smart contract or an externally owned account (in cases when the Ethereum platform is targeted[7]). Fig. 9 depicts the workflow of execution tracing methods. These solutions use transactions as their input. After that, the transactions are filtered to keep only the ones associated with a specific account, specific smart contract, or a concrete action (e.g., attack). Next, the filtered transactions are executed by the instrumented blockchain virtual machine (e.g., EVM). The instrumented code passively observes the execution of the given transactions and produces a special data structure called *execution traces*. Formally, an execution trace is a path in a control flow graph (CFG) of a smart contract that describes the execution of a specific

[7]Ethereum has two types of accounts: smart contract account and externally owned account (EOA). Both EOAs and smart contract accounts can be referenced by their 160-bit public addresses.

transaction (or a sequence of transactions). The execution traces are then analyzed to produce a human-readable report.

EthScope [160] is a security analysis framework that detects suspicious smart contracts in three steps: collecting related blockchain states, replaying transactions, and reporting data for manual introspective analysis. Perez et al. [128] propose an automated execution tracing framework for Ethereum for detecting both vulnerabilities and actual attacks exploiting these vulnerabilities. DEFIER [143] is a tool for the investigation of attack instances associated with Ethereum decentralized applications (DApps), which use Ethereum transaction tracing. Horus [67] is an execution tracing framework for the detection and investigation of attacks on smart contracts that use logic-based and graph-based analyses of Ethereum transactions. Another execution tracing solution is E-EVM [126] that performs emulation and visualization of smart contracts.

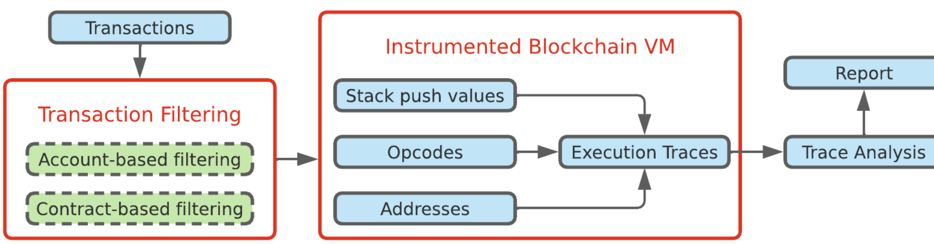


Fig. 9. Workflow of the execution tracing core method.

5.7 Code Synthesis Workflow

The code synthesis methods produce the source code or bytecode of a smart contract with or without a template. The objective of code synthesis methods is to produce a smart contract resistant to specific attacks or vulnerabilities. Fig. 10 shows the workflow of the code synthesis core method. We observe that some code synthesis solutions produce code from specifications only; others require a template to apply specifications to (e.g., ContractLarva [61]). Custom source code annotations are an example of specifications, as we can see in Cecchetti et al. [48].

Some code synthesis solutions utilize language BNF grammars or custom code libraries (e.g., SafelyAdministrated [87] and OpenZeppelin Contracts [9]) to aid the process. The result of code synthesis is a source code or a bytecode of a smart contract with specific security properties. In addition, some threat mitigation solutions utilize the code synthesis core method to patch vulnerable smart contracts on the bytecode level (e.g., SmartShield [172]).

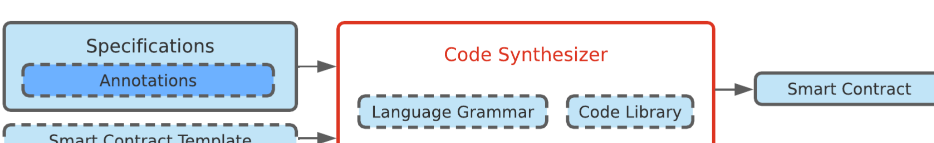


Fig. 10. Workflow of the code synthesis core method.

5.8 Transaction Interception Workflow

A blockchain network is a set of peer-to-peer (P2P) nodes. In this type of workflow, we assume that each node sustains the entire copy of the blockchain, i.e., we assume that the blockchain node is a *full node*. Furthermore, each node has a *transaction pool*, which is a queue of transactions-candidates for addition to the blockchain. Transaction interception methods are dynamic approaches that read submitted transactions from the transaction pool of the blockchain node and prevent the node from including unsafe transactions in the blockchain. Fig. 11 shows the general workflow of the transaction interception core method. Transaction interception methods employ the blockchain P2P node instrumentation, which means that there is a custom code injected into the routines responsible for transaction ordering or smart contract execution. All the transaction interception solutions surveyed in this work also produce a human-readable report of their operation, which is reasonable: deleting transactions from the pool is a deep intervention into the blockchain network protocol, so it must leave a log of the action.

Transaction interception solutions, although not numerous, exhibit a diverse spectrum of approaches. SODA [53] is a transaction-interception framework for EVM-compatible platforms that allows users to develop custom apps for dynamic defense against attacks. ÆGIS [65] is another transaction interception solution that uses a committee of voting security experts to create and approve attack patterns that steer transaction interception by instrumented nodes. Another transaction interception solution is EVM* [113], which monitors overflows and timestamp bugs.

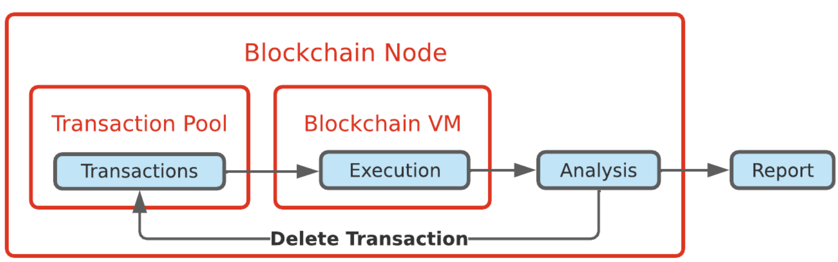


Fig. 11. Workflow of the transaction interception core method.

6 VULNERABILITY COVERAGE

In this section, we compare threat mitigation solutions from the perspective of their ability to address the known smart contract vulnerabilities. First, we select all the solutions that explicitly declare the list of vulnerabilities they cover, 38 total, and translate the information about these vulnerabilities into the model adopted by the popular SWC Registry [18]. Then we build the vulnerability map, presented in Table 3, which juxtaposes the threat mitigation methods by their ability to address the 37 known smart contract vulnerabilities. The first column of the table has the names of the threat mitigation solutions and corresponding references; if the names are not available, we use the authors instead. The next 37 columns each correspond to the numbered SWC Registry vulnerabilities. Thus, the table constitutes a compact map showing which vulnerabilities are supported (i.e., defended against), which ones are partially supported, and which ones are not supported at all for each of the 38 threat mitigation methods.

The challenge of this approach lies in the fact that different threat mitigation solutions refer to the same vulnerabilities using different names. Moreover, some solutions refer to a group of SWC vulnerabilities as a single weakness. Rodler et al. [132] declare the coverage of three vulnerabilities, which correspond to the single reentrancy vulnerability in the SWC Registry, viz., SWC-107 [13]. Some other solutions do the opposite: they break down a single SWC vulnerability into several fine-grained subgroups. For instance, the SWC-100 [12] and SWC-108 [14] vulnerabilities are often treated as a single vulnerability called the "`private` modifier", as we can see in SmartCheck [145] and in SolidityCheck [168].

Table 3 unambiguously demonstrates that different vulnerabilities exhibit unequal attention from different threat mitigation solutions. For example, 24 solutions declare defense against reentrancy (SWC-107 [13]), whereas none of the solutions declare defense against shadowing the state variables (SWC-119 [15]) and RTL-override control character (SWC-130 [17]). Remarkably, we observe that both of the vulnerabilities exhibiting close attention by the existing threat mitigation solutions as well as the ones overlooked by these solutions are often particularly challenging to pinpoint.

Lessons learned: By studying the vulnerability coverage by smart contract threat mitigation solutions, we discovered that some vulnerabilities are covered by multiple threat mitigation solutions. In contrast, many vulnerabilities are not covered by any solutions.

7 TRENDS AND PERSPECTIVES

In this section, we discuss the emerging trends in smart contract threat mitigation (§7.1, §7.2, §7.3), the overlooked types of smart contracts (§7.4), and the necessity for data-driven studies in smart contract security (§7.5). To avoid speculations and opinion-based statements, we only make inferences based on our survey data and other strong evidence.

Lessons learned: By exploring trends and perspectives associated with smart contract threat mitigation solutions, we discovered that there is a substantial room for future work despite the abundance of existing studies.

7.1 Dynamic Transaction Interception

Most smart contract threat mitigation solutions use predominantly static code-based detection approaches. However, we note that the focus of the research community is shifting in three major directions:

(1) static approaches are shifting into the dynamic paradigm;
(2) the code based methods are shifting into the transaction-based ones; and
(3) the detection methods are shifting towards verification.

Following these observations, it would be reasonable to suppose that the next generation of smart contract threat mitigation solutions will likely continue exploring the primarily overlooked area of vulnerability-agnostic dynamic transaction interception. We believe that there are two significant reasons these methods are particularly promising: they are blockchain state-aware and can address zero-day attacks.

To demonstrate the blockchain state awareness, consider the Ethereum smart contract `Foo` in Fig. 12a, which transfers cryptocurrency funds to a smart contract `Bar` (Fig. 12b). `Bar` is deployed on Ethereum Mainnet[8], but not on Ropsten testnet[9]. Moreover, `Bar` does not have any payable functions[10], and therefore it cannot accept incoming Ether. As a result, the transfer in line 6 (Fig. 12a) will fail, reverting the entire transaction — but only on Mainnet, not on Ropsten. Even if the states of all the variables of contract `Foo` on Ropsten are identical to their counterparts on Mainnet, the

[8]Ethereum Mainnet is the major production Ethereum network supporting the Ether cryptocurrency.
[9]Testnets are alternative blockchain networks utilized for development and experiments. Testnets normally execute the same protocols as production networks, but the test cryptocurrency on the testnet does not have any market value.
[10]A payable function allows to transfer (deposit) cryptocurrency to the smart contract.

Table 3. Summary of the defense tools against smart contract vulnerabilities.

Threat Mitigation Solution	*Vulnerability (SWC Registry Number)*†																																				
	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136
Oyente [111]	○	○	○	○	○	○	○	●	○	○	○	○	○	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Securify [149]	○	○	○	○	●	◐	●	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○
Mythril [120]	○	●	○	○	●	○	●	●	○	○	●	○	●	○	●	○	●	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Sereum [132]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Vandal [44]	○	○	○	○	○	●	●	●	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○
sGuard [122]	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ZEUS [92]	○	●	○	○	○	○	○	●	○	○	○	○	○	○	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
ConFuzzius [66]	○	●	○	○	●	○	●	●	○	○	●	○	●	○	●	○	●	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
VeriSmart [141]	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SmarTest [140]	○	●	○	○	○	●	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	◐	○	○	○	○	○	○	○	○	○	○	○	○	○
Maian [125]	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ECFChecker [78]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Osiris [147]	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
FSolidM [115]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ContractFuzzer [89]	○	○	○	○	●	○	○	●	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○
MadMax [74]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	●	○	○
SmartCheck [145]	●	●	●	○	●	○	○	●	●	○	○	○	○	●	○	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○
ReGuard [105]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ILF [81]	○	○	○	○	●	●	●	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
NPChecker [155]	○	○	○	○	●	○	○	●	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
EasyFlow [72]	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Vultron [154]	○	●	○	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
SoidityCheck [168]	●	●	○	○	●	○	○	○	●	○	○	○	○	●	○	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
GasFuzz [112]	●	●	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○
SolAnalyzer [22]	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	○	○	○
GasTap [24]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	●	○	○	○	○	○	●	○	○
Momeni et al. [118]	○	●	○	○	●	○	○	●	●	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Harvey [162]	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
sFuzz [123]	○	●	○	○	●	○	○	●	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
Artemis [151]	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
EthPloit [170]	○	○	○	○	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●
EthScope [160]	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
RA [57]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SeRIF [47]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Huang et al. [86]	○	●	○	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
DefectChecker [52]	○	○	○	○	●	●	○	●	○	○	○	○	○	◐	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ExGen [91]	○	●	○	○	○	○	●	◐	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
MythX [7]	○	●	○	○	●	●	●	●	○	●	●	○	●	●	○	●	●	○	○	○	●	○	○	○	●	○	○	●	●	○	○	○	○	○	●	○	○

○ — full support; ◐ — partial support; ○ — no support.

† Available at https://swcregistry.io/ and https://github.com/SmartContractSecurity/SWC-registry

```
contract Foo {
  function deposit() public payable {}
  function withdraw() public {
    address admin =
0xEc125A03C6F9E75BEB1A420e94d655B2f1352584;
    payable(admin).transfer(1000000000 wei);
    payable(msg.sender)
        .transfer(address(this).balance);
  }
}
```

(a) smart contract Foo

```
contract Bar {
  constructor() public { }
}
```

(b) smart contract Bar

Fig. 12. A pair of smart contracts demonstrating the importance of the block state.

behavior of the `withdraw()` function will be different. This example demonstrates that the state of blockchain is an important factor that determines the outcome of smart contract execution. Unlike the static ones, dynamic transaction interception methods consider the current state of the blockchain, thereby preventing situations such as those illustrated in this example.

A recent study by Zhou et al. [174] reveals that novel (zero-day) smart contract attacks constantly appear on Ethereum. This trend creates a major challenge: how to defend against attacks we do not yet know about? One way to address this problem is to utilize the prevention methods that enforce security properties instead of searching for flaws, attacks, and vulnerabilities. Unfortunately, the security properties in static prevention solutions are tightly associated with known attacks and vulnerabilities. ECFChecker (STM-009) [78] is a prevention method that verifies the "callback-free" property that ensures the safety of a smart contract from the family of reentrancy vulnerabilities. These properties, however, might not be universal enough to protect the smart contract from new vulnerabilities. One possible way to fill this gap is to verify the properties associated with expected outcomes of smart contract functions instead of vulnerability-related properties.

7.2 AI-driven Security

We identify another recent salient trend in smart contract threat mitigation solutions — AI-driven approaches involving machine learning. There are two major reasons why these approaches are capable of making a significant contribution: they allow to embrace the expressiveness of modern smart contracts, and also these approaches have been proven successful in securing other domains of computing [1, 35].

The expressiveness of smart contracts limits the capacity of static and formal analytical methods. Most modern smart contracts are Turing-complete, which allows them to implement sophisticated algorithms using high-level programming languages, such as Solidity and Rust. However, the smart contract expressiveness is a double-edged sword, as it creates a virtually infinite number of coding possibilities, which are very hard to embrace by static methods that predominantly rely upon patterns. Although machine learning methods also rely upon some patterns, recent machine learning models (e.g., deep neural network based) could explore much higher-dimensional feature spaces than static approaches.

In the past few years, we have been observing a growing trend of using AI and machine learning for security purposes, such as malware detection [134]. Although the machine learning methods for smart contract threat mitigation have not yet gained considerable popularity, the flexibility and universality of these methods will likely play an important role in smart contract defense.

7.3 Human-machine Interaction in Smart Contracts

Smart contracts are often opposed to traditional user software based on the idea of replacing human-based decisions with a deterministic algorithm. However, such a vision is overly idealistic because a human is an integral part of a smart contract lifecycle. Specifically, humans write the source code of smart contracts. Even in the case of automatically synthesized smart contracts, we still require sufficient human intervention for developing templates and specifications. Testing a smart contract also requires a human, even for unit tests, which are developed by a human developer too. The security audit of a smart contract is also impossible without human judgment despite a wide variety of auditing tools available. Finally, interaction with smart contracts is always initiated by a user, regardless of the degree of automation. However, the impact of a human on the security of smart contracts is not sufficiently studied.

The study of human-machine interaction in smart contracts is limited by exploring honeypots and revealing a potential for some social engineering attacks. Honeypots are malicious smart contracts that entrap naive attackers who try to exploit a known vulnerability in a smart contract, making honeypots a class of social engineering attacks, i.e., attacks targeting humans as the major attack vector. HoneyBadger [148] is the automated tool that identifies such honeypots. Ivanov et al. [88] expand the scope of social engineering attacks with two more categories: address manipulation and homograph. However, the two efforts mentioned above do not embrace the entire complexity of human-smart contract interaction.

One unexplored area of human-smart contract interaction is the security implication of the growing population of smart contract users who do not have a deep knowledge of the working mechanics of the blockchain and smart contracts. Another security-sensitive aspect of human-smart contract interaction is the assumption that the decentralization of blockchain implies decentralized applications (i.e., smart contracts) enabled by that blockchain. Specifically, many smart contracts implement routines (e.g, the `Ownable` parent class in OpenZeppelin Contracts [9]) that grant excessive power to specified accounts. This excessive power may be abused by the owner or stolen by the attacker [87] with potentially detrimental consequences. These two examples show the importance of studying human-smart contract interaction from the security perspective, and we envision many future studies in this area.

7.4 Non-Ethereum Contracts

As it is revealed in §4, the vast majority of the existing smart contract threat mitigation methods target the smart contracts on the Ethereum platform. However, in recent years, the world has been experiencing major growth in the popularity of non-Ethereum smart contract platforms, such as NEO [8], Hyperledger Fabric [27], EOS [3], and others. Our analysis of the evolution of smart contract threat mitigation solutions clearly shows the growing attention by the research community to the security of non-Ethereum smart contracts. One reason for such disproportional attention to Ethereum, compared to other platforms, is that Ethereum is an open-data environment with the second-largest market capitalization after Bitcoin, so it is both convenient and important to study [127]. However, these choices come at the expense of overlooking other major smart contract platforms. At the same time, our analysis shows that it is often impossible to extrapolate the lessons learned in Ethereum to the other platforms. Many of the existing vulnerabilities and other security issues are directly related to the design of the Ethereum platform or the syntax of Solidity — the most popular programming language for Ethereum smart contracts. Therefore, we expect increased attention to non-Ethereum platforms in the future development of smart contract threat mitigation research.

7.5 Large-scale Measurements

Although blockchain is an open-data environment, there are multiple facts and statistics that we are unaware of. One reason is that a large amount of blockchain-related data, such as failed transactions and ERC20 token prices, is stored outside of the blockchain. Moreover, the growing popularity of Decentralized Finance (DeFi) further intensified the exchange of off-chain data [5, 6]. As a result, we have seen the growing amounts of on-chain and off-chain data that have not been analyzed from a security perspective.

Yet, the existing security-related measurement studies [65, 128, 148, 174] of smart contracts do not give answers to all the important questions. Specifically, we identify two areas important for the security of smart contracts in which there is no systematic data:

(1) the measurement and flow of the market value of non-cryptocurrency blockchain assets (e.g., ERC20 tokens);
(2) study of the purchases and sales of cryptocurrency and tokens by the crypto exchanges, mining rewards, and crypto money laundering.

Such data would be very helpful for applying weights to attacks and vulnerabilities based on the actual value flow of the smart contract assets.

8 CONCLUSION

We surveyed the full spectrum of smart contract threat mitigation solutions in this work. We presented a general taxonomy for the classification of such solutions, which applies to today's methods and is suitable for future methods, even if new paradigms, blockchain platforms, or vulnerabilities appear. Using this taxonomy, we classified 133 existing smart contract threat mitigation solutions. We identified eight distinct core defense methods employed by the existing solutions and developed synthesized workflows of these core methods. We studied the ability of the existing smart contract threat mitigation solutions to address the known vulnerabilities. We conducted an evidence-based evolutionary study of smart contract threat mitigation solutions to outline trends and perspectives. To further benefit the community of smart contract security researchers, users, and developers, we deployed an open-source, regularly updated online registry for smart contract threat mitigation at https://stmregistry.io/.

REFERENCES

[1] 2022. Artificial intelligence (AI) for cybersecurity. https://www.ibm.com/security/artificial-intelligence. Accessed: 2022-03-07.
[2] 2022. Dedadub Contract Library. https://dedaub.com/contract-library. Accessed: 2022-02-28.
[3] 2022. EOS.IO Technical White Paper v2. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md. Accessed: 2022-03-07.
[4] 2022. Etherscan Token Tracker. https://etherscan.io/tokens. Accessed: 2022-03-05.
[5] 2022. Go implementation of MEV-Auction for Ethereum. https://github.com/flashbots/mev-geth.
[6] 2022. Miner Extractable Value (MEV). https://ethereum.org/en/developers/docs/mev/.
[7] 2022. MythX. https://mythx.io/. Accessed: 2022-02-26.
[8] 2022. Neo White Paper. https://docs.neo.org/v2/docs/en-us/basic/whitepaper.html. Accessed: 2022-03-07.
[9] 2022. OpenZeppelin Contracts. https://openzeppelin.com/contracts/. Accessed: 2022-02-28.
[10] 2022. Polygon. https://polygon.technology/.
[11] 2022. RSK Whitepaper. https://www.rsk.co/Whitepapers/RSK_White_Paper-ORIGINAL.pdf.
[12] 2022. SWC-100: Function Default Visibility. https://swcregistry.io/docs/SWC-100. Accessed: 2022-03-21.
[13] 2022. SWC-107: Reentrancy. https://swcregistry.io/docs/SWC-107. Accessed: 2022-03-21.
[14] 2022. SWC-108: State Variable Default Visibility. https://swcregistry.io/docs/SWC-108. Accessed: 2022-03-21.

[15] 2022. SWC-119: Shadowing State Variables. https://swcregistry.io/docs/SWC-119. Accessed: 2022-04-17.
[16] 2022. SWC-123: Requirement Violation. https://swcregistry.io/docs/SWC-123. Accessed: 2022-03-10.
[17] 2022. SWC-130: Right-To-Left-Override control character (U+202E). https://swcregistry.io/docs/SWC-130. Accessed: 2022-04-17.
[18] 2022. SWC Registry. https://swcregistry.io/. Accessed: 2022-03-15.
[19] 2022. Z3Prover/z3. https://github.com/Z3Prover/z3.
[20] Tesnim Abdellatif and Kei-Léo Brousmiche. 2018. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.
[21] Wolfgang Ahrendt, Richard Bubel, Joshua Ellul, Gordon J Pace, Raúl Pardo, Vincent Rebiscoul, and Gerardo Schneider. 2019. Verification of smart contract business logic. In *International Conference on Fundamentals of Software Engineering*. Springer, 228–243.
[22] Sefa Akca, Ajitha Rajan, and Chao Peng. 2019. SolAnalyser: A Framework for Analysing and Testing Smart Contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 482–489.
[23] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. 2019. SAFEVM: a safety verifier for Ethereum smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 386–389.
[24] Elvira Albert, Pablo Gordillo, Albert Rubio, and Ilya Sergey. 2019. Running on fumes. In *International Conference on Verification and Evaluation of Computer and Communication Systems*. Springer, 63–78.
[25] Sarra Alqahtani, Xinchi He, Rose Gamble, and Papa Mauricio. 2020. Formal verification of functional requirements for smart contract compositions in supply chain management systems. In *Proceedings of the 53rd Hawaii International Conference on System Sciences*.
[26] 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*. 66–77.
[27] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
[28] Pedro Antonino and AW Roscoe. 2021. Solidifier: bounded model checking Solidity using lazy contract deployment and precise memory modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 1788–1797.
[29] Mauro Argañaraz, Mario Berón, Maria João Pereira, and Pedro Henriques. 2020. Detection of vulnerabilities in smart contracts specifications in ethereum platforms. In *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*, Vol. 83. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 1–16.
[30] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*. Springer, 164–186.
[31] Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. 2019. Developing secure Bitcoin contracts with BitML. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1124–1128.
[32] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. 2018. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th international conference on software and computer applications*. 322–326.
[33] Massimo Bartoletti and Roberto Zunino. 2019. Verifying liquidity of Bitcoin contracts. In *8th International Conference on Principles of Security and Trust, POST 2019 Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019*, Vol. 11426. Springer, 222–247.
[34] Bernhard Beckert, Mihai Herda, Michael Kirsten, and Jonas Schiffl. 2018. Formal specification and verification of hyperledger fabric chaincode. In *Proc. Int. Conf. Formal Eng. Methods*. 44–48.
[35] Elisa Bertino, Murat Kantarcioglu, Cuneyt Gurcan Akcora, Sagar Samtani, Sudip Mittal, and Maanak Gupta. 2021. AI for Security and Security for AI. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*. 333–334.
[36] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*. 91–96.
[37] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. 2015. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*. Springer, 142–161.
[38] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure derivative contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 453–467.

[39] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2021. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. *arXiv preprint arXiv:2104.08638* (2021).

[40] Santiago Bragagnolo, Henrique Rocha, Marcus Denker, and Stéphane Ducasse. 2018. SmartInspect: solidity smart contract inspector. In *2018 International workshop on blockchain oriented software engineering (IWBOSE)*. IEEE, 9–18.

[41] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2017. An in-depth look at the parity multisig bug. *Hacking, Distributed, July* (2017).

[42] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. 2018. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1335–1352.

[43] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 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.

[44] 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).

[45] R Browne. 2017. Accidental bug may have frozen 280 million worth of digital coin ether in a cryptocurrency wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[46] Ramiro Camino, Christof Ferreira Torres, Mathis Baden, and Radu State. 2020. A data science approach for detecting honeypots in ethereum. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.

[47] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. [n. d.]. Compositional Security for Reentrant Applications. *contract* 12, 13 ([n. d.]), 14.

[48] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C Myers. 2020. Securing smart contracts with information flow. In *International Symposium on Foundations and Applications of Blockchain*.

[49] Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, Yan Cai, and Zijiang Yang. 2019. scompile: Critical path identification and analysis for smart contracts. In *International Conference on Formal Engineering Methods*. Springer, 286–304.

[50] 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.

[51] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining smart contract defects on ethereum. *IEEE Transactions on Software Engineering* (2020).

[52] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* (2021).

[53] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A generic online detection framework for smart contracts. In *27th Ann. Network and Distributed Systems Security Symp.* The Internet Society.

[54] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446.

[55] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1503–1520.

[56] Weili Chen, Zibin Zheng, Jiahui Cui, Edith Ngai, Peilin Zheng, and Yuren Zhou. 2018. Detecting ponzi schemes on ethereum: Towards healthier blockchain technology. In *Proceedings of the 2018 world wide web conference*. 1409–1418.

[57] Yuchiro Chinen, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. 2020. RA: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis. In *2020 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 327–336.

[58] 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)*. IEEE, 69–78.

[59] Mengjie Ding, Peiru Li, Shanshan Li, and He Zhang. 2021. HFContractFuzzer: Fuzzing Hyperledger Fabric Smart Contracts for Vulnerability Detection. In *Evaluation and Assessment in Software Engineering*. 321–328.

[60] Wang Duo, Huang Xin, and Ma Xiaofeng. 2020. Formal analysis of smart contract based on colored petri nets. *IEEE Intelligent Systems* 35, 3 (2020), 19–30.

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

[62] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[63] Yu Feng, Emina Torlak, and Rastislav Bodik. 2019. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067* (2019).

[64] Yu Feng, Emina Torlak, and Rastislav Bodik. 2020. Summary-based symbolic evaluation for smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1141–1152.
[65] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. 2020. Ægis: Shielding vulnerable smart contracts against attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 584–597.
[66] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, et al. 2021. CONFUZZIUS: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. (2021).
[67] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, et al. 2021. The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts. In *International Conference on Financial Cryptography and Data Security, Grenada 1-5 March 2021*.
[68] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *29th USENIX Security Symposium (USENIX Security 20)*. 2757–2774.
[69] Ernesto Frontera. [n. d.]. A History of The DAO Hack. https://coinmarketcap.com/alexandria/article/a-history-of-the-dao-hack. Accessed: 2022-02-15.
[70] Ying Fu, Meng Ren, Fuchen Ma, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2019. Evmfuzz: Differential fuzz testing of ethereum virtual machine. *arXiv preprint arXiv:1903.08483* (2019).
[71] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1110–1114.
[72] Jianbo Gao, Han Liu, Chao Liu, Qingshan Li, Zhi Guan, and Zhong Chen. 2019. Easyflow: Keep ethereum away from overflow. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 23–26.
[73] Dan Goodin. 2021. Really stupid “smart contract” bug let hackers steal $31 million in digital coin. https://arstechnica.com/information-technology/2021/12/hackers-drain-31-million-from-cryptocurrency-service-monox-finance/.
[74] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.
[75] 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.
[76] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Ethertrust: Sound static analysis of ethereum bytecode. *Technische Universität Wien, Tech. Rep* (2018).
[77] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *International Conference on Principles of Security and Trust*. Springer, 243–269.
[78] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28.
[79] Ákos Hajdu and Dejan Jovanović. 2019. solc-verify: A modular verifier for Solidity smart contracts. *arXiv preprint arXiv:1907.04262* (2019).
[80] Ákos Hajdu, Dejan Jovanović, and Gabriela Ciocarlie. 2020. Formal Specification and Verification of Solidity Contracts with Events (Short Paper). In *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.
[81] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 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.
[82] 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.
[83] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.
[84] 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.
[85] Xinwen Hu, Yi Zhuang, Shang-Wei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. 2021. A Security Type Verifier for Smart Contracts. *Computers & Security* (2021), 102343.

[86] Jianjun Huang, Songming Han, Wei You, Wenchang Shi, Bin Liang, Jingzheng Wu, and Yanjun Wu. 2021. Hunting Vulnerable Smart Contracts via Graph Embedding Based Bytecode Matching. *IEEE Transactions on Information Forensics and Security* 16 (2021), 2144–2156.

[87] Nikolay Ivanov, Hanqing Guo, and Qiben Yan. 2021. Rectifying Administrated ERC20 Tokens. In *International Conference on Information and Communications Security*. Springer, 22–37.

[88] Nikolay Ivanov, Jianzhi Lou, Ting Chen, Jin Li, and Qiben Yan. 2021. Targeting the Weakest Link: Social Engineering Attacks in Ethereum Smart Contracts. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 787–801.

[89] Bo Jiang, Ye Liu, and WK Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.

[90] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic understanding of smart contracts: executable operational semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1695–1712.

[91] Ling Jin, Yinzhi Cao, Yan Chen, Di Zhang, and Simone Campanoni. 2022. EXGEN: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2022).

[92] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts.. In *Ndss*. 1–12.

[93] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[94] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*. 363–373.

[95] Jaturong Kongmanee, Phongphun Kijsanayothin, and Rattikorn Hewett. 2019. Securing smart contracts in blockchain. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 69–76.

[96] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1317–1333.

[97] Ton Chanh Le, Lei Xu, Lin Chen, and Weidong Shi. 2018. Proving conditional termination for smart contracts. In *Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts*. 57–59.

[98] 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.

[99] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. 2020. A survey on the security of blockchain systems. *Future Generation Computer Systems* 107 (2020), 841–853.

[100] Xiaoyu Li, Cheng Su, Yan Xiong, Wenchao Huang, and Wansen Wang. 2019. Formal verification of BNB smart contract. In *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 74–78.

[101] Yue Li. 2019. Finding concurrency exploits on smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 144–146.

[102] Yue Li, Han Liu, Zhiqiang Yang, Qian Ren, Lei Wang, and Bangdao Chen. 2020. SafePay on Ethereum: A Framework For Detecting Unfair Payments in Smart Contracts. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1219–1222.

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

[104] Shlomi Linoy, Suprio Ray, and Natalia Stakhanova. [n. d.]. EtherProv: provenance-aware detection, analysis, and mitigation of Ethereum smart contract security issues. ([n. d.]).

[105] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[106] Han Liu, Chao Liu, Wenqi Zhao, Yu Jiang, and Jiaguang Sun. 2018. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 814–819.

[107] Ye Liu, Yi Li, Shang-Wei Lin, and Qiang Yan. 2020. ModCon: A model-based testing platform for smart contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1601–1605.

[108] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. *arXiv preprint arXiv:2106.09282* (2021).

[109] Ning Lu, Bin Wang, Yongxin Zhang, Wenbo Shi, and Christian Esposito. 2019. NeuCheck: A more practical Ethereum smart contract security analysis tool. *Software: Practice and Experience* (2019).

[110] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning. *arXiv preprint arXiv:2103.12607* (2021).
[111] 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*. 254–269.
[112] Fuchen Ma, Ying Fu, Meng Ren, Wanting Sun, Zhe Liu, Yu Jiang, Jun Sun, and Jiaguang Sun. 2019. Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *arXiv preprint arXiv:1910.02945* (2019).
[113] Fuchen Ma, Ying Fu, Meng Ren, Mingzhe Wang, Yu Jiang, Kaixiang Zhang, Huizhong Li, and Xiang Shi. 2019. EVM*: from offline detection to online reinforcement for ethereum virtual machine. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 554–558.
[114] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti EJ Hyvärinen, and Natasha Sharygina. 2020. Accurate smart contract verification through direct modelling. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 178–194.
[115] Anastasia Mavridou and Aron Laszka. 2018. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*. Springer, 523–540.
[116] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-design smart contracts for Ethereum. In *International Conference on Financial Cryptography and Data Security*. Springer, 446–465.
[117] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*. Springer, 696–701.
[118] Pouyan Momeni, Yu Wang, and Reza Samavi. 2019. Machine learning model for smart contracts security analysis. In *2019 17th International Conference on Privacy, Security and Trust (PST)*. IEEE, 1–6.
[119] 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)*. IEEE, 1186–1189.
[120] Bernhard Mueller. 2018. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference (HITBSecConf)*, Vol. 54.
[121] Zeinab Nehai, Pierre-Yves Piriou, and Frédéric Daumas. 2018. Model-Checking of Smart Contracts. In *IEEE International Conference on Blockchain*.
[122] Tai D Nguyen, Long H Pham, and Jun Sun. [n. d.]. sGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. *arXiv preprint arXiv:2101.01917* ([n. d.]).
[123] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.
[124] Yuandong NI, Chao ZHANG, and Tingting YIN. 2020. A Survey of Smart Contract Vulnerability Research. *Journal of Cyber Security* 5, 3 (2020), 78–99.
[125] 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.
[126] Robert Norvill, Beltran Borja Fiz Pontiveros, Radu State, and Andrea Cullen. 2018. Visual emulation for Ethereum's virtual machine. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–4.
[127] Daniel Perez and Benjamin Livshits. 2019. Smart contract vulnerabilities: Does anyone care? *arXiv preprint arXiv:1902.06710* (2019), 1–15.
[128] Daniel Perez and Ben Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium (USENIX Security 21)*.
[129] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1661–1677.
[130] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. 2019. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605* (2019).
[131] Lijin Quan, Lei Wu, and Haoyu Wang. 2019. EVulHunter: detecting fake transfer vulnerabilities for EOSIO's smart contracts at Webassembly-level. *arXiv preprint arXiv:1906.10362* (2019).
[132] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).
[133] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2021. EVMPatch: timely and automated patching of ethereum smart contracts. In *30th USENIX Security Symposium (USENIX Security 21)*.
[134] Justin Sahs and Latifur Khan. 2012. A machine learning approach to android malware detection. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 141–147.

[135] Noama Fatima Samreen and Manar H. Alalfi. 2021. A Survey of Security Vulnerabilities in Ethereum Smart Contracts. *CoRR* abs/2105.06974 (2021). arXiv:2105.06974 https://arxiv.org/abs/2105.06974

[136] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and provably sound static analysis of Ethereum smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 621–640.

[137] Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in flint. In *Conference companion of the 2nd international conference on art, science, and engineering of programming*. 218–219.

[138] MacKenzie Sigalos. 2021. Bug puts $162 million up for grabs, says founder of DeFi platform Compound. https://www.cnbc.com/2021/10/03/162-million-up-for-grabs-after-bug-in-defi-protocol-compound-.html.

[139] Amritraj Singh, Reza M Parizi, Qi Zhang, Kim-Kwang Raymond Choo, and Ali Dehghantanha. 2020. Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities. *Computers & Security* 88 (2020), 101654.

[140] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SMARTEST: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*.

[141] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. 2020. VeriSmart: A highly precise safety verifier for Ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1678–1694.

[142] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *IEEE S&P*.

[143] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil Under the Sun: Understanding and Discovering Attacks on Ethereum Decentralized Applications. In *30th USENIX Security Symposium (USENIX Security 21)*.

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

[145] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[146] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–38.

[147] 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.

[148] Christof Ferreira Torres, Mathis Steichen, et al. 2019. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium (USENIX Security 19)*. 1591–1607.

[149] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[150] Anna Vacca, Andrea Di Sorbo, Corrado A. Visaggio, and Gerardo Canfora. 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software* 174 (2021), 110891. https://doi.org/10.1016/j.jss.2020.110891

[151] Anqi Wang, Hao Wang, Bo Jiang, and Wing Kwong Chan. 2020. Artemis: An improved smart contract verification tool for vulnerability detection. In *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 173–181.

[152] Bin Wang, Han Liu, Chao Liu, Zhiqiang Yang, Qian Ren, Huixuan Zheng, and Hong Lei. 2021. BLOCKEYE: Hunting For DeFi Attacks on Blockchain. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 17–20.

[153] Dong Wang, Bo Jiang, and WK Chan. 2020. WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection. *arXiv preprint arXiv:2007.15510* (2020).

[154] Haijun Wang, Yi Li, Shang-Wei Lin, Lei Ma, and Yang Liu. 2019. VULTRON: catching vulnerable smart contracts once and for all. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 1–4.

[155] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

[156] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2020. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering* (2020).

[157] Yajing Wang, Jingsha He, Nafei Zhu, Yuzi Yi, Qingqing Zhang, Hongyu Song, and Ruixin Xue. 2021. Security enhancement technologies for smart contracts in the blockchain: A survey. *Transactions*

on Emerging Telecommunications Technologies 32, 12 (2021), e4341. https://doi.org/10.1002/ett.4341 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.4341

[158] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. 2019. Formal verification of workflow policies for smart contracts in azure blockchain. In *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 87–106.

[159] Zeli Wang, Hai Jin, Weiqi Dai, Kim-Kwang Raymond Choo, and Deqing Zou. 2021. Ethereum smart contract security research: survey and future research opportunities. *Frontiers of Computer Science* 15, 2 (2021), 1–18.

[160] Lei Wu, Siwei Wu, Yajin Zhou, Runhuai Li, Zhi Wang, Xiapu Luo, Cong Wang, and Kui Ren. [n. d.]. Ethscope: A transaction-centric security analytics framework to detect malicious smart contracts on ethereum. ([n. d.]).

[161] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. 2021. DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications. *arXiv preprint arXiv:2104.15068* (2021).

[162] 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.

[163] Wentian Yan, Jianbo Gao, Zhenhao Wu, Yue Li, Zhi Guan, Qingshan Li, and Zhong Chen. 2020. Eshield: protect smart contracts against reverse engineering. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 553–556.

[164] Zheng Yang and Hang Lei. 2018. Lolisa: Formal syntax and semantics for a subset of the solidity programming language. *arXiv e-prints* (2018), arXiv–1803.

[165] Zhiqiang Yang, Han Liu, Yue Li, Huixuan Zheng, Lei Wang, and Bangdao Chen. 2020. Seraph: enabling cross-platform security analysis for evm and wasm smart contracts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 21–24.

[166] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 274–275.

[167] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *29th USENIX Security Symposium (USENIX Security 20)*. 2775–2792.

[168] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2019. SolidityCheck: Quickly detecting smart contract problems through regular expressions. *arXiv preprint arXiv:1911.09425* (2019).

[169] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. 2020. A framework and dataset for bugs in ethereum smart contracts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 139–150.

[170] Qingzhao Zhang, Yizhuo Wang, Juanru Li, and Siqi Ma. 2020. Ethploit: From fuzzing to efficient exploit generation against smart contracts. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 116–126.

[171] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven Stewart, and Vijay Ganesh. 2019. Mpro: Combining static and symbolic analysis for scalable testing of smart contract. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 456–462.

[172] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 23–34.

[173] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5.

[174] Shunfan Zhou, Malte Möser, Zhemin Yang, Ben Adida, Thorsten Holz, Jie Xiang, Steven Goldfeder, Yinzhi Cao, Martin Plattner, Xiaojun Qin, et al. 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.

[175] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum's opaque smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1371–1385.

[176] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* (2019).