# A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses

HUASHAN CHEN, The University of Texas at San Antonio, USA
MARCUS PENDLETON, U.S. Air Force Research Laboratory and 90 COS/CYD
LAURENT NJILLA, U.S. Air Force Research Laboratory
SHOUHUAI XU, The University of Texas at San Antonio, USA

Blockchain technology is believed by many to be a game changer in many application domains. While the first generation of blockchain technology (i.e., Blockchain 1.0) is almost exclusively used for cryptocurrency, the second generation (i.e., Blockchain 2.0), as represented by Ethereum, is an open and decentralized platform enabling a new paradigm of computing—Decentralized Applications (DApps) running on top of blockchains. The rich applications and semantics of DApps inevitably introduce many security vulnerabilities, which have no counterparts in pure cryptocurrency systems like Bitcoin. Since Ethereum is a new, yet complex, system, it is imperative to have a systematic and comprehensive understanding on its security from a holistic perspective, which was previously unavailable in the literature. To the best of our knowledge, the present survey, which can also be used as a tutorial, fills this void. We systematize three aspects of Ethereum systems security: vulnerabilities, attacks, and defenses. We draw insights into vulnerability root causes, attack consequences, and defense capabilities, which shed light on future research directions.

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

Additional Key Words and Phrases: Blockchain, Ethereum, smart contract, security

**ACM Reference format:**
Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. 2020. A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Comput. Surv.* 53, 3, Article 67 (June 2020), 43 pages.
https://doi.org/10.1145/3391195

## 1 INTRODUCTION

The notion of *blockchain* was implicitly introduced in 2008 as the key underlying technique of the cryptocurrency known as Bitcoin [148], which uses a *transaction-centered* model known as *unspent transaction outputs* (UTXO). In this model, a blockchain is a distributed and public ledger, which records the payment transactions between parties over a peer-to-peer (P2P) network. Unlike traditional digital cash systems [80], in which there is a trusted third party (e.g., bank), there is no trusted



This research was supported in part by US AFRL Grant No. FA8750-19-1-0019, ARO Grant No. W911NF-17-1-0566, NSF Grant No. 1814825, and NSF CREST Grant No. 1736209. The views and opinions of the authors do not reflect those of the US DoD, AFRL, ARO, or NSF.
Authors' addresses: H. Chen and S. Xu (corresponding author), One UTSA Circle, University of Texas at San Antonio, San Antonio, TX 78249; emails: {huashan.chen, shouhuai.xu}@utsa.edu; M. Pendleton, 250 Hall Blvd., Suite 359, San Antonio, Texas 78243-7078; email: marcus.pendleton.2@us.af.mil; L. Njilla, 26 Electronics Parkway, Rome, New York 13441; email: laurent.njilla@us.af.mil.

0360-0300/2020/06-ART67 $15.00
https://doi.org/10.1145/3391195

third party in a blockchain system in general, and in Bitcoin in particular. Bitcoin is often referred to as Blockchain 1.0, because it only offers payment services. The innovation of the Bitcoin system is its consensus protocol, which allows mutually distrusting nodes in a P2P network to eventually reach a consensus on the outcome after executing payment transactions. Unlike traditional consensus protocols [100], the participants are from an open network and are incentivized by the payment of Bitcoins (or BTCs), which are "mined" through a clever cryptographic hash function known as Proof-of-Work (PoW), an idea originally proposed as an anti-spam technique [95].

Perhaps inspired by the success of Bitcoin as well as the need to support semantically richer (than just payment) applications, the notion of *smart contracts* has been introduced to represent autonomous programs, leading to a new paradigm of Decentralized Applications (DApps) that run on top of blockchains and consist of many interacting smart contracts. The Ethereum system was launched in 2015 to support smart contracts, while offering its inherent cryptocurrency known as Ether [184] and using an *account-centered* model (rather than the UTXO model mentioned above). Ethereum has become the de facto standard platform for DApps. At the moment of writing, the market value of Ethereum is over US$31B with approximately one million smart contracts executing on top of the Ethereum blockchain [56]. The success of Ethereum ushers in Blockchain 2.0, which has many applications in Artificial Intelligence [165], Internet of Things [62, 122, 172], and digital assets [113, 114].

However, supporting rich applications makes Ethereum have a large vulnerability surface, as evidenced by the many high-profile attacks. One example is the DAO attack [5] in 2016, where an attacker exploited the reentrancy vulnerability (which will be detailed later) to steal about US$60M. In July 2017, a vulnerability in the Parity wallet contract caused the loss of US$31M [19]. In April 2018, the MyEtherWallet wallet fell victim to a BGP and DNS hijacking attack, enabling the hacker to steal US$17M [26]. These attacks highlight that our capabilities in securing the Ethereum system are limited. This should not be taken as a surprise, because Ethereum is a new programming paradigm with DApps running on top of blockchains with many autonomous contracts.

The motivation of the present survey is threefold, to serve *researchers*, *practitioners*, and *students*. From the standpoint of a *researcher* who wants to investigate Ethereum security, there is a need for a source of systematized treatment on the problems. Despite the fact that there have been some surveys, they did not offer a systematic and comprehensive view on Ethereum vulnerabilities, attacks, and defenses as we do. While referring to the related prior work in Section 1.2 for details, we mention the following: There is neither systematic understanding of the Ethereum vulnerabilities that have been discovered, nor systematic understanding of their root causes; this may explain why there are still a number of vulnerabilities that are completely open. From the standpoint of a *practitioner*, there is a need for a source of best practices and guiding principles. Industry has conducted due diligence in summarizing many best practices [49], which, however, may overwhelm practitioners. Therefore, it might be more useful to have a small number of guiding principles that are easier to adopt in practice. From the standpoint of a *student* who wants to learn about Ethereum security, there is a need for a succinct yet comprehensive and systematic source that also offer references to materials of greater details.

### 1.1 Our Contributions

We provide a systematic and comprehensive survey on Ethereum systems security, where "systematic" means that *vulnerabilities*, *attacks*, and *defenses* as well as the relationships between them are accommodated and "comprehensive" means that it covers both the layer Ethereum platform and the environment in which Ethereum operates. In terms of vulnerabilities, we enumerate 40 types of Ethereum vulnerabilities at layers of the Ethereum architecture and systematize their root causes. Some of our insights are highlighted as follows. (i) Authentication and authorization

failures in Ethereum smart contracts are a major problem, which calls for standardized, stronger identity management solutions and access control mechanisms. (ii) External dependence makes it hard, if not impossible, to assure the security of Ethereum smart contracts, highlighting the importance of enforcing security controls on external calls. (iii) Incompetent Ethereum smart contract programming introduces many new kinds of vulnerabilities, highlighting the importance of standardizing domain-specific best practices for the new programming paradigm. (iv) The unreliability of Solidity makes Ethereum smart contracts vulnerability-prone, highlighting the importance of reliable programming languages. (v) Arbitrary choices of parameters in Ethereum specification and implementation cause many vulnerabilities in Ethereum, highlighting the importance of executing the "open design" principle. (vi) Vulnerabilities in Ethereum are harder to cope with than vulnerabilities in other systems, hinting that Ethereum blockchain is inherently more complex.

In terms of attacks, we systematize 29 attacks against Ethereum according to the layers of the Ethereum architecture. We relate these attacks to the vulnerabilities and systematize their consequences. Some of our insights are highlighted as follows. (i) Ethereum blockchain has two security barriers: permissionless (allowing attackers to exploit vulnerabilities at will) and immutability (disabling the vulnerability-patching mechanism widely used in cyber defense). (ii) While application-layer attacks have caused huge financial losses, the damage did not spread to the host computers because of the EVM isolation. (iii) Code-reuse in smart contracts can impose a higher risk than its counterpart in traditional systems, highlighting the importance of security auditing on widely-reused smart contracts and libraries.

We systematize 51 defenses into two classes: proactive defenses, which aim to prevent attacks; and reactive defenses, which aim to cope with hidden vulnerabilities. We present a deeper analysis according to defenses' capabilities. Some of our insights are highlighted as follows: (i) Proactive defenses can defend against attacks exploiting many vulnerabilities; reactive defenses can defend against attacks exploiting a few vulnerabilities. (ii) There is no single silver-bullet defense at the application layer, let alone the entire Ethereum system, highlighting the necessity of defense-in-depth. (iii) A better programming language can not only make smart contracts more secure but also achieve a higher degree of fault-tolerance. (iv) There is a large discrepancy between the efforts that have been invested to defend against attacks that exploit different vulnerabilities.

Although the present article focuses on the Ethereum system, the aforementioned vulnerability-attack-defense framework and insights can be adopted or adapted to accommodate other blockchain systems. We discuss Ethereum's planned PoS upgrade toward Ethereum 2.0 (called Casper) and its plan on coping with attacks against PoS protocols. To get a glance at the vulnerability perspective of popular blockchain systems, we apply our taxonomy of Ethereum vulnerabilities to Hyperledger and EOS blockchains and find that blockchains using different programming languages and architectures have very different vulnerabilities. We discuss three important future research directions in securing Ethereum systems, including: (i) eliminating known Ethereum vulnerabilities; (ii) developing Ethereum test tools and environments; (iii) formalizing, analyzing and quantifying Ethereum security. Under each direction, we list a set of open problems and suggest possible technical approaches to tackling them. The preceding discussion justifies how the present article can serve the needs of students and researchers. For practitioners, we systematize the best practices into a small number of principles that may be easier to adopt in practice.

### 1.2 Related Work

Table 1 highlights the relationship between the present work and related prior surveys, which accommodate some of vulnerabilities, attacks, defenses, and building-blocks. The most closely related survey is Atzei et al. [65], which discussed 12 types of vulnerabilities, 9 attacks, and

Table 1. Comparison between Surveys Related to Blockchain Security

| Study | Vulnerabilities | | | Attacks | | | Defenses | | | Building-blocks | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Bitcoin | Ethereum | Insights | Bitcoin | Ethereum | Insights | Bitcoin | Ethereum | Insights | Cryptography | Consensus |
| [65] | | 12 | | | 9 | | | 3 | | | |
| This work | | 40 | ✓ | | 29 | ✓ | | 51 | ✓ | | |
| [21] | 20 | | | 6 | | | 5 | | | | |
| [200] | 11 | | | 11 | | | 30 | | | | |
| [164] | | | | 22 | | ✓ | 33 | | | | ✓ |
| [112] | | | | | | | | 10 | ✓ | | |
| [94] | | | | | | | | 27 | ✓ | | |
| [196] | | | | | | | | | | ✓ | |
| [67, 78, 179, 186] | | | | | | | | | | | ✓ |

3 defenses in the context of Ethereum smart contracts. In contrast, we present a much more systematic treatment by accommodating 40 types of vulnerabilities, 29 attacks, and 51 defenses. From a methodological standpoint, we further discuss the root causes of vulnerabilities (e.g., the root cause of the *unchecked call return value* vulnerability is the *inconsistent exception handling* of Solidity). Moreover, we draw insights from the perspectives of vulnerability, attack, and defense.

There are a number of surveys on blockchain security from different perspectives than ours. First, Li et al. [21] reviewed blockchain security through 20 types of vulnerabilities, 6 attacks, and 5 defenses, without making distinction between Bitcoin and Ethereum. Similarly, Zhu et al. [200] reviewed 11 smart contract vulnerabilities, 11 attacks against blockchain data, and 30 defenses. On the contrary, we focus on Ethereum blockchain by accommodating 40 types of vulnerabilities, 29 attacks, and 51 defenses. Second, Saad et al. [164] explored blockchains' attack surface in terms of cryptographic constructions, distributed system architecture, and applications; they cover 22 attacks and 33 defenses, but not vulnerabilities. Their survey is orthogonal to ours, because (i) we focus on Ethereum, rather than multiple implementations of blockchains, (ii) we discuss vulnerabilities and their root causes as well as the attacks exploiting them, rather than attack surface, and (iii) we provide insights into, and contrast, the vulnerabilities at different layers of the Ethereum architecture. Third, Harz et al. [112] discussed 10 smart contract verification tools. Similarly, Angelo et al. [94] discussed 27 tools for analyzing Ethereum smart contracts. On the contrary, we focus on Ethereum defenses, rather than purely on smart contracts. Fourth, Zhang et al. [196] presented a comprehensive review on Bitcoin-like transactions and the underlying (cryptographic) mechanisms. Their review is geared toward the abstract blockchain model for Bitcoin-like transactions; in contrast, we focus on the Ethereum ecosystem, including the design and implementation of the blockchain platform and DApps. There are surveys on Bitcoin and cryptocurrencies [74, 85, 177], which use a transaction-centered model known as *unspent transaction outputs* (UTXO). In contrast, we focus on Ethereum, which uses an account-centered model. Fifth, there are surveys on blockchain consensus protocols [67, 78, 179, 186] and blockchain-based applications in IoT security [122]. In contrast, we focus on the Ethereum system.

### 1.3 Paper Outline

Section 2 briefly reviews the Ethereum system and discusses the survey methodology. Section 3 presents 40 Ethereum vulnerabilities and analyzes their root causes. Section 4 presents 29 attacks against Ethereum and analyzes their consequences. Section 5 presents 51 defenses and analyzes their capabilities and investments. Section 6 discusses ongoing Ethereum development and comparison with other blockchains. Section 7 discusses future research directions. Section 8 concludes the article. The Appendix presents vulnerabilities in, and attacks against, the Ethereum environment.

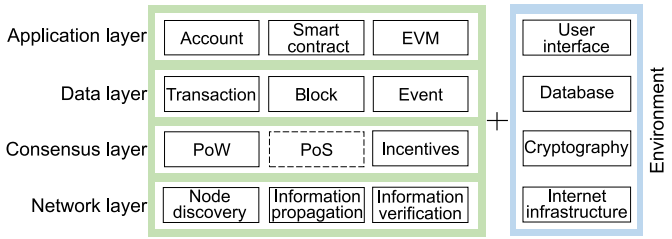


Fig. 1. Architecture of the Ethereum blockchain and its environment in which the Ethereum blockchain runs. The *environment* serves the four layers of the Ethereum architecture via a web user interface to interact with applications, databases for storing blockchain data, cryptographic mechanisms for supporting the consensus protocols, and Internet service for the network layer.

## 2 ETHEREUM REVIEW AND SURVEY METHODOLOGY

### 2.1 A Brief Review of the Ethereum System

Figure 1 highlights a four-layer architecture of Ethereum. At the *application layer*, Ethereum clients execute smart contracts in EVM, where smart contracts are associated to Ethereum accounts. The *data layer* contains the blockchain data structures. The *consensus layer* assures a consistent state of the blockchain. Note that Ethereum plans to replace its current use of Proof-of-Work (PoW) with Proof-of-Stake (PoS). The *network layer* manages an Ethereum peer-to-peer (P2P) network of *nodes* or *clients* such that a node can always get the updated state of the blockchain from some active nodes.

*2.1.1 The Application Layer.* Ethereum supports two types of accounts: *externally owned accounts* (EOA) and *contract accounts*. An EOA is used to keep a user's funds in Wei, which is the smallest subdenomination of Ether and is worth $10^{-18}$ Ether. An EOA is associated with, and addressed by, a public key; access to an EOA is authenticated by showing the ownership of the corresponding private key. In contrast, a contract account is associated with a piece of executable bytecode (i.e., smart contracts), which defines some business logic of interest. An EOA or contract account has a dynamic state, which is defined by: (i) *nonce*, which tracks the number of *transactions* that have been initiated by the owner of the EOA or the number of contracts created by the contract account; (ii) *balance*, which is the amount of Wei (i.e., $10^{-18}$ Ether) owned by the EOA or contract account; (iii) *storageRoot*, which is the hash of the root of the account's storage data structure *trie* that records a contract's state variables associated to the corresponding piece of bytecode (i.e., not applicable to EOA); (iv) *codeHash*, which is the hash value of a contract account's bytecode (i.e., not applicable to EOA). The state of a blockchain is defined by the states of the accounts on the blockchain.

Smart contracts are DApp building-blocks. A DApp often has a user interface as its front-end and some smart contracts as its back-end. At the moment of writing, 3,410 DApps are running on top of Ethereum, including finance, governance, gambling, exchange, and wallet applications [59]. Some DApps issue their own cryptocurrency, called *tokens*, for purposes like Initial Coin Offering (ICO) and exchanges. An Ethereum-based token is a special kind of smart contract (e.g., ERC-20 [2]). Smart contracts execute in EVMs, which are quasi-Turing-complete machines using a stack-based architecture; the term "quasi" means the execution is limited by the amounts of gas offered by transactions.

*2.1.2 The Data Layer.* A *transaction* is an interaction between an EOA (called *sender*) and another EOA or contract account (called *recipient*). A transaction is specified by: (i) *nonce*, which is

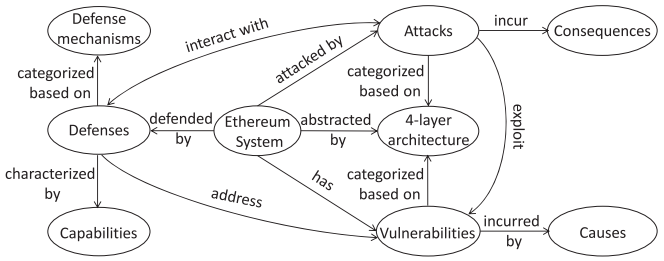


Fig. 2. An ontology of Ethereum systems security through the lens of vulnerabilities, attacks and defenses. In this ontology, the Ethereum system is *abstracted by* a four-layer architecture (i.e., application, data, consensus and network layers), *has* vulnerabilities, is *attacked by* attacks, and is *defended by* defenses; attacks *exploit* vulnerabilities, defenses *address* vulnerabilities, and attacks and defenses *interact with* each other via vulnerabilities.

a counter for tracking the total number of transactions that have been initiated by the sender; (ii) *recipient*, which specifies a transaction's destination EOA or contract account; (iii) *value*, which is the amount of money (unit: Wei) to be transferred from the sender to the recipient (if applicable); (iv) *input*, which is the bytecode or data corresponding to the purpose of the transaction; (v) *gasPrice* and *gasLimit*, which, respectively, specify the unit price and the maximum amount of gas the sender is willing to pay the winning miner of a *block* containing the transaction; (vi) $(v, r, s)$, which is the Elliptic Curve Digital Signature Algorithm (ECDSA) signature of the sender. Execution of a transaction updates the states of the accounts involved and therefore the blockchain. Owing to space limit, the lifecycle of an Ethereum transaction is depicted in Figure 1 of the Appendix.

Trie [184] is the data structure for storing Ethereum blockchain data (e.g., account states). Like a Patricia tree, a trie stores $(key, value)$ pairs and facilitates search as follows: The path from the root to a leaf node corresponds to a *key* and the leaf node contains a *value* (e.g., the state of an account). A block header may point to a state trie, a transaction trie (for bookkeeping transaction data), and a receipt trie (for bookkeeping the data related to the execution of transactions). Each contract account corresponding to a leaf or branch node on the state trie uses a separate storage trie to bookkeep the persistent data of the contract; this storage trie also uses a $(key, value)$ structure, where the position of each slot corresponds to a *key* and the contract's state variable in each slot corresponds to a *value*. Ethereum has a *single* state trie, because the state of the blockchain dynamically evolves.

*2.1.3 The Consensus Layer.* At the moment of writing, Ethereum takes about 14 seconds to create a block, meaning that multiple miners could create valid blocks simultaneously and that there could be many *stale* blocks. Ethereum uses a variant of the GHOST consensus protocol [170] to select the "heaviest" branch as the *main chain* where the "heaviest" branch is the sub-tree rooted at the fork in question and has the highest cumulative block difficulty [179], while noting that *stale* blocks are not on the main chain. Ethereum rewards not only the *regular* blocks on the main chain but also the stale blocks referred by a regular block. As illustrated in Figure 2 of the Appendix, the miner of a regular block receives one unit of "static block reward," which is worthy of 2 Ethers at the time of writing. To incentivize referencing to uncle blocks, the miner of a nephew regular block further receives $1/32$ of the static block reward for a reference (and for up to 2 references). The miner of the referenced uncle block is rewarded with $1 - d/8$ of the static block reward, where $1 \leq d \leq 6$ is the distance between the uncle block and the referencing nephew block.

*2.1.4 The Network Layer.* The Ethereum network is a structured P2P network where each node (i.e., client) stores a copy of the entire blockchain. For node discovery and routing purposes, each node maintains a dynamic routing table of 160 buckets and each bucket contains up to 16 entries of other nodes' IDs, IP addresses, UDP/TCP ports. Ethereum uses the RLPx protocol [125, 142] to discover target clients and uses the Ethereum Wire Protocol [50] to facilitate the exchange of Ethereum blockchain information (e.g., transactions, blocks) between clients.

*2.1.5 The Environment.* The Ethereum blockchain runs in an environment that operates across the four layers as follows: a web interface for users to interact with the Ethereum blockchain; a database for Ethereum clients to store the blockchain data; cryptographic mechanisms for security purposes; and the Internet infrastructure to support blockchain communications among Ethereum nodes. We separate the Ethereum blockchain architecture from the environment, because attacks against the Ethereum blockchain may come from the environment and these attacks may be better addressed in the environment rather than by the Ethereum blockchain.

## 2.2 Survey Methodology

*2.2.1 Scope.* Since we focus on Ethereum security, we systematize Ethereum vulnerabilities, attacks and defenses. Since these aspects are related to the programming language for writing smart contracts and the client software, we focus on the widely used Solidity and Geth [58] as well as Parity [57], respectively. The literature covered includes (i) papers published in major academic conferences and journals and (ii) preprints, whitepapers, forums, and Ethereum documentations.

*2.2.2 Methodology.* Our methodology is characterized as follows: Use our domain knowledge to identify and select literature and materials; and use a novel ontology to guide us in systematizing the selected literature and materials. First, we use our domain knowledge to identify and select literature and materials as follows. For academic literature, we use our domain knowledge to collect literature from major conferences (including ACM CCS, IEEE SP, NDSS, Usenix Security, Eurocrypt, Crypto, ACSAC, DSN, Financial Crypto, ICSE, FSE, and ASE), journals (including ACM CSUR, IEEE COMST, IEEE TDSC, IEEE T-IFS, IEEE Access) and preprints (arxiv.org); then, we recursively trace the relevant literature cited by the papers we have collected. Since academic literature does not contain all of the relevant materials (e.g., newly disclosed security incidents and techniques that have not been investigated by the academic community), we use Google to find Internet reports and blogs and then use our domain knowledge to select the relevant materials. Since Ethereum is evolving, we also consider Ethereum whitepapers and documents describing ongoing developments (e.g., Casper).

Second, we use a novel ontology to systematize the selected literature and materials. As highlighted in Figure 2, our ontology abstracts the Ethereum system as the four-layer architecture mentioned above and accommodates three key aspects of Ethereum security in vulnerabilities, attacks, and defenses. Under this ontology, vulnerabilities are categorized based on the layers at which they reside, attacks are categorized based on the layers at which they target, and defenses are categorized based on their defense mechanisms (e.g., proactive vs. reactive). More importantly, we aim to draw insights into: (i) What are the root causes and status of Ethereum vulnerabilities (i.e., eliminated or not)? (ii) What are the attack tactics (i.e., how the vulnerabilities are exploited to wage attacks) and their consequences? (iii) What are the defensive efforts that have been made and how effective are they? (iv) What are the relationships between vulnerabilities, attacks and defenses (i.e., which attacks exploit which vulnerabilities and which defenses address which vulnerabilities and defeat which attacks)?

It is worth mentioning that the taxonomy of vulnerabilities, attacks and defenses resulting from our methodology can be extended to accommodate new vulnerabilities, attacks and defenses that

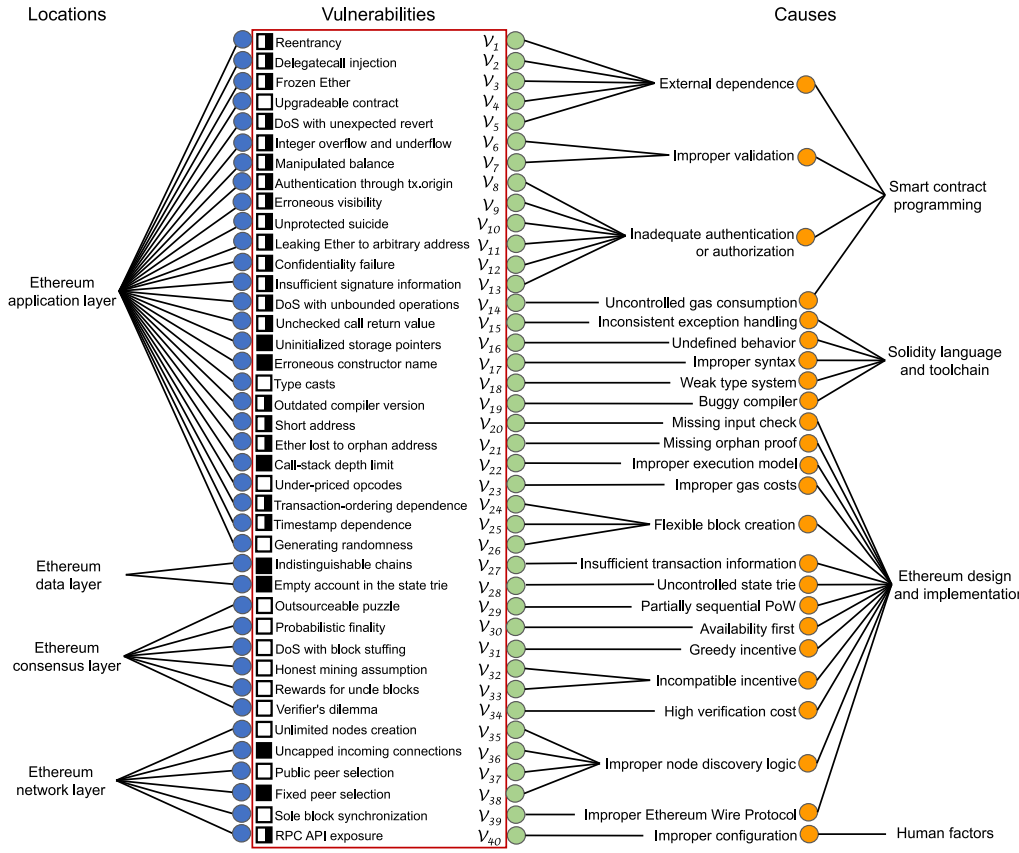


Fig. 3. A classification of Ethereum vulnerabilities and their state-of-the-art treatments, where a filled box ■ means the vulnerability has been eliminated already, an empty box □ means the vulnerability is open (i.e., has yet to be eliminated), and a half-empty half-filled box means the vulnerability can be avoided by best practice).

may have been discovered at the time of reading. If these new vulnerabilities, attacks, and defenses are not fundamentally different from the ones systematized in the present article, then they can be merged with their similar counterparts in present taxonomy; otherwise, they can be accommodated by extending the present taxonomy (i.e., vulnerabilities and attacks can be accommodated according to the layer or layers they are associated with, and defenses can be accommodated according to their category, such as proactive or reactive, and/or their capabilities).

To improve readability, we denote the vulnerabilities by $\mathcal{V}_1, \ldots, \mathcal{V}_{40}$, respectively. Similarly, we respectively denote the attacks by $\mathcal{A}_1, \ldots, \mathcal{A}_{29}$ and the defenses by $\mathcal{D}_1, \ldots, D_{51}$. We use "$\mathcal{A}_i(\mathcal{V}_j, \cdots)$" to denote that attack $\mathcal{A}_i$ exploits vulnerability $\mathcal{V}_j$ and possibly others.

## 3 VULNERABILITIES

Figure 3 highlights our classification of Ethereum vulnerabilities based on their *location*, *cause*, and *status* (i.e., *eliminated* vs. *can be avoided by best practice* vs. *open*). For ease of reference, we denote the 40 types of vulnerabilities as $\mathcal{V}_1, \ldots, \mathcal{V}_{40}$, respectively. In what follows, we group them according to location.

### 3.1 Vulnerabilities at the Application Layer

*3.1.1 Reentrancy ($V_1$).* This vulnerability was first observed from the DAO attack [5]; its variants were later reported in Reference [162]. The vulnerability occurs when an external callee contract calls back to a function in the caller contract *before* the caller contract finishes (i.e., cyclic calls in a sense). This allows the attacker to bypass the due validity check until the caller contract is drained of Ether or the transaction runs out of gas. The vulnerability is caused by: (i) a contract's control-flow decision relies on some of its state variable(s) that should be, but are not, updated by the contract itself before calling another contract [162]; and (ii) there is no gas limit when handing the control-flow to another contract. The vulnerability can be prevented by one of the following methods [49]: (i) assuring that a contract's state variables are updated *before* calling another contract; (ii) introducing a mutex lock on the contract state to assure that only the lock owner can change the state; (iii) using the transfer method to send money to other contracts, because this method only forwards 2,300 gas to the callee contract.

*3.1.2 Delegatecall Injection ($V_2$).* This vulnerability was first observed from an attack against the Parity wallet [16]. To facilitate code-reuse, EVM provides an opcode, DELEGATECALL, for inserting a callee contract's bytecode into the bytecode of the caller contract [130]. As a consequence, a malicious callee contract can directly modify (or manipulate) the state variables of the caller contract. This vulnerability is caused by the fact that a callee contract can update the caller contract's state variables. The vulnerability can be completely prevented by declaring a contract that is meant to be shared via the delegatecall as a library, which is stateless [30].

*3.1.3 Frozen Ether ($V_3$).* This vulnerability was first observed from another attack against the Parity wallet [13]. The vulnerability results from the ability of users to deposit their money to their contract accounts with the inability to spend their money from those accounts, effectively freezing their money. The vulnerability is caused by Reference [119]: (i) contracts not providing any function for spending money, relying on the money-spending function of another contract (as a library) and (ii) the callee contract (i.e., the library) being killed accidentally or deliberately. The vulnerability can be prevented by assuring that mission-critical functions, or money-spending functions in this case, are not outsourced to another contract.

*3.1.4 Upgradable Contract ($V_4$).* This vulnerability was first discussed in Reference [65]. The idea of contract upgrading was introduced to mitigate the problem that smart contracts, once deployed, cannot be modified even if they are later found to have vulnerabilities. To allow contract upgrading, there are two approaches: (i) splitting a contract into a *proxy contract* and a *logic contract* such that developers can upgrade the latter but not the former; and (ii) using a *registry contract* to bookkeep the updated contracts. While effective, these approaches introduce a new vulnerability: When the contract developer becomes malicious, the updated contract can be malicious. This vulnerability (i.e., insecure contact updating) remains to be an open problem.

*3.1.5 DoS with Unexpected Revert ($V_5$).* This vulnerability was first reported in Reference [49]. It occurs either when a transaction is reverted due to a caller contract encountering a failure in an external call, or the callee contract deliberately performs the revert operation to disrupt the execution of the caller contract. This vulnerability is caused by the execution of a caller contract being reverted by a callee contract. This vulnerability can be prevented by letting a recipient invoke a transaction to "pull" the money that was set aside by a sender for the recipient, which effectively prevents a sender's transaction from being reverted [38].

*3.1.6 Integer Overflow and Underflow ($V_6$).* This vulnerability was first observed from the attack against the BEC tokens [32]. It occurs when the result of an arithmetic operation falls outside

of the range of a Solidity data type, causing (for example) unauthorized manipulation to the attacker's balance [32] or other state variables. The vulnerability is caused by Solidity source code not performing proper validation on numeric inputs, and that neither the Solidity compiler nor the EVM provides integer overflow/underflow detection. This vulnerability can be prevented by using the *SafeMath* library [37] that handles these issues.

*3.1.7 Manipulated Balance ($\mathcal{V}_7$).* This vulnerability was first reported in Reference [34] and was also known as the "forcing Ether to contracts" vulnerability. This vulnerability occurs when a contract's control-flow decision relies on the value of `this.balance or address(this).balance`, which can be leveraged by an attacker to make itself the only one who can obtain the money; see Reference [30] for a detailed description. This vulnerability can be prevented by not using a contract's balance in any conditional statement [34].

*3.1.8 Authentication Through* `tx.origin` *($\mathcal{V}_8$).* This vulnerability was first discussed in Reference [12]. The `tx.origin` is a global variable in Solidity and refers to the original EOA that initiates the transaction in question. This vulnerability occurs when a contract uses `tx.origin` for authorization, which can be compromised by a phishing attack. This vulnerability can be prevented by using `msg.sender`, instead of `tx.origin`, for authentication, because `msg.sender` returns the account that incurred the message.

*3.1.9 Erroneous Visibility ($\mathcal{V}_9$).* This vulnerability was first observed in an attack against the Parity wallet [16]. It occurs when a function's visibility is incorrectly specified and thus permits unauthorized access. Specifically, Solidity provides four types of visibility to restrict access to a contract's functions, `namely`, `public`, `external`, `internal`, and `private`, which respectively says a function can be called arbitrarily, only externally, only internally (i.e., within the contract and its derived contracts), or only within the contract. Functions that should not be called from an external contract should be specified as `private` or `internal`. However, Solidity makes functions as `public` by default, allowing attackers to call improperly specified functions. Solidity (starting version 0.5.0) mitigates the vulnerability by making it mandatory for programmers to explicitly specify function visibility [39]. Still, this vulnerability cannot be prevented unless programmers correctly specify functions' visibilities.

*3.1.10 Unprotected Suicide ($\mathcal{V}_{10}$).* This vulnerability was first observed from an attack against the Parity wallet [13]. A contract can be killed by the contract's owner (or a trusted third-party) using the suicide or self-destruct method. When a contract is killed, its associated bytecode and storage are deleted. The vulnerability is caused by inadequate authentication enforced by a contract. The vulnerability can be mitigated by enforcing, e.g., multi-factor authentication, meaning a suicide operation must be approved by multiple parties [54].

*3.1.11 Leaking Ether to Arbitrary Address ($\mathcal{V}_{11}$).* This vulnerability was first reported in Reference [152]. The vulnerability occurs when a contract's funds can be withdrawn by any caller, who is neither the owner of the contract nor an investor who deposited funds to the contract. This vulnerability is caused by the failure to check a caller's identity when the caller invokes a function to send Ether to an arbitrary address. This vulnerability can be prevented by enforcing an adequate authentication on the functions for sending funds.

*3.1.12 Confidentiality Failure ($\mathcal{V}_{12}$).* This vulnerability was first observed from a multi-player game in Reference [92] and was also called *keeping secrets* in Reference [65]. It can be exploited to benefit an attacker. In blockchain, restricting the visibility of a variable or function does not assure that the variable or function is confidential because of the public nature of blockchain (i.e., details of transactions are publicly known). Although restricting a state variable to be `private` can

prevent other contracts from accessing it, anyone can figure out the value of a state variable from the relevant transaction data. This vulnerability is caused by the lack of confidentiality assurance for sensitive data (e.g., transaction information) in untrusted environment. A possible solution to preventing this vulnerability is to use cryptographic techniques, such as timed commitments [65, 92].

*3.1.13 Insufficient Signature Information ($\mathcal{V}_{13}$).* The vulnerability occurs when a digital signature is valid for multiple transactions, which can happen when one sender (say Alice) sends money to multiple recipients through a proxy contract (instead of initiating multiple transactions). In the proxy contract mechanism, Alice can send a digitally signed message off-chain (e.g., via email) to the recipients, similar to writing personal checks in the real world, to let the recipients withdraw money from the proxy contract via transactions. To assure that Alice does approve a certain payment, the proxy contract verifies the validity of the digital signature in question. However, if the signature does not give the due information (e.g., nonce, proxy contract address), then a malicious recipient can replay the message multiple times to withdraw extra payments. This vulnerability was first exploited in a replay attack against smart contracts [36]. This vulnerability can be prevented by incorporating the due information in each message, such as a nonce value and timestamps [62].

*3.1.14 DoS with Unbounded Operations ($\mathcal{V}_{14}$).* This vulnerability was first observed from the `GovernMental` contract [15] and its variants were later discussed in Reference [104]. Recall that each block has a "gas limit" field that specifies the maximum total amount of gas that can be consumed by the transactions in a block. This vulnerability occurs when the amount of gas required for executing a contract exceeds the block gas limit. It is caused by improper programming with unbounded operations in a contract (e.g., loop over a large array). This vulnerability can be mitigated by assuring that (i) contracts never use loops over data structures, especially those data structures that can be operated by EOA; and (ii) when a contract has to use loops over data structures, it should keep track of the loop and resume the aborted execution when the sender of the transaction re-invokes the same contract (to finish the execution of the contract).

*3.1.15 Unchecked Call Return Value ($\mathcal{V}_{15}$).* This vulnerability [139] is also known as *mishandled exceptions*. It has two variants, called *gasless send* and *unchecked send* [65, 121]. Recall that Solidity provides two methods for a contract to call another: (i) directly referencing to a callee contract's instance; (ii) using one of the following four low-level methods: send, call, delegatecall and callcode. There is a discrepancy in Solidity's handling of exceptions occurring in the execution of callee contracts [139]: if an exception occurs in case (i), then the exception is automatically propagated back to the caller and the transaction is reverted entirely; if an exception occurs in case (ii), then the callee contract returns FALSE back to the caller contract. This discrepancy can lead to unintended transactions unless the caller contract carefully addresses the discrepancy. At the moment of writing, neither the Solidity compiler nor the EVM addresses the discrepancy. This vulnerability can be prevented by letting a caller contract check and address the discrepancy mentioned above.

*3.1.16 Uninitialized Storage Pointer ($\mathcal{V}_{16}$).* This vulnerability was first reported in Reference [41]. Recall that in Solidity, the contract state variables are always laid out consecutively in storage, starting from slot 0. For a compound local variable (e.g., struct, array, or mapping), a reference is assigned to an unoccupied slot in the storage to point to the state variable. If the local variable is not explicitly initialized, then the local variable's reference points to slot 0 by default, causing the content starting from slot 0 to be overwritten. This vulnerability is caused by Solidity's treatment of uninitialized compound local variables. This vulnerability has been eliminated by Solidity

```
1 contract CounterLibrary { function add(uint) public returns (uint) }
2 contract CounterLib { function add(uint) public returns (uint) }
3 contract Game {
4     function play(CounterLibrary c) public { c.add(1) ; } }
```

Fig. 4. The type casts vulnerability.

compiler, starting version 0.5.0, by reporting an error to contracts that contain uninitialized storage pointers [39].

*3.1.17 Erroneous Constructor Name ($\mathcal{V}_{17}$).* This vulnerability was first observed from the `Rubixi` contract [56], where the constructor function has an incorrect name that allows anyone to become the owner of the contract. Prior to Solidity version 0.4.22, a function declared with the same name as the contract's is considered as the contract constructor, which is executed only when creating the contract. If the constructor's name is misspelled by a programmer for whatever reason, then the intended constructor becomes a public, normal function that can be invoked by *any* EOA. This vulnerability is caused by Solidity not providing a special syntax to distinguish a constructor function from a regular function. The vulnerability has been eliminated in Solidity version 0.4.22 by introducing the new keyword `constructor` [40].

*3.1.18 Type Casts ($\mathcal{V}_{18}$).* This vulnerability was first reported in Reference [65]. Recall that a contract written in the Solidity language can call another contract by directly referencing the callee contract's instance. As illustrated in Figure 4, contract `Game` calls function add() in contract `CounterLibrary` by referencing its instance c (Line 5). When function play() (Line 4) is invoked, the argument specifying the callee contract's address is cast to `CounterLibrary`. However, the Solidity compiler only checks whether or not `CounterLibrary` declared function add(), but cannot check whether or not the address argument conforms to that of `CounterLibrary`'s. If the address associated to `CounterLib` (Line 2) contains a function that is named add() and has the same declaration, then the add() function in `CounterLib` is executed, instead of the desired add() function in `CounterLibrary`. As a consequence, the EVM can be misled to run the attacker's contract. This vulnerability is caused by the incompetent type system of Solidity. Currently, there is no feasible way to avoid the vulnerability.

*3.1.19 Outdated Compiler Version ($\mathcal{V}_{19}$).* This vulnerability was first reported in Reference [54]. It occurs when a contract uses an outdated compiler, which contains bugs and thus makes a compiled contract vulnerable. This vulnerability can be prevented by using an up-to-date compiler.

*3.1.20 Short Address ($\mathcal{V}_{20}$).* This vulnerability was first discussed in Reference [17]. Recall that in a `contract-invocation` transaction, the function selector and arguments are encoded in the *input* field as follows: the first four bytes specify the callee function and the remaining data arranges arguments in chunks of 32 bytes. However, if the length of the encoded arguments is shorter than expected, then EVM will auto-pad extra zeros to the arguments to make up for 32 bytes. Consider function transfer(address addr, uint tokens) as an example. If the trailing (i.e., last) byte of addr is left off, then two extra hex zeros will be added to the end of tokens, which amplifies the number of tokens being sent. This vulnerability is caused by EVM not checking the validity of addresses. This vulnerability can be prevented by checking the length of a transaction's input (i.e., `msg.data`) [14].

*3.1.21 Ether Lost to Orphan Address ($\mathcal{V}_{21}$).* This vulnerability was first reported in Reference [65]. When transferring money, Ethereum only checks that the length of the recipient's address is no greater than 160-bit but not the validity of the recipient's address. If money is sent to a non-existing orphan address, then Ethereum automatically registers for the address rather than terminating the transaction. Since the address is not associated to *any* EOA or contract account, no

one can withdraw the transferred money, which is effectively lost. This vulnerability is caused by EVM not being orphan-proof. At the moment of writing, this vulnerability can only be prevented by manually assuring the correctness of the recipient's address.

*3.1.22 Call-stack Depth Limit ($\mathcal{V}_{22}$).* This vulnerability was first reported in Reference [3]. Recall that in the original specification of the Ethereum execution model [184], EVM's call-stack has a hard limit of 1,024 frames. When a contract calls another contract, the call-stack depth of the transaction increases by one; when the number of nested calls exceeds 1,024, Solidity throws an exception and aborts the call. An attacker can recursively call a contract, which may be deployed by the attacker, 1,023 times and then call a victim contract to reach the stack depth limit, which causes any subsequent external call made by the victim contract to fail. Since Solidity does not propagate exceptions in low-level external calls, the victim contract may not be aware of the failure. This vulnerability is caused by EVM's inadequate execution model, and has been eliminated by the hard fork for EIP-150, which re-defines the gas-consumption rules of external calls to make it impossible to reach 1,024 in call stack depth [6].

*3.1.23 Under-priced Opcodes ($\mathcal{V}_{23}$).* This vulnerability was first observed from two DoS attacks [11, 20, 82]. Recall that Ethereum uses the gas mechanism to prevent the abuse of computing resources (e.g., CPU, disk, network). This vulnerability occurs when a contract contains many under-priced opcodes that consume a large amount of resources at a low gas cost, meaning that the execution of the contract wastes a lot of computing resources. This vulnerability is caused by the failure in properly setting the gas cost for consuming computing resources. To mitigate this vulnerability, Ethereum has raised the gas cost for the opcodes that were abused to launch the two DoS attacks described in Reference [6]. However, it is not clear whether the vulnerability can be completely prevented by this mechanism or not [82].

*3.1.24 Transaction Ordering Dependence (a.k.a. Front Running; $\mathcal{V}_{24}$).* This vulnerability was first discussed in Reference [139]. It refers to the concurrency issue that the forthcoming state of blockchain depends on the execution order of transactions, which is, however, determined by the miners. Typically, miners group and order transactions into a new block based on the reward offered by the transactions. Since transactions are publicly broadcast to the network, a malicious EOA can offer a higher *gasPrice* to have its transactions assembled into blocks sooner than the others'. Moreover, a malicious miner can always pick up its own transactions regardless of the *gasPrice*. This vulnerability is caused by that the state of a contract depending on how miners select transactions to assemble into blocks. This vulnerability can be mitigated by using a cryptographic commit-reveal scheme to hide the information (e.g., *gasPrice*, *value*) offered by transactions [9, 22], or by introducing a guard condition to assure that an invocation of a contract either returns the expected output or fails [139].

*3.1.25 Timestamp Dependence ($\mathcal{V}_{25}$).* This vulnerability [139] occurs when a contract uses the `block.timestamp` as part of the triggering condition when executing a critical operation (e.g., money transfer) or as the source of randomness, which can be manipulated by a malicious miner. The vulnerability is caused by Ethereum only requiring that a timestamp be greater than the timestamp of its parent block and be within 900 seconds of the current clock. If a contract uses a timestamp-based condition (e.g., `block.timestamp %25 == 0`) to determine whether or not to transfer money, then a malicious miner can slightly shift the timestamp to satisfy the condition to benefit the attacker. This vulnerability can be prevented by not using `block.timestamp`.

*3.1.26 Generating Randomness ($\mathcal{V}_{26}$).* This vulnerability was first reported in Reference [35]. Many gambling and lottery contracts select winners randomly, for which a common practice

is to generate a pseudorandom number based on some initial private seed (e.g., `block.number`, `block.timestamp`, `block.difficulty` or `blockhash`). However, these seeds are fully controlled by miners, meaning that a malicious miner can manipulate these variables to make itself the winner. This vulnerability is caused by manipulable entropy sources. There are several proposals for addressing this problem, and each proposal has its own pros and cons. The Oracle RNG proposal [52] uses existing external services to generate random numbers off-chain and then send back to the requesting contract, meaning that there is a single-point-of-failure in the Oracle RNG. The RANDAO proposal [53] initially uses a distributed cryptographic commitment scheme for multiple participants to jointly generate a random number, which may be subject to the last-revealer attack where the last participant can create a bias by choosing whether or not to reveal its committed entropy [96]. To resolve this issue, Verifiable Delay Functions (VDFs) [72] are introduced to assure that no participant can compute the random seed before submitting its own entropy. However, existing VDFs are not post-quantum secure [73, 181] and have limited throughput.

### 3.2 Vulnerabilities at the Data Layer

*3.2.1 Indistinguishable Chains ($V_{27}$).* This vulnerability was first observed from the cross-chain replay attack when Ethereum was divided into two chains, namely, ETH and ETC [10]. Recall that Ethereum uses ECDSA to sign transactions. Prior to the hard fork for EIP-155 [7], each transaction consisted of six fields (i.e., *nonce*, *recipient*, *value*, *input*, *gasPrice*, and *gasLimit*). However, the digital signatures were not chain-specific, because no chain-specific information was even known back then. As a consequence, a transaction created for one chain can be reused for another chain. This vulnerability has been eliminated by incorporating chainID into the fields.

*3.2.2 Empty Account in the State Trie ($V_{28}$).* This vulnerability was first observed from a DoS attack reported in References [20, 82]. An *empty* account is an account that has zero nonce, zero balance, and no code or storage associated to it. An empty account is functionally equivalent to a non-existing account, except that an empty account needs to be bookkept in the Ethereum state trie and thus increases the synchronization and transaction processing time. This means that an attacker can incur a large number of empty accounts to substantially increase the the synchronization and transaction processing time, effectively causing a DoS attack [20, 82]. An empty account can be incurred by an attacker using the SUICIDE opcode to transfer zero Ethers to a non-existing account. This vulnerability was caused by the lack of control over empty accounts in the state trie. This vulnerability has been eliminated by the hard fork for EIP-161 [8], which removed those empty accounts from the state trie and prevents any empty account from being stored in the state trie.

### 3.3 Vulnerabilities at the Consensus Layer

*3.3.1 Outsourceable Puzzle ($V_{29}$).* This vulnerability was reported in Reference [179]. Recall that Ethereum adopts the PoW puzzle called Ethash, which was meant to be ASIC-resistant and be able to limit the use of parallel computing (owing to the fact that the vast majority of a miner's effort will be reading a dataset via the limited memory bandwidth). However, a crafty miner can still divide the task of searching for a puzzle solution into multiple smaller tasks and then outsource them. This vulnerability is caused by Ethash only making the puzzle solution partially sequential in preimage search, rather than relying on sequential PoW. Several puzzles are proposed to cope with this problem [88, 146]. However, they have not been adopted by the Ethereum community.

*3.3.2 Probabilistic Finality ($V_{30}$).* This vulnerability is inherent to PoW and PoS protocols [66, 91]. The vulnerability refers to the fact that Ethereum blockchain can only achieve a probabilistic rather than a deterministic assurance that a newly generated block will be finalized in the

blockchain. The deeper a block sinks into the chain, the more likely it will not be reverted. This vulnerability is caused by the design that Ethereum blockchain favors availability over consistency, which is choice made under the CAP theorem [100]. To mitigate this vulnerability, it is recommended to consider a new block persistent in the main chain after seeing 12 blocks. Still, the probability of finality is affected by many factors (e.g., communication delays) [103]. In principle, the 50% fault tolerance of *Nakamoto consensus* implies that a powerful attacker (e.g., controlling 51% hashrate for PoW) can formulate a new main chain at will [133, 164]. To mitigate this vulnerability, Ethereum's upcoming PoS upgrade, known as Casper, plans to use periodic check-pointing [77] and a safety oracle [149]. Nevertheless, reaching fast and deterministic finality is still an open problem.

*3.3.3 DoS with Block Stuffing ($V_{31}$).* This vulnerability was first observed from the Fomo3D contract [23]. The vulnerability entails only the attacker's transactions being included in the newly mined blocks while others are abandoned by miners for a period of time. This can happen when the attacker offers a higher *gasPrice* to incentivize the miners to select the attacker's transactions. This vulnerability is caused by the greedy mining incentive mechanism. At the moment of writing, there is no solution to prevent this vulnerability.

*3.3.4 Honest Mining Assumption ($V_{32}$).* This vulnerability was first reported in Reference [99], referring to that the assumption inherent to the *Nakamoto consensus* protocol—honest mining (i.e., including the most valuable transactions in new blocks) is the most profitable strategy for each miner—may not be true. This is because it can be more profitable to deviate from the honest mining strategy, such as conducting selfish mining [161], accepting bribes [144], and reaping ordering optimization fees [89]. This vulnerability is caused by the consensus protocol for not being incentive-compatible. At the moment of writing, this vulnerability remains to be an open problem.

*3.3.5 Rewards for Uncle Blocks ($V_{33}$).* This vulnerability was reported in References [102, 154, 161]. It refers to the uncle-rewarding mechanism for coping with the increase in stale blocks caused by fast block generation. However, this mechanism had a side-effect in allowing selfish miners to make their stale blocks become uncle blocks and receive rewards, effectively incentivizing selfish mining and double-spending. At the moment of writing, it is not clear how to eliminate this vulnerability.

*3.3.6 Verifier's Dilemma ($V_{34}$).* This vulnerability was first reported in Reference [140], referring to when the verification of a new transaction requires nontrivial computation effort, miners are subject to attacks regardless of whether they choose to verify the transaction or not. If miners verify a computationally heavy transaction, then they will spend a significant amount of time and give attackers an advantage in the race for the next block; if miners accept the transaction without verification, then the blockchain may include an incorrect transaction. This vulnerability is caused by the high cost in verifying resource-demanding transactions in Ethereum. This vulnerability can be mitigated by limiting the amount of computation that is required for verifying all transactions in a block [140]. However, it is not clear how to eliminate this vulnerability.

### 3.4 Vulnerabilities at the Network Layer

*3.4.1 Unlimited Nodes Creation ($V_{35}$).* This vulnerability was reported for the Geth client prior to its version 1.8 [142]. In the Ethereum network, each node is identified by a unique ID, which is a 64-byte ECDSA public key. An attacker could create an unlimited number of nodes on a single machine (i.e., with the same IP address) and use these nodes to monopolize the incoming and outgoing connections of some victim nodes, effectively isolating the victims from the other peers

in the network. This vulnerability is caused by the weak restriction on the node generation process. This vulnerability can be eliminated by using a combination of IP address and public key as node ID. This countermeasure has not been adopted by the Geth developers who argue that it has a negative impact on the usability of the client.

*3.4.2 Uncapped Incoming Connections ($\mathcal{V}_{36}$).* This vulnerability was in the Geth client prior to its version 1.8 [142]. Each node can have a total number of `maxpeers` (with a default value 25) connections at any point in time, and can initiate up to $\lfloor (1 + \texttt{maxpeers})/2 \rfloor$ outgoing TCP connections with the other nodes. However, there was no upper limit on the number of incoming TCP connections initiated by the other nodes. This gives the attacker an opportunity to eclipse a victim by establishing `maxpeers` many incoming connections to a victim node that has no outgoing connections. This vulnerability has been eliminated in Geth v1.8 by enforcing an upper limit on the number of incoming TCP connections to a node, with a default value $\lfloor \texttt{maxpeers}/3 \rfloor = 8$.

*3.4.3 Public Peer Selection ($\mathcal{V}_{37}$).* This vulnerability was detected in Geth client prior to its version 1.8 [142]. Recall that the Ethereum P2P network uses a modified Kademlia DHT for node discovery and that each node maintains a routing table of 256 *buckets* for storing information about the other nodes. The buckets are arranged based on the XOR distance between a node's ID and its neighboring node's ID [126]. When a node, say $A$, needs to locate a target node, $A$ queries the 16 nodes in its bucket that are relatively close to the target node and asks each of these 16 nodes, say $B$, to return the 16 IDs of $B$'s neighbors that are closer to the target node. The process iterates until the target node is identified. However, the mapping from node IDs to buckets in the routing table is public, meaning that the attacker can freely craft node IDs that can land in a victim node's buckets and insert malicious node IDs into the victim node's routing table [142]. This vulnerability can be limited by making the "node IDs to buckets" mapping private. This countermeasure has not been adopted by the Geth developers who argue that it has a negative impact on the usability of the client.

*3.4.4 Fixed Peer Selection ($\mathcal{V}_{38}$).* This vulnerability was reported for the Geth client prior to its version 1.9 [115]. The vulnerability refers to the Geth client always fetching the heads of randomly chosen buckets when selecting nodes from its routing table to establish outbound connections. Since the nodes in each bucket are sorted by activity, an attacker could make its node always stay ahead of the other nodes by regularly sending message to the Geth client. This vulnerability has been eliminated in Geth v1.9 by selecting nodes uniformly at random from the set of all nodes in the routing table instead of only the heads of each bucket.

*3.4.5 Sole Block Synchronization ($\mathcal{V}_{39}$).* This vulnerability was first reported in Reference [185]. It allows an attacker to partition the Ethereum P2P network without monopolizing the connections of a victim client. Recall that each block header contains a *difficulty* field, which records the mining difficulty of the block. The total difficulty of the blockchain, denoted by *totalDifficulty*, is the sum of the difficulty of the blocks up to the present one. When a client, say $A$, receives from, say client $B$, a block of which the difference *totalDifficulty* − *difficulty* is greater than the *totalDifficulty* at the blockchain stored on client $A$ (meaning that client $A$ missed a number of blocks), $A$ should start a block synchronization with $B$. Ethereum only allows a client to synchronize with one other client at a time (for network load considerations). This means that if client $B$ is malicious and deliberately delays the synchronization in response to $A$'s request, the blockchain at client $A$ is stalled and $A$ rejects every subsequent block, which may facilitate double-spending and DoS attacks. This vulnerability can be mitigated by synchronizing $A$ with multiple nodes, which, however, increases the network load.

*3.4.6 RPC API Exposure ($\mathcal{V}_{40}$).* This vulnerability was first observed from the attack against the Geth and Parity clients [27]. The JSON-RPC of Ethereum clients provide various APIs for EOAs to communicate with the Ethereum network. For security purposes, the interface should only be available locally and not be accessible from the internet. However, the standard port 8545 assigned to JSON-RPC can be accessed remotely in the Geth and Parity clients by default, which makes it possible for an attacker to call these remote clients via a JSON request [180]. Once having access to the remote client, the attacker can obtain sensitive data and perform certain unauthorized actions on the remote client. The vulnerability is caused by insecure API design and improper configuration. The vulnerability can be prevented by configuring the listening port (rather than using the default one) and adding access control to filter remote RPC calls.

### 3.5 Further Analysis of Vulnerability Causes

Now, we present a taxonomy of the root causes of the vulnerabilities reviewed above. As highlighted in Figure 3, the vulnerabilities are caused by incompetence or flaws in *smart contract programming*, *solidity language and toolchain*, *Ethereum design and implementation*, and *human factors*.

*3.5.1 Smart Contract Programming.* These causes can be further divided into four sub-causes: *external dependence*, meaning a contract's execution relies on the behavior of an external contract; *improper validation*, meaning a failure in checking a condition allows the passing of an invalid input; *inadequate authentication or authorization*, causing failures in checking a caller's identity or privilege when the caller attempts to access a protected data item or functionality; and *uncontrolled gas consumption*, meaning a failure in gas allocation permits a DoS attack. These causes led to 14 types of vulnerabilities. From Figure 3, we draw the following insights. First, 6 (out of the 14) types of vulnerabilities are caused by inadequate authentication and authorization, leading to:

Insight 1. *Authentication and authorization failures in Ethereum smart contracts are a major problem that calls for standardized stronger identity management solutions and access control mechanisms.*

Second, 5 (out of the 14) types of vulnerabilities are caused by external dependence (i.e., when invoking a smart contract, the control-flow is transferred to another contract that may be malicious), leading to:

Insight 2. *External dependence makes it hard, if not impossible, to assure the security of Ethereum smart contracts, highlighting the importance of implementing adequate security auditing on external calls.*

Third, only 4 (i.e., $\mathcal{V}_6, \mathcal{V}_9, \mathcal{V}_{13}, \mathcal{V}_{14}$) of the 14 types of vulnerabilities exist in traditional software and the other 10 are unique to Ethereum smart contract programming. Among these 10, only $\mathcal{V}_4$ cannot be prevented by best practices and the other 9 types of vulnerabilities are incurred by programmers' misunderstanding of Solidity.

Insight 3. *Incompetent Ethereum smart contract programming introduces many new kinds of vulnerabilities, highlighting the importance of standardizing domain-specific best practices for the new programming paradigm.*

*3.5.2 Solidity Language and Tool Chain.* These causes can be further divided into five subcauses: one is *buggy compiler* (i.e., insufficient tool chain support) and the other four are related to improper design of the Solidity language, namely, (i) *inconsistent exception handling* between direct call and low-level calls, (ii) *undefined behavior* of uninitialized storage pointers, (iii) *improper*

*syntax* of constructor function, and (iv) *weak type system* with flexible typing rules). These five sub-causes contribute to five types of vulnerabilities. According to Sebesta's criteria [168], *syntax design*, *exception handling*, and *type checking* are three important factors affecting a language's reliability. Solidity falls short in these aspects and causes vulnerabilities, leading to:

Insight 4. *The unreliability of Solidity makes Ethereum smart contracts vulnerability-prone, highlighting the importance of reliable programming languages.*

*3.5.3 Ethereum Design and Implementation.* These causes can be further divided into 14 sub-causes, which are related to, among other things, EVM, blockchain, PoW consensus, incentive mechanism, and P2P protocol. These causes cut across the application, data, consensus, and network layers. The root causes related to EVM include: (i) *missing input check*, meaning no check on the validity of a transaction's data; (ii) *missing orphan proof*, meaning no check on a nonexistent recipient address; (iii) *improper execution model*, meaning the behavior of EVM is not properly specified; and (iv) *improper gas costs*, meaning the gas costs of EVM opcodes are not properly specified.

The root causes related to blockchain include: (i) *flexible block creation*, meaning there are no restrictions on miners when they create blocks, allowing them to create blocks to favor themselves; (ii) *insufficient transaction information*, meaning that a transaction can be accepted by multiple Ethereum blockchains (e.g, ETH, ETC), rather than a specific blockchain, owing to the lack of information specifying a target blockchain; and (iii) *uncontrolled state trie*, meaning that there are no restrictions on the accounts that can be stored in the state trie.

The root causes related to the PoW consensus protocol include: (i) *partially sequential PoW*, meaning that Ethereum's PoW puzzle can still be outsourced to different miners; and (ii) *availability first*, meaning that the Ethereum consensus protocol prefers availability over consistency.

The root causes related to incentive mechanism include: (i) *greedy incentive*, meaning that a miner always selects transactions with higher *gasPrice*; (ii) *incompatible incentive*, meaning that a miner can get a higher payment by deviating from the consensus protocol; and (iii) *high verification cost*, meaning that a miner may skip the verification of resource-consuming transactions to gain an advantage in the mining race.

The root causes related to the P2P protocol include: (i) *improper node discovery logic*, meaning that the Kademlia-based bucket structure and node discovery algorithms are not properly designed; and (ii) *improper Ethereum wire protocol*, meaning that the blockchain synchronization algorithm is not properly designed.

The root causes mentioned above contributed 20 types of vulnerabilities, cutting across the application, data, consensus, and network layers. Among these 20, at least 8 types of vulnerabilities (i.e., $\mathcal{V}_{22}$, $\mathcal{V}_{23}$, $\mathcal{V}_{27}$, $\mathcal{V}_{31}$, $\mathcal{V}_{33}$, $\mathcal{V}_{35}$, $\mathcal{V}_{36}$, $\mathcal{V}_{39}$) are incurred by the arbitrary choices of parameters in Ethereum specification and implementation without thorough analysis, which violates the *open design* principle for computer security [166]. This leads to:

Insight 5. *Arbitrary choices of parameters in Ethereum specification and implementation caused many vulnerabilities in Ethereum blockchain systems, highlighting the importance of executing the "open design" principle.*

In addition, 11 (out of the 20) types of vulnerabilities remain to be fixed, suggesting:

Insight 6. *Vulnerabilities in Ethereum blockchain are harder to cope with than vulnerabilities in other systems, hinting that Ethereum blockchain is inherently more complex.*

*3.5.4 Human Factors.* This cause includes *improper configuration*, meaning that an Ethereum client is installed with incorrect permissions, which can cause serious security vulnerability.

```
1  contract Wallet {
2      address _walletLibrary = new WalletLibrary();
3      address owner;
4      ...
5      function() payable {
6          if (msg.data.length > 0)
7              _walletLibrary.delegatecall(msg.data);
8      }
9  }
```

```
10 contract WalletLibrary {
11     ...
12     function initWallet(address[] _owners, uint _required,
           ↪ uint _daylimit){
13         initDaylimit(_daylimit);
14         initMultiowned(_owners, _required);
15     }
16 }
```

Fig. 5. Simplified vulnerable Parity multisignature wallet.

Asking an average user to carefully manage permissions indicates insufficient usability. This leads to:

Insight 7. *There is an inherent trade-off between flexibility in configuring Ethereum clients and usability, but inadequate usability often leads to vulnerabilities.*

## 4 ATTACKS

Corresponding to the presentation of vulnerabilities, we group the 29 attacks we consider according to the locations of the vulnerabilities they exploit. For each attack, we describe its *history*, *cause*, *tactic*, and *direct consequence*. For ease of reference, we denote the 29 attacks by $\mathcal{A}_1, \ldots, \mathcal{A}_{29}$, respectively. Note that some of these 29 attacks may correspond to the same type of attacks (i.e., sharing the same name), but they exploit different vulnerabilities or vulnerability combinations and/or cause different consequences. For example, although $\mathcal{A}_2$ and $\mathcal{A}_3$ belong to the *parity multisignature wallet attack*, $\mathcal{A}_2$ exploits $\mathcal{V}_2$ and $\mathcal{V}_9$ to cause *unauthorized code execution*, while $\mathcal{A}_3$ exploits $\mathcal{V}_3$ and $\mathcal{V}_{10}$ to cause *DoS*.

### 4.1 Attacks at the Application Layer

*4.1.1 The DAO Attack ($\mathcal{A}_1$).* The contract `DAO` is a financial application running on top of Ethereum. In June 2016, it was attacked to cause the loss of US$60M [5]. DAO is an application by which investors vote on investment proposals for spending their money (i.e., "investment crowdsourcing"). Once a proposal is approved by a majority, the money approved by the supporting investors is moved to the proposer's account; the money owned by the investors opposing the proposal is respectively "refunded" to each of them via newly created contract accounts. This mechanism was implemented in the splitDAO() function; Figure 3 in the Appendix shows how the attack exploits the *reentrancy* vulnerability ($\mathcal{V}_1$).

*4.1.2 Parity Multisignature Wallet Attacks ($\mathcal{A}_2$ and $\mathcal{A}_3$).* In Ethereum, a multisignature wallet is a smart contract that requires multiple private keys to unlock a wallet. As shown in Figure 5, a multisignature wallet supported by the Parity client consists of two contracts: (i) a library contract called `WalletLibrary`, which implements all of the core functions of a wallet; and (ii) an actual `Wallet` contract, which holds a reference (i.e., `_walletLibrary`) that forwards all of the unmatched function calls to the library contract via delegatecall (Line 7). The Parity multisignature wallet was compromised twice in 2017. These incidents are reviewed below.

The first attack ($\mathcal{A}_2$) exploited the *delegatecall injection* vulnerability ($\mathcal{V}_2$) and the *erroneous visibility* vulnerability ($\mathcal{V}_9$) to drain Ethers approximately worth US$31M [16]. The attacker took over the ownership of contract `Wallet` by sending a transaction to the contract with msg.data containing initWallet() as the callee function (Line 12). Since contract `Wallet` did not provide a function named initWallet(), the contract's fallback function was triggered to delegate the wallet initialization task to the function initWallet() in `WalletLibrary`, which replaced the original multi-owner

```
1 function batchTransfer(address[] _receivers, uint256 _value)
      ↪ public whenNotPaused returns (bool) {
2   uint cnt = _receivers.length;
3   uint256 amount = uint256(cnt) * _value;
4   require(_value > 0 && balances[msg.sender] >=
      ↪ amount);
5   ... // transfer to specified recipients
6 }
```

Fig. 6. The vulnerable function in BECToken.

of contract `Wallet` with the attacker's address specified in msg.data. This attack succeeds when the following four conditions are satisfied simultaneously [16, 18]: (i) the function initWallet() in the library was not specified as an *internal* one, meaning it can be externally called via delegatecall; (ii) the `WalletLibrary` was actually a stateful contract, meaning it can change the state of `Wallet`; (iii) the function initWallet() did not check whether or not the `Wallet` contract had already been initialized (if so, no more initialization should be done); (iv) the `Wallet`'s fallback function did not check the function being called, but forwarded any unmatched calldata to `WalletLibrary`, allowing unintended invocations.

The second attack ($\mathcal{A}_3$) exploited the *unprotected suicide* vulnerability ($\mathcal{V}_{10}$) and the *frozen Ether* vulnerability ($\mathcal{V}_3$), freezing US$280M in the affected wallets forever [13]. This attack was incurred by the response to the first attack. The response was to add a modifier, `only_uninitialized`, to protect function initWallet() such that a re-initialization of `Wallet` via delegatecall will throw an exception and be rejected by the modifier. However, the shared `WalletLibrary` itself was left uninitialized, allowing an attacker to bypass the `only_uninitialized` modifier and set himself as the owner of the `WalletLibrary` [93]. Once taking over the library, the attacker invoked the suicide method to kill the library, causing the `Wallet` contracts relying on the library unusable.

*4.1.3 BECToken Attack ($\mathcal{A}_4$).* BECToken, an ERC-20 contract, was attacked in April 2018 by an exploitation of the *integer overflow* vulnerability ($\mathcal{V}_6$), causing a large number of stolen tokens and a temporary shutdown of token trading at exchange [25]. The vulnerability was in the function batchTransfer() shown in Figure 6, and the function was meant for users to transfer tokens to multiple recipients via two arguments: one specifying the array of the recipients' addresses and the other specifying the respective number of tokens. The statement at Line 3 calculates the total number of tokens the sender should pay for a particular transaction, but may have the following integer overflow: By setting `_value` to $2^{255}$ and `_receivers` to two accounts controlled by the attacker, the attack overflows the 256-bit variable amount and makes it zero [131]. As a consequence, the attack bypasses the two checks at Line 4, sending the two receivers extremely large numbers of tokens.

*4.1.4 GovernMental Attacks ($\mathcal{A}_5$, $\mathcal{A}_6$, $\mathcal{A}_7$, and $\mathcal{A}_8$).* The contract `GovernMental` was an array-based pyramid Ponzi scheme, where the last participant wins a jackpot if no one joins the scheme within 12 hours after the last participant [68]. The contract has four vulnerabilities [65], which allowed the following four attack tactics and explains why $\mathcal{A}_5$, $\mathcal{A}_6$, $\mathcal{A}_7$, and $\mathcal{A}_8$ belong to the same type of attacks. The first attack, denoted by $\mathcal{A}_5$, exploits the *DoS with unbounded operations* vulnerability ($\mathcal{V}_{14}$) that when the array bookkeeping the number of participants becomes too large, the amount of gas required for operating on the array will go beyond the maximum gas that is permitted for assembling a block. This effectively halts the transaction and the winner cannot receive the 1,100 ETH jackpot. The second attack, denoted by $\mathcal{A}_6$, exploits the *unchecked call return value* vulnerability ($\mathcal{V}_{15}$) that the contract does not check the returned value when sending jackpot to the winner and the *call-stack depth limit* vulnerability ($\mathcal{V}_{22}$). As a consequence, the owner of the malicious contract `GovernMental` can wage the attack to steal a victim participant's jackpot money by resetting the contract state variable that records participants' information. More specifically, the attack proceeds as follows: (i) the owner of `GovernMental` deploys another malicious contract,

```
1  contract HYIP {
2      Investor[] private investors;
3      ...
4      function performPayouts() {
5          for(uint idx = investors.length; idx−− > 0; ) {
6              uint payout=(investors[idx].amount*33)/1000;
7              if(!investors[idx].addr.send(payout)) throw;
8          }
9  }}
```

```
10 contract Mallory {
11     bool private attack = true;
12     function() payable {
13         if (attack) throw;
14     }
15     function stopAttack() {
16         if(msg.sender == owner) attack = false;
17     }
18 }
```

Fig. 7. Simplified HYIP contract and attack.

say $X$; (ii) the owner invokes $X$, which further recursively calls $X$ itself for 1,023 times; (iii) $X$ calls `GovernMental`, which now executes at the stack depth 1,024; (iv) `GovernMental` sends the victim's jackpot to the victim contract; (v) the victim contract returns False to `GovernMental` owing to the excess of call-stack's depth limit (i.e., 1,024) in the EVM, meaning that the victim contract cannot receive its jackpot; (vi) `GovernMental` is supposed to check the returned value (i.e., False in this case) and then proceed correspondingly (i.e., reverting the transaction in this case and keeping the contract state variable intact), but the malicious `GovernMental` ignores the returned False value and proceeds to reset contract `GovernMental`'s state for the next round, causing the victim's jackpot to belong to the owner of `GovernMental`. The third attack, denoted by $\mathcal{A}_7$, exploits the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$) that a malicious miner can abandon some transactions related to `GovernMental` or reorder transactions to make itself the last player (i.e., winner) in each round. The fourth attack, denoted by $\mathcal{A}_8$, exploits the *timestamp dependence* vulnerability ($\mathcal{V}_{25}$) that a malicious miner can manipulate `block.timestamp` so that its own block appears to be the last block to make itself win.

*4.1.5 HYIP Attack ($\mathcal{A}_9$).* The contract `HYIP` was another Ponzi scheme, which pays existing investors from funds contributed by new investors at the end of each day. This mechanism is implemented by the function performPayouts() highlighted in Figure 7, which contains the *DoS with unexpected revert* vulnerability ($\mathcal{V}_5$) (Line 7) [68]. The attack proceeds as follows: (i) The attacker, say Alice, writes an exploitation contract, named `Mallory`, in which the attacker invests and throws an exception in the fallback function (Line 12). (ii) When function performPayouts() is called to pay the investors, the fallback function is invoked and throws an exception, causing a reversion of the money transfer (Line 7) and thus DoS to contract `HYIP`. (iii) The attacker can blackmail `HYIP` to pay a ransom for halting its attack, by undoing the throw operation (Line 13) via function stopAttack (Line 15), which can only be done by the contract owner, Alice.

*4.1.6 Fomo3D Attacks ($\mathcal{A}_{10}$ and $\mathcal{A}_{11}$).* The contract `Fomo3D` was a popular Ponzi game in 2018, where the last participant who buys a key before the timer runs out won the jackpot. The price of keys grows with the number of buyers. When a key was sold, the countdown extends for 30 seconds. In addition to the jackpot winner, Fomo3D implemented an airdrop lottery to attract participants. For each purchase over 0.1 ETH, the buyer had a chance to be picked up for a tiny profit from the prize pool. These two incentive mechanisms can be attacked [42].

The first attack ($\mathcal{A}_{10}$) is against the airdrop mechanism. Specifically, the attack exploited the *generating randomness* vulnerability ($\mathcal{V}_{26}$). As shown in Figure 8, function airdrop() generates a random seed by performing a deterministic computation on the current block state (i.e., `block.timestamp`, `block.difficulty`, etc.) and the address of `msg.sender` (Lines 2–8). If the seed satisfies a certain condition (Line 10), then the current key buyer wins an airdrop. However, since the block information is predictable, an attacker can simply pre-compute the addresses of new contracts and brute-forces the winning seed (Line 2). The second attack ($\mathcal{A}_{11}$) is against the

```
function airdrop() private view returns(bool) {
    uint256 seed= uint256(keccak256(abi.encodePacked(
        (block.timestamp).add
        (block.difficulty).add
        ((uint256(keccak256(abi.encodePacked(block.coinbase)))) / (now)).add
        (block.gaslimit).add
        ((uint256(keccak256(abi.encodePacked(msg.sender)))) / (now)).add
        (block.number)
    )));
    if((seed-((seed/1000)*1000)) < airDropTracker_)
        return(true);
    else
        return(false);
    }
}
```

Fig. 8. A snippet source code of Fomo3D.

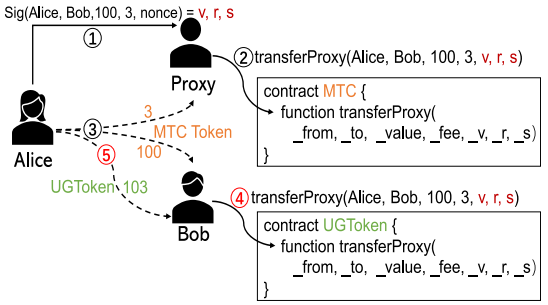


Fig. 9. Cross-contract replay attack via transferProxy().

winning mechanism [42]. Specifically, the attack exploited the *DoS with block stuffing* vulnerability ($\mathcal{V}_{31}$) and caused the attacker to win US$3M [23]. The attack proceeds as follows: When the timer of the game reaches about three minutes, the attacker buys a key and then sends multiple transactions to his own accounts with high enough *gasPrice*. Owing to the choice of miners, these transactions are first assembled into blocks. Since the maximum amount of gas consumption for a block is limited, any transactions related to `Fomo3D` are not assembled into blocks. By congesting the network until the game is over, the attacker succeeds in becoming the last player.

*4.1.7 ERC-20 Signature Replay Attack ($\mathcal{A}_{12}$).* This attack [43] exploits the *insufficient signature information* vulnerability ($\mathcal{V}_{13}$). When a user transfers ERC-20 tokens, the user must have enough Ether to pay the transaction fee, which is inconvenient when the user does not own any Ether. To alleviate the problem, the proxy-transfer method is introduced such that a user can authorize a proxy to carry out a transaction and pay the proxy some extra tokens as its service fee. As shown in Figure 9, when Alice is to transfer 100 MTC tokens to Bob, she can send a signed message off-chain to a proxy (Step 1) such that the proxy launches a transaction to transfer 100 tokens to Bob and receives 3 token from Alice for the service (Step 2). The signature is verified using function transferProxy(), which uses the Solidity function ecrecover() to identify Alice's account address that issued the signature. However, Alice's off-chain message may not provide her token contract address that should be bound to her signature. As a consequence, the signature can be accepted as valid with respect to any token contract address (e.g., MTC, UGToken, and GGoken), meaning Bob can replay the signed message to other kinds of token contracts, such as UGToken (Step 4), to obtain extra money from Alice (Step 5) [36].

*4.1.8 Rubixi Attack ($\mathcal{A}_{13}$).* The `Rubixi` contract is a Ponzi scheme containing an *erroneous constructor name* vulnerability ($\mathcal{V}_{17}$). The contract was originally named `DynamicPyramid` and later renamed by the developer to `Rubixi`. However, the contract's constructor name was not updated

accordingly, allowing anyone that calls the public function DynamicPyramid() to become the owner of the contract and steal the funds of the contract.

### 4.2 Attacks at the Data Layer

*4.2.1 ETH and ETC Cross-chain Replay Attack ($\mathcal{A}_{14}$).* Ethereum had a hard fork after the DAO attack, splitting into ETH and ETC that share the same transaction history. This means that a transaction that was validated by the ETH network could also be accepted by the ETC network when the recipient immediately rebroadcast the transaction on the ETC network, and vice versa [124]. Since both ETC and ETH networks did not implement any defense against this attack that exploited the *indistinguishable chains* vulnerability ($\mathcal{V}_{27}$), exchanges participating in both chains (e.g., Coinbase and Yunbi) lost a large amount of money [10].

*4.2.2 Under-priced DDoS Attacks ($\mathcal{A}_{15}$ and $\mathcal{A}_{16}$).* These attacks [11, 20, 82] exploit both application-layer and data-layer vulnerabilities. The first attack ($\mathcal{A}_{15}$) exploited the *under-priced opcodes* vulnerability ($\mathcal{V}_{23}$) owing to the improper gas cost of EVM's EXTCODESIZE opcode. Prior to the EIP 150 hard fork, the EXTCODESIZE opcode only charged 20 gas for reading a contract's bytecode from disk and then deriving its length. As a consequence, the attacker can repeatedly send transactions to invoke a deployed smart contract with many EXTCODESIZE opcodes to cause a 2-3x slower block creation rate [11]. The second attack ($\mathcal{A}_{16}$) exploited the *under-priced opcodes* vulnerability ($\mathcal{V}_{23}$) owing to the improper gas cost of EVM's SUICIDE opcode and the *empty account in the state trie* vulnerability ($\mathcal{V}_{28}$). The SUICIDE opcode (renamed to SELFDESTRUCT after EIP 6) is meant to remove a deployed contract from the blockchain and send the remaining balance of Ether to the account designated by the caller. When the target account does not exist, a new account is created even though no Ether may be transferred; this consumes merely 90 gas [20]. Since an existing empty account is stored in the Ethereum state trie, the attacker created 19 million new empty accounts via the SUICIDE opcode at a low gas cost, which wasted considerable disk space while increasing the synchronization and transaction processing time.

### 4.3 Attacks at the Consensus Layer

*4.3.1 Ethereum Classic (ETC) 51% Attack ($\mathcal{A}_{17}$).* In January 2019, ETC suffered from a 51% attack that exploited the *probabilistic finality* vulnerability ($\mathcal{V}_{30}$), in which the attacker carried out double-spending transactions against several exchanges, causing an estimated loss of US$1.1M [47]. Since 2018, ETC's mining hashrate has dropped significantly due to its declining price, which lowered the amount of computing resources that are required for launching a 51% attack. Moreover, cloud mining services (e.g., NiceHash) make it even easier to launch 51% attacks. The attack was disrupted when exchanges increased the number of blocks that are required for transaction confirmation and limited the participation of malicious addresses in ETC trade.

*4.3.2 Selfish-mining Attack ($\mathcal{A}_{18}$).* This attack exploits the *probabilistic finality* vulnerability ($\mathcal{V}_{30}$), the *honest mining assumption* vulnerability ($\mathcal{V}_{32}$), and the *rewards for uncle blocks* vulnerability ($\mathcal{V}_{33}$) such that miners may withhold their newly mined blocks and selectively publish some blocks to earn an unfair share of reward. A selfish miner monitors the situation on a blockchain's public branches, estimates its advantages, and reveals its private blocks accordingly. When the public branches are shorter than the selfish-miner's private branch, the honest miners will switch to latter, rendering their previous mining effort useless and making the selfish-miners receive a higher reward. Ritz et al. [161] conducted a Monte Carlo simulation to emulate the block generation process in Ethereum and quantified the impact of uncle rewards on selfish-mining. Their simulation results showed that the uncle-block reward mechanism not only lowered the threshold of computational power at which selfish-mining becomes profitable but also weakened the overall

resilience against other attacks such as *double-spending*. Niu et al. [154] developed a mathematical analysis on a selfish-mining strategy through a two-dimensional Markov process model, showing that Ethereum is more vulnerable to selfish-mining than Bitcoin. Grunspan et al. [109, 110] rigorously analyzed selfish-mining strategies in Ethereum and proved that brutal fork (i.e., keeping the fork secret as long as possible and releasing it at once) is more profitable than releasing the fork one block at a time.

*4.3.3 Bribery Attacks ($\mathcal{A}_{19}$, $\mathcal{A}_{20}$, $\mathcal{A}_{21}$, and $\mathcal{A}_{22}$).* Bribery attacks exploit the *honest mining assumption* vulnerability ($\mathcal{V}_{32}$); this assumption is vulnerable, because miners can make profit by taking bribes and changing their mining strategy to benefit the bribers. There are two kinds of bribery attacks: (i) the *in-band* bribery attack pays bribes with the cryptocurrency that is under attack; and (ii) the *out-of-band* bribery attack pays bribes with another cryptocurrency. McCorry et al. [144] presented an in-band bribery attack ($\mathcal{A}_{19}$), which offers miners an extra bonus for mining uncle blocks; as a consequence, the briber with 25% of the mining power can have full control over the blockchain. This attack also exploits the *rewards for uncle blocks* vulnerability ($\mathcal{V}_{33}$), because rewards from mining uncle blocks are directly used to subsidize bribes paid by the attacker. Winzer et al. [182] presented three in-band bribery attacks ($\mathcal{A}_{20}$), which allow the attacker to prevent state changes to an account (i.e., temporary censorship). Out-of-band bribery attacks against PoW cryptocurrencies include the Goldfinger bribery attack [144] ($\mathcal{A}_{21}$) and the no-fork and near-fork incentive attacks [120] ($\mathcal{A}_{22}$).

*4.3.4 Balance Attack ($\mathcal{A}_{23}$).* This attack exploits the *probabilistic finality* vulnerability ($\mathcal{V}_{30}$) in the presence of communication delays (or asynchronous networks). This attack was first reported in Reference [150] via a theoretic analysis and testnet demonstration. The attack entails transiently partitioning the network into multiple subgroups of similar mining power to launch the double-spending attack on a subgroup of lower mining power. This allows the attacker to initiate transactions with merchants in one subgroup, while mining blocks in another subgroup to make its subtree outweigh the subtree mined in the victim group. After transactions in the victim subgroup get confirmed, the attacker reconnects the network. Since the mining power is roughly equally distributed among the subgroups, the subtree broadcast by the attacker has a good chance to be selected as the main chain, meaning that the attacker can double-spend in the victim subgroup. This attack was later deemed only theoretically possible, because partitioning a public Ethereum network (e.g., using BGP-hijacking) may not be feasible in practice [97].

*4.3.5 Resource Exhaustion Attack ($\mathcal{A}_{24}$).* This attack exploits the *verifier's dilemma* vulnerability ($\mathcal{V}_{34}$) that miners may deviate from the protocol by not verifying resource-consuming transactions to have an advantage in the race of mining the next block [140]. This attack incurs no price to the miner that includes resource-intensive transactions in the new block it mined. The *gasLimit* mechanism can mitigate, but cannot eliminate, this attack.

*4.3.6 Incorrect Transaction Attack ($\mathcal{A}_{25}$).* This attack exploits the *honest mining assumption* vulnerability ($\mathcal{V}_{32}$) and the *verifier's dilemma* vulnerability ($\mathcal{V}_{34}$). Miners in Ethereum have a strong incentive to skip the verification of resource-consuming transactions by accepting them to gain an advantage in mining new blocks. This causes unverified transactions to be included in the blockchain, posing serious consequences.

### 4.4 Attacks at the Network Layer

*4.4.1 Account Hijacking Attack ($\mathcal{A}_{26}$).* This attack exploits the *RPC API exposure* vulnerability ($\mathcal{V}_{40}$). To sign transactions, an EOA must first decrypt its private key that is stored on the local host and encrypted with a passphrase. This can be achieved by using the unlockAccount()

API of an Ethereum client, which uses the passphrase to obtain the private key and loads it into the memory for 300 seconds (by default). The private key in memory can be accessed by any Ethereum API without authentication. This can be exploited as follows. Ethereum clients (e.g., Geth, Parity) typically use the default ports 8545 (HTTP) and 8546 (WebSocket) as the JSON-RPC interface. However, these clients neither configure those ports as local-only by default, nor adopt precautions (e.g., disabling remote calls). This allows an attacker to scan open Ethereum nodes and invoke eth_sendTransaction() API to transfer victims' money to the attacker's account. Once a victim types its passphrase to unlock its account, the eth_sendTransaction() API will be successfully executed. By the time the attack was observed in March 2018 [27], attackers had stolen around US$20M from exposed Ethereum clients.

*4.4.2 Eclipse Attacks ($\mathcal{A}_{27}$, $\mathcal{A}_{28}$, and $\mathcal{A}_{29}$).* This attack allows an attacker, who can hijack the connections of some victim nodes in the P2P network, to isolate those victim nodes from the rest of the network. Victim nodes' connections can be hijacked by connection monopolization [142], poisoning victims' routing tables [142], and falsifying neighbors [115]. First, the connection monopolization attack ($\mathcal{A}_{27}$) exploits the *unlimited nodes creation* vulnerability ($\mathcal{V}_{35}$) and the *uncapped incoming connections* vulnerability ($\mathcal{V}_{36}$). When a client reboots, it has no incoming or outgoing connections. An attacker can create plenty of node IDs in advance and initiate enough incoming connections to the victim node immediately after its reboot. A node is eclipsed when its connection slots (25 by default) are occupied by incoming connections from the attacker. Second, despite the defense (against the connection monopolization attack) that imposes an upper limit on the number of incoming TCP connections, the following attack $\mathcal{A}_{28}$ can still succeed. An attacker can exploit the *public peer selection* vulnerability ($\mathcal{V}_{37}$) to poison the victim nodes' routing tables when these tables are reboot and reset. For example, the attacker can craft fake nodes and insert them into those routing tables to make the victim nodes' outgoing TCP connections point to the fake nodes. The attacker can further occupy the victim nodes' remaining connection slots by initiating connections to the victim nodes. Third, instead of poisoning a victim's entire routing table, the *false friends attack* ($\mathcal{A}_{29}$) allows an attacker to insert a number of fake nodes into the victim's routing table to eclipse the victim node without restarting it. For example, the Geth client gets new peers by choosing nodes from its routing table or lookup-buffer. By exploiting the *fixed peer selection* vulnerability ($\mathcal{V}_{38}$), an attacker can make the victim node select the peers (from its routing table) that are under the attacker's control. The attacker can further exploit the *unlimited nodes creation* vulnerability ($\mathcal{V}_{35}$) to generate a large number of node IDs to poison the lookup-buffer. As a consequence, the victim node's outbound connections point to the nodes controlled by the attacker.

### 4.5 Further Analysis of Attack Consequences

Now, we present a taxonomy of the attack consequences mentioned above: *unauthorized code execution*, *DoS*, *unfair income*, *double-spending*, and *private key leakage*. Figure 10 highlights the relationships between these attack consequences, attacks and vulnerabilities.

Table 2 summarizes the attacks that have incurred substantial financial losses. Based on Figure 10 and Table 2, we make the following observations and insights. First, attacks against Ethereum have successfully exploited a number of vulnerabilities, especially those that reside at the application layer (i.e., $\mathcal{V}_1 \leq \mathcal{V}_i \leq \mathcal{V}_{26}$). These vulnerabilities may be inherent to the permissionless and immutability nature of the Ethereum blockchain, because the former allows anyone to access and invoke the smart contracts on the blockchain (i.e., vulnerabilities can be exploited at will) and the latter assures that the code of deployed smart contracts cannot be modified by anyone (i.e., impossible to patch their vulnerabilities). This leads to:

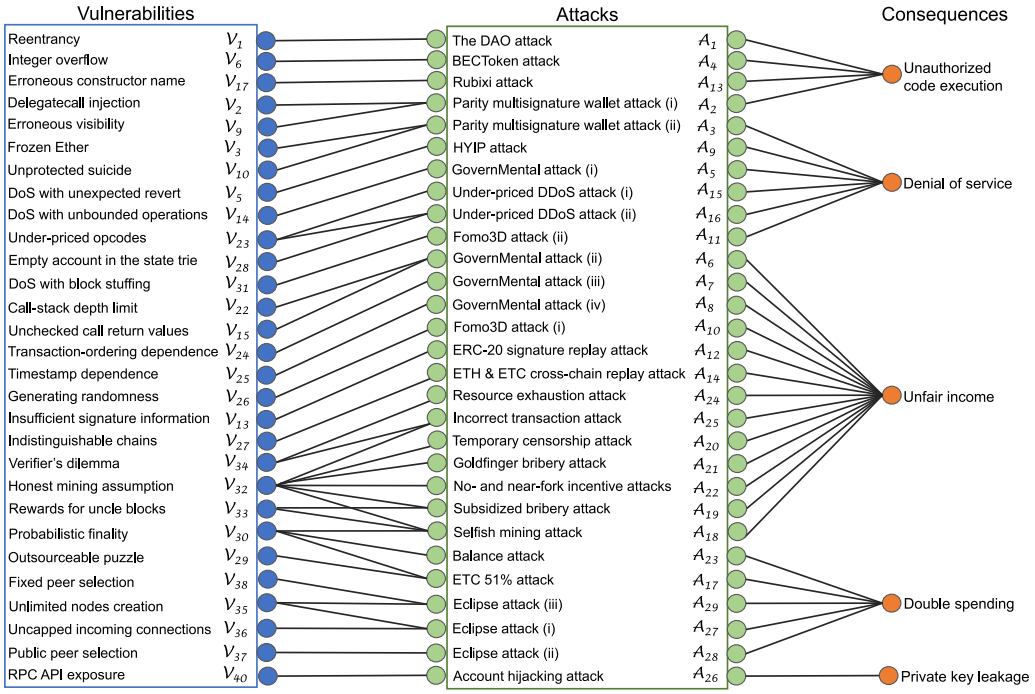


Fig. 10. The relation between vulnerabilities, attacks, and attack consequences.

Table 2. Overview of Attacks and Financial Losses Incurred by Them

| Attack targets | Vulnerability location | Real-world attacks | Attack consequences | Financial losses |
|---|---|---|---|---|
| Smart contract | Application | The DAO ($\mathcal{A}_1$) | Unauthorized code exec. | US$60M |
| | | Parity (i) ($\mathcal{A}_2$) | Unauthorized code exec. | US$31M |
| | | Parity (ii) ($\mathcal{A}_3$) | DoS | US$280M |
| Ethereum | Consensus | Fomo3D (ii) ($\mathcal{A}_{11}$) | DoS | US$3M |
| | | ETC 51% ($\mathcal{A}_{17}$) | Double spending | US$1.1M |
| | Data | ETH & ETC replay ($\mathcal{A}_{14}$) | Unfair income | US$0.5M |
| | Network | Account hijacking ($\mathcal{A}_{20}$) | Private key leakage | US$20M |

Insight 8. *Ethereum blockchain has two security barriers: permissionless (allowing attackers to exploit vulnerabilities at will) and immutability (disabling the vulnerability-patching mechanism widely used in cybersecurity).*

Second, application-layer attacks have caused larger financial losses than their counterparts at the lower layers. This is somewhat counterintuitive, because a vulnerability at a lower layer often causes a more significant damage than a vulnerability at a higher layer. This can be attributed to the fact that smart contracts operate directly on digital assets in EVM, which are isolated from the host computer (i.e., smart contracts cannot access the file system or any process running on the computer hosting the EVM). This leads to:

Insight 9. *While application-layer attacks have caused huge financial losses, the damage did not spread to the underlying hosts because of the EVM isolation.*

Third, the so-far largest financial loss, $280M, is caused by a DoS attack against the Parity wallet, which disables a library that is used by many contracts. This manifests both a new *risk* (i.e., DoS causing *direct*, rather than indirectly, economic losses) and a new *cause* (i.e., DoS caused by code-reuse). This leads to:

Insight 10. *Code-reuse in smart contracts can impose a higher risk than its counterpart in traditional systems, highlighting the importance of security auditing on widely reused smart contracts and libraries.*

## 5 DEFENSES

In this section, we describe the 51 defenses that have been proposed for securing the Ethereum ecosystem, which are denoted by $\mathcal{D}_1, \ldots, \mathcal{D}_{51}$, respectively. Unlike vulnerabilities and attacks that naturally correspond to some layer of the Ethereum architecture, defenses are by no means geared toward the layers. Therefore, we propose categorizing them into two classes: proactive defenses and reactive defenses.

### 5.1 Proactive Defenses

We categorize existing proactive defense mechanisms into the following five sub-classes based on the respective focus of these mechanisms: *contract programming language*, *contract development*, *contract analysis*, *contract and Ethereum enhancement*, and *consensus protocols*.

*5.1.1 Language-based Security.* Programming language approaches to securing smart contracts can be divided into two categories: high-level languages for developing more secure smart contracts and intermediate-level languages for facilitating contract formal analysis.

*High-level languages.* Vyper [55] ($\mathcal{D}_1$) removes a number of functionalities provided by Solidity (e.g., recursive calling, infinite loops, modifiers) and adds several new features (e.g., bounds and overflow checking) to eliminate vulnerabilities, such as the *DoS with unbounded operations* vulnerability ($\mathcal{V}_{14}$), the *erroneous visibility* vulnerability ($\mathcal{V}_9$), and the *integer overflow and underflow* vulnerability ($\mathcal{V}_6$). Bamboo [24] ($\mathcal{D}_2$) uses polymorphism to mitigate the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$), while eliminating the *reentrancy* vulnerability ($\mathcal{V}_1$) and the *DoS with unbounded operations* vulnerability ($\mathcal{V}_{14}$). Obsidian [84] ($\mathcal{D}_3$) models smart contracts as finite state machines and tracks contracts' states to eliminate the *reentrancy* vulnerability ($\mathcal{V}_1$). Flint [167] ($\mathcal{D}_4$) uses an `Asset` type to assure the atomicity of operations and introduces restrictions on callers' capabilities to protect contract functions from unauthorized access, while aiming to eliminate the *reentrancy* vulnerability ($\mathcal{V}_1$), the *erroneous visibility* vulnerability ($\mathcal{V}_9$), the *integer overflow and underflow* vulnerability ($\mathcal{V}_6$), the *unchecked call return value* ($\mathcal{V}_{15}$), and the *timestamp dependency* vulnerability ($\mathcal{V}_{25}$).

*Intermediate-level languages.* Simplicity [156] ($\mathcal{D}_5$) is an intermediate representation between high-level languages and low-level EVM. It provides formal semantics using the proof-assistant Coq [44], thus allowing formal analysis of contracts properties (e.g., safety and liveness). Scilla [169] ($\mathcal{D}_6$) distinguishes intra-contract computation from inter-contract communication to disentangle contract-specific effects from each other.

*5.1.2 Contract Development.* Since smart contracts are a new programming paradigm, these defenses can help developers in avoiding or mitigating common vulnerabilities.

*Principles and best practices.* A number of Solidity best practices were recommended in Reference [49], such as *check, update, then interact* (i.e., checking conditions first, then updating state variables, and finally interacting with other contracts); and *favor pull over push for external calls*

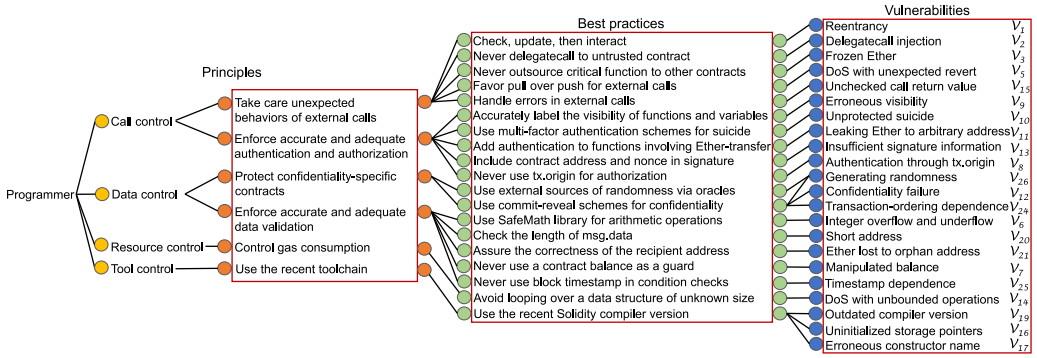


Fig. 11. Best practices and principles for guiding contract development.

(i.e., letting a recipient withdraw or "pull" the money set by the sender, rather than letting the sender directly transfer or "push" the money to the recipient). To help practitioners adopt the best practices, we systematize 19 best practices ($\mathcal{D}_7$) according to the vulnerabilities toward which they are geared, while noting that 15 of the 19 best practices were scattered in References [30, 49, 54]. As highlighted in Figure 11, our systematization leads to four principles or six specific sub-principles. Intuitively, a programmer should be conscious of four kinds of controls: *call control*, *data control*, *resource control*, and *tool control*. (i) The *call control* principle says that a programmer should secure the interactions between smarts contracts and the interactions between EOAs and smart contracts. It is further divided into two sub-principles: one coping with the callee's unexpected behaviors and the other coping with the caller's access control. (ii) The *data control* principle says that a programmer should secure the data flow of a contract. It is further divided into two sub-principles: one dealing with the protection of sensitive data and the other dealing with the prevention of malformed data from entering a smart contract. (iii) The *resource control* principle says that a programmer should mitigate the exhaustion of available gas in Ethereum. (iv) The *tool control* principle says that a programmer should use updated tools (e.g., compiler, debugger) to eliminate known vulnerabilities.

*Software engineering mechanisms.* To defend against attacks that may exploit unknown vulnerabilities, several blockchain-oriented software engineering mechanisms were introduced [49] ($\mathcal{D}_8$), such as: *rate limit*, which restricts the number of consecutive actions incurred by an EOA or restricts the amount of Ether sent by a contract within a period of time; *balance limit*, which regulates the maximum amount of funds that can be held by a contract; *speed bump*, which postpones some potentially damaging operations.

*5.1.3 Smart Contract Analysis.* These defenses aim to enhance security of smart contracts via symbolic execution, abstract interpretation, formal verification, fuzzing, and model-based vulnerability detection.

*Symbolic execution.* It works on a program' control-flow graph (CFG) and traverses all of the feasible execution paths on the graph to identify vulnerabilities [127]. This approach achieves neither soundness (i.e., zero false-negatives) nor completeness (i.e., zero false-positives), owing to the omission of some execution paths and the exploration of unreachable paths in real executions. Oyente [139] ($\mathcal{D}_9$) can detect four types of vulnerabilities—the *reentrancy* vulnerability ($\mathcal{V}_1$), the *mishandled exceptions* vulnerability ($\mathcal{V}_{15}$), the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$), and the *timestamp dependence* vulnerability ($\mathcal{V}_{25}$)—but incurs a high false-positive rate [121]. Maian [152] ($\mathcal{D}_{10}$) extends Oyente by considering multiple invocations of a contract, rather

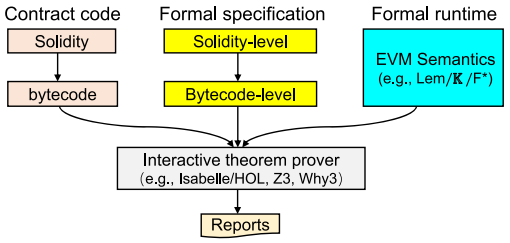


Fig. 12. Theorem proving for verifying smart contracts.

than a single invocation. It can detect three types of vulnerabilities: the *frozen Ether* vulnerability ($\mathcal{V}_3$), the *leaking Ether to arbitrary address* vulnerability ($\mathcal{V}_{11}$), and the *unprotected suicide* vulnerability ($\mathcal{V}_{10}$). Mythril [51] ($\mathcal{D}_{11}$) uses "concolic analysis," which integrates symbolic and concrete execution of a smart contract, to detect eight types of vulnerabilities, such as the *manipulated balance* vulnerability ($\mathcal{V}_7$), the *authentication through tx.origin* vulnerability ($\mathcal{V}_8$), and the *generating randomness* vulnerability ($\mathcal{V}_{26}$). The teEther tool [130] ($\mathcal{D}_{12}$) can detect vulnerabilities like the *erroneous visibility* vulnerability ($\mathcal{V}_9$) and the *erroneous constructor name* vulnerability ($\mathcal{V}_{17}$). The sCompile tool [79] ($\mathcal{D}_{13}$) reduces the number of suspicious execution paths that are not related to money-transfer. It can detect four types of vulnerabilities: the *reentrancy* vulnerability ($\mathcal{V}_1$), the *frozen Ether* vulnerability ($\mathcal{V}_3$), the *unprotected suicide* vulnerability ($\mathcal{V}_{10}$), and the *Ether lost to orphan address* vulnerability ($\mathcal{V}_{21}$). ECF [108] ($\mathcal{D}_{14}$) focuses on detecting callback-related vulnerabilities, such as the *reentrancy* vulnerability ($\mathcal{V}_1$).

*Abstract interpretation.* Abstract interpretation aims to over-approximate the semantics of a program to achieve soundness in program analysis [87]. Securify [176] ($\mathcal{D}_{15}$) defines a set of *compliance* and *violation* patterns to characterize how contract comply and violate security properties extracted from some known vulnerabilities, such as the *reentrancy* vulnerability ($\mathcal{V}_1$), the *delegatecall injection* vulnerability ($\mathcal{V}_2$), and the *frozen Ether* vulnerability ($\mathcal{V}_3$). Zeus [121] ($\mathcal{D}_{16}$) defines a set of correctness and fairness policies and then embeds them as `assert` statements into the source code of contracts. It can detect six types of vulnerabilities, such as the *reentrancy* vulnerability ($\mathcal{V}_1$) and the *unchecked call return value* vulnerability ($\mathcal{V}_{15}$). FSolidM [143] ($\mathcal{D}_{17}$) abstracts smart contracts as finite state machines and can detect the *reentrancy* vulnerability ($\mathcal{V}_1$) and the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$). MadMax [104] ($\mathcal{D}_{18}$) disassembles EVM bytecode into an intermediate representation and then leverages both data-flow analysis and context-sensitive flow analysis to detect gas-related vulnerabilities, such as the *DoS with unbounded operations* vulnerability ($\mathcal{V}_{14}$). Vandal [75] ($\mathcal{D}_{19}$) translates EVM bytecode into logic relations and use them to detect a number of vulnerabilities, such as the *reentrancy* vulnerability ($\mathcal{V}_1$), the *authentication through tx.origin* vulnerability ($\mathcal{V}_8$), the *unprotected suicide* vulnerability ($\mathcal{V}_{10}$), and the *unchecked call return value* vulnerability ($\mathcal{V}_{15}$). To facilitate the aforementioned high-level contract analysis, several reverse engineering tools have been developed to convert EVM bytecode to source or intermediate code. Porosity [171] ($\mathcal{D}_{20}$) is a decompiler for producing Solidity source code from EVM bytecode. Erays [199] ($\mathcal{D}_{21}$) lifts EVM bytecode to a high-level pseudocode by recovering the control-flow structure and transforming EVM from a stack-based machine to a register-based machine. EthIR [61] ($\mathcal{D}_{22}$) decompiles EVM bytecode to a high-level rule-based representation, which can then be fed into an automated static analyzer to infer high-level properties of the EVM bytecode.

*Formal verification.* Formal verification proves the correctness of contract implementation with respect to a specification. This approach assures completeness (i.e., no false positives). Figure 12 illustrates how to use theorem-proving to verify smart contracts at the EVM bytecode-level. Hirai

[117] ($\mathcal{D}_{23}$) took the first step toward formalizing the EVM semantics, which can be accommodated by interactive theorem provers like Isabelle/HOL [153] to prove invariants and safety properties of smart contracts. Amani et al. [63] ($\mathcal{D}_{24}$) extend the work in Reference [117] by splitting a contract into some basic blocks and then using a Hoare-style program logic to reason about semantic properties of contracts from properties of its parts. Hildenbrandt et al. [116] ($\mathcal{D}_{25}$) present a complete semantics of the EVM, referred to as KEVM, using the $\mathbb{K}$ framework [163] to achieve language-independent program verification. Park et al. [157] ($\mathcal{D}_{26}$) optimize the KEVM verifier by introducing EVM-specific abstractions and lemmas to avoid non-tractable reasoning in the underlying theorem prover. Grishchenko et al. [107] ($\mathcal{D}_{27}$) define a complete small-step semantics of EVM bytecode and formalize most of the semantics in the proof assistant F* [173]. Grishchenko et al. [105, 106] ($\mathcal{D}_{28}$) leverage the complete small-step semantics of EVM bytecode to build *EtherTrust*, which is the first sound and automated static analyzer to achieve formal security related to the reachability properties of EVM bytecode. Other early stage investigations include References [4, 70, 71].

*Fuzzing.* Fuzz testing has been used to detect vulnerabilities in smart contracts. ContractFuzzer [119] ($\mathcal{D}_{29}$) can detect five types of vulnerabilities, such as the *reentrancy* vulnerability ($\mathcal{V}_1$) and the *unchecked call return value* vulnerability ($\mathcal{V}_{15}$). It generates inputs by crawling the ABI interfaces of smart contracts to extract their function selectors and data types of each argument, and instruments EVM to log contract execution behaviors for inspection. ReGuard [138] ($\mathcal{D}_{30}$) aims to detect the *reentrancy* vulnerability ($\mathcal{V}_1$) in smart contracts by transforming smart contracts to semantically equivalent C++ program and generating random transactions via a fuzzing engine to check the execution traces of the C++ program.

*Model-based vulnerability detection.* Tann et al. [174] ($\mathcal{D}_{31}$) use sequence learning to detect three types of vulnerabilities, namely, the *frozen Ether* vulnerability ($\mathcal{V}_3$), the *leaking Ether to arbitrary address* vulnerability ($\mathcal{V}_{11}$), and the *unprotected suicide* vulnerability ($\mathcal{V}_{10}$). Tikhomirov et al. [175] ($\mathcal{D}_{32}$) propose SmartCheck to detect vulnerabilities in Solidity contracts by translating Solidity source code into an XML-based parse-tree and checks it against specific XPath patterns; they can detect 10 types of vulnerabilities, such as the *DoS with unexpected revert* vulnerability ($\mathcal{V}_5$), the *manipulated balance* vulnerability ($\mathcal{V}_7$), but incur a high false-positives rate.

*5.1.4 Contract and Ethereum Enhancement.* Kosba et al. [129] proposed a countermeasure ($\mathcal{D}_{33}$) to use cryptographic mechanisms to hide transaction data and allow contract developers to build confidentiality-preserving smart contracts. This countermeasure can defend against the attacks that exploit the *confidentiality failure* vulnerability ($\mathcal{V}_{12}$) and the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$). Zhang et al. [195] designed an authenticated data feed system ($\mathcal{D}_{34}$), dubbed Town Crier, for smart contracts that require access to external data. This system can be used to mitigate the *generating randomness* vulnerability ($\mathcal{V}_{26}$). Adler et al. [60] proposed a new defense ($\mathcal{D}_{35}$), which extends Town Crier by implementing a voting-based decentralized oracle to address the centralized point-of-failure that is inherent to Town Crier. Chen [82] proposed an adaptive gas cost mechanism ($\mathcal{D}_{36}$) to defend against DoS attacks exploiting the *under-priced opcodes* vulnerability ($\mathcal{V}_{23}$), by dynamically adjusting the cost of EVM operations according to their execution time. To harden the Ethereum network against eclipse attacks, Marcus et al. [142] proposed a countermeasure ($\mathcal{D}_{37}$) to eliminate complicated artifacts of the Kademlia protocol used by Ethereum. Henningsen et al. [115] proposed a countermeasure ($\mathcal{D}_{38}$) against the eclipse attack exploiting the *fixed peer selection* vulnerability ($\mathcal{V}_{38}$). Wang et al. [180] proposed a countermeasure ($\mathcal{D}_{39}$) to defend Ethereum clients against attacks exploiting the *RPC API exposure* vulnerability ($\mathcal{V}_{40}$).

*5.1.5 New Blockchain Protocols.* To tackle the *outsourceable puzzle* vulnerability ($\mathcal{V}_{29}$), Miller et al. [146] formalized the notion of non-outsourceable puzzles and employed Merkle-tree-based

proofs for puzzle design ($\mathcal{D}_{40}$). The basic idea underlying non-outsourceable puzzles is: If a pool operator outsources the mining task to miners, then the miners can collect the full credit while the pool operator gets nothing, which effectively disincentivizes pool operators from outsourcing their mining tasks. Zeng et al. [194] extended the work of Reference [146] by proposing a non-outsourceable puzzle ($\mathcal{D}_{41}$) that is compatible with the GHOST protocol used by Ethereum. Daian et al. [88] designed a two-stage non-outsourceable puzzle ($\mathcal{D}_{42}$) where the outer puzzle relies on the solution to the inner puzzle. Eyal et al. [1] proposed two-phase proof of work ($\mathcal{D}_{43}$) to disincentivize large mining pools, by incorporating two puzzles (instead of one) for miners to solve. Luu et al. [141] implemented a new decentralized pooled mining protocol ($\mathcal{D}_{44}$) to defend against mining centralization, by replacing the traditional mining pool operator with an Ethereum smart contract. To address the *probabilistic finality* vulnerability ($\mathcal{V}_{30}$), the Ethereum community proposed the checkpoint mechanism [77] ($\mathcal{D}_{45}$) and the safety oracle decision mechanism [149] ($\mathcal{D}_{46}$) in the upcoming PoS upgrade to ensure deterministic finality as long as at least 2/3 of the validators behave honestly. To mitigate the *verifier's dilemma* vulnerability ($\mathcal{V}_{34}$), Luu et al. [140] implemented a consensus-based protocol ($\mathcal{D}_{47}$) in Ethereum to incentivize miners to verify all transaction in each new block.

### 5.2 Reactive Defenses

Reactive defenses aim to react to potential exploitations of (unknown) vulnerabilities during the contract runtime to mitigate the damage. A runtime verification method monitors the execution traces to detect and possibly react to suspicious activities that may violate certain properties. DappGuard [86] ($\mathcal{D}_{48}$) actively monitors the incoming transactions to a smart contract and leverages the aforementioned tool Oyente [139] to decide whether or not an incoming transaction can cause a security violation and, if so, then a counter transaction can be invoked to kill the contract in question. ContractLarva [98] ($\mathcal{D}_{49}$) generates a new Solidity contract from the original contract and its specification, checks the original contract's runtime behaviors against this new contract's, and takes appropriate actions in the case of any discrepancy. Sereum [162] ($\mathcal{D}_{50}$) uses taint analysis to monitor runtime data flows during smart contract execution for detecting and preventing the *reentrancy* vulnerability ($\mathcal{V}_1$). When detecting a violation, various mechanisms ($\mathcal{D}_{51}$) have been proposed for mitigating the damage: (i) disabling the vulnerable smart contract or sensitive functionalities by using (for example) the *emergency stop* mechanism [183]; (ii) adopting a *stake-placing* mechanism to assure that any invocation, which potentially violates some properties, should pay a stake of compensation before running the contract and returns the stake back to the caller after the contract terminates normally; (iii) replacing vulnerable contracts with secure ones using the *virtual upgrade* mechanism [49], which can be realized by using a registry contract to hold the address of the latest version of a contract or by introducing a proxy contract to delegate calls to the latest version of a contract.

### 5.3 Further Analysis Based on Defense Capabilities

Now, we analyze defenses from the perspective of *defense capabilities*, meaning which defense can defend against the attacks that exploit certain vulnerabilities. Figure 13 plots the Venn diagram of the six kinds of defenses discussed above, including five kinds of proactive defenses (i.e., *alternate language*, *contract analyzer*, *security enhancement*, *contract best practices*, *blockchain protocols*) and one kind of reactive defenses (i.e., *runtime verification*). For proactive defense, we observe: (i) using *contract best practices* in the course of developing contracts can prevent or mitigate attacks that attempt to exploit 22 types of application-layer vulnerabilities; (ii) using *smart contract analyzer* can detect or mitigate attacks that attempts to exploit 18 types of application-layer vulnerabilities; (iii) using *security enhancement* can prevent or mitigate attacks that attempts to exploit 9 types of vulnerabilities, including 4 application-layer vulnerabilities and 5 network-layer ones; (iv) using

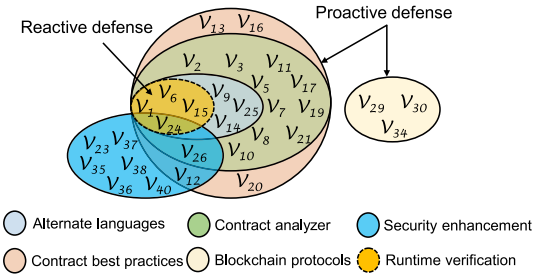


Fig. 13. Venn diagram representation of defenses against the attacks that exploit 31 vulnerabilities, while noting that the other 9 vulnerabilities either have been eliminated or have yet to be coped with. In total, proactive defenses can defend against attacks that attempt to exploit 31 types of vulnerabilities, whereas reactive defenses can defend attacks that attempt to exploit 4 types of vulnerabilities, which are also covered by proactive defenses.

better-designed contract programming language (i.e., *alternate languages* for short) can prevent or mitigate attacks that attempt to exploit 7 types of application-layer vulnerabilities; (v) using better-designed *blockchain protocols* can mitigate attacks that attempt to exploit 3 types of consensus-layer vulnerabilities. For reactive defense, we observe that using *runtime verification* can mitigate attacks that attempt to exploit 4 types of application-layer vulnerabilities.

Insight 11. *Proactive defenses can defend against attacks that attempt to exploit many vulnerabilities. In contrast, reactive defenses can defend against attacks that attempt to exploit a few vulnerabilities.*

Insight 11 reflects the state-of-the-art. Nevertheless, reactive defenses are still important, because they may be able to defend attacks attempting to exploit the vulnerabilities that survived proactive defenses.

We observe that the vulnerabilities that can be defended by *runtime verification* are subsets of the vulnerabilities that can be defended by *alternate languages*, which are in turn a subset of the vulnerabilities that can be defended by *contract analyzers*. While *contract development best practices* can mitigate most vulnerabilities at the application layer, they rely on developers' programming skills and cannot tackle the other vulnerabilities (e.g., $\mathcal{V}_{23}$).

Insight 12. *There is no single silver-bullet defense at the application layer, let alone the entire Ethereum system, highlighting the necessity of layered defense-in-depth.*

We further observe that alternate programming languages can prevent the attacks that attempt to exploit seven types of vulnerabilities, among which four (i.e., $\mathcal{V}_1$, $\mathcal{V}_6$, $\mathcal{V}_9$, $\mathcal{V}_{14}$) are caused by smart contract programming, one (i.e., $\mathcal{V}_{15}$) is caused by Solidity language, and the other two (i.e., $\mathcal{V}_{24}$, $\mathcal{V}_{25}$) are caused by Ethereum blockchain. This leads to:

Insight 13. *A better language can make smart contracts more secure and achieve a higher fault-tolerance.*

### 5.4 Further Analysis Based on Defense Investment

Now, we present an analysis of defenses from the perspective of *defense investment*, meaning how much effort has been invested in designing defense against attacks that exploit a certain vulnerability. However, we note that some defenses are *not* geared toward any specific vulnerabilities or attacks; for example, software engineering mechanisms (i.e., $\mathcal{D}_8$) are neither geared toward any specific vulnerability nor geared toward any specific attack.

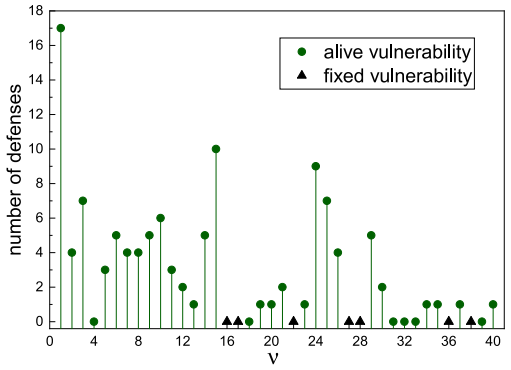


Fig. 14. The number of defenses with respect to each vulnerability.

Figure 14 plots the summary of the number of defense mechanisms with respect to individual vulnerabilities. Table 1 of the Appendix highlights which vulnerabilities may be protected by which defenses. For example, Figure 14 shows that $\mathcal{V}_1$ can be defended by 17 defenses, which are highlighted in Table 1. We observe that the most extensively investigated vulnerability is the *reentrancy* vulnerability ($\mathcal{V}_1$), which can be mitigated by 17 defense mechanisms. Other vulnerabilities that have been substantially investigated include: the *unchecked call return value* vulnerability ($\mathcal{V}_{15}$) and the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$), which can be defended by 10 and 9 defense mechanisms, respectively; the *frozen Ether* vulnerability ($\mathcal{V}_3$) and the *timestamp dependence* vulnerability ($\mathcal{V}_{25}$) can be defended by 7 defense mechanisms. It appears that the vulnerabilities that have been thoroughly investigated are (i) the ones that have caused large financial losses, (ii) the ones that are inherent to the design of the Solidity language, and (iii) the ones that are inherent to the profit-making factor for assembling blocks. This is so because the *reentrancy* vulnerability ($\mathcal{V}_1$) has caused the loss of US$60M in the DAO attack (i.e., attack $\mathcal{A}_1$ in Figure 10), the *frozen Ether* vulnerability ($\mathcal{V}_3$) has caused the loss of US$280M in the Parity wallet attack (i.e., attack $\mathcal{A}_3$), the *unchecked call return value* vulnerability ($\mathcal{V}_{15}$) is inherent to the exception handling mechanism in the Solidity language, the *transaction-ordering dependence* vulnerability ($\mathcal{V}_{24}$) is inherent to the unpredictable nature of the Ethereum blockchain, and the *timestamp dependence* vulnerability ($\mathcal{V}_{25}$) is inherent to the manipulable block information of the Ethereum blockchain. However, we observe 14 vulnerabilities that have zero or one defense mechanism. Most of these vulnerabilities are caused by the Ethereum design and implementation. This leads to:

Insight 14. *There is a large discrepancy between the efforts invested to defend against different attacks.*

Insight 14 is interesting, because it seems that the defense effort has been driven by the consequential financial loss incurred by the exploitation of certain vulnerabilities. This prioritization strategy is *not* adequate, because it suggests in a sense that the defender is always chasing behind the attacker, who detects and exploits an vulnerability that then becomes known to the defender.

## 6 ETHEREUM ONGOING DEVELOPMENT AND COMPARISON WITH OTHER BLOCKCHAINS

### 6.1 Upcoming PoS Upgrade Toward Ethereum 2.0

Since PoW-based consensus is not environment-friendly, Ethereum plans to upgrade to PoS-based consensus or Ethereum 2.0 [48]. At the time of writing, two PoS proposals are under consideration: Casper the Friendly Finality Gadget (Casper FFG) [77] and Casper Correct-by-Construction (Casper CBC) [193]. Casper FFG is initially introduced to run on top of the current PoW-based

main chain, but is later adjusted to run on top of a separate PoS-based chain, dubbed the beacon chain, which runs in parallel to the PoW-based main chain. Casper FFG leverages the checkpoint mechanism to attain finality, meaning that the blockchain is immutable up to the last *finalized* checkpoint. A checkpoint is set for every $m$ (e.g., $m = 100$) blocks and is considered *finalized* when the checkpoint itself and its subsequent checkpoint both receive votes from at least 2/3 of the voters. In contrast, Casper CBC is a framework to create a family of "correct-by-construction" consensus protocols based on rigorous mathematical models. It intends to achieve asynchronous safety in the presence of Byzantine faults. It uses a decision mechanism called *safety oracle*, which allows validators to reach decisions on the finalized blocks while assuming that 2/3 of the validators are honest [149].

Although PoS has advantages over PoW, existing PoS protocols [69, 76, 90, 123, 128, 145] are vulnerable to one or more of the following attacks. (i) The *nothing-at-stake* attack [132], where the attacker generates multiple conflicting blocks to maximize its benefit without risking its stake. This attack is possible because generating blocks in PoS incurs essentially no cost. (ii) The *long-range* attack [101, 132], where the attacker generates a new branch starting from an earlier block to overtake the main chain. This attack is possible when the attacker controls majority of the stake at some past long-range block. There are three kinds of long-range attacks: *simple attack*, *posterior corruption attack*, and *stake bleeding attack* [91]. (iii) The *grinding* attack [123], where the attacker manipulates the leader election process to increase its chance of being elected to generate blocks. This attack is possible when a PoS protocol leverages data in the blockchain itself (e.g, previous block hash) to generate randomness for electing the next block proposer.

The details of Casper CBC has not been released, because it is still under investigation. In what follows, we only discuss how Casper FFG defends against the attacks mentioned above. (i) To cope with the *nothing-at-stake* attack, Casper FFG introduces the *slashing* mechanism to heavily penalize the misbehaving validators that generate conflicting blocks, while rewarding validators who provide evidence for the misbehavior. This mechanism can deter some misbehaving validators, but cannot prevent targeted attacks (e.g., double-spending) where the attacker can benefit from forking the blockchain (after paying the penalty). (ii) To defend against the *long-range* attack, Casper FFG modifies the fork choice rule such that the blockchain up to the most recently finalized checkpoint will never be reverted. A freezing mechanism is also used to prevent malicious coalition of validators, by locking down validators' deposits for a long period of time before allowing them to take their deposits back. This defense cannot eliminate the long-range attack, because block reorganisation can still happen [91]. (iii) To create non-exploitable randomness, Ethereum 2.0 is considering to combine the RANDAO protocol with VDFs to randomly select block-proposers on the beacon chain. However, the current VDFs are complex and are not post-quantum secure [72, 73, 181]. Note that the *grinding* attack is not applicable to Casper FFG, because this protocol is a finality overlay atop the beacon chain of Ethereum 2.0, providing no leader election function.

### 6.2 Comparing Ethereum with Other Blockchains Supporting Smart Contracts

Perhaps inspired by the success of Ethereum, there have been many blockchain platforms that support smart contracts [31]. Among them, EOS [33] and Hyperledger Fabric (where smart contracts are also called *chaincode*) [64] are widely used. We observe that the taxonomy of root causes of Ethereum smart contract vulnerabilities (Section 3) can be equally applied to EOS and Hyperledger Fabric (and other platforms supporting smart contracts). This allows us to compare these three representative platforms in the following aspects: vulnerabilities caused by *smart contract programming*, vulnerabilities caused by *programming languages and tool chain*, and vulnerabilities caused by *blockchain platform design and implementation*. Since there are very few

Table 3. Comparing Smart Contract Vulnerabilities in Three Representative Blockchain Platforms

| Platform \ Cause | Smart contract programming | Programming languages and tool chain | Blockchain platform design and implementation |
|---|---|---|---|
| Ethereum | $\mathcal{V}_1, \ldots, \mathcal{V}_{14}$ (14 in total) | $\mathcal{V}_{15}, \ldots, \mathcal{V}_{19}$ (5 in total) | $\mathcal{V}_{20}, \ldots, \mathcal{V}_{26}$ (7 in total) |
| EOS | Numerical overflow [46], Authorization check [46], Apply check [46], Transfer error prompt [46, 159] | Buffer overflows [28], Dangling pointer references [28] | Roll back [46], Generating randomness [46], Transaction congestion [45] |
| Hyperledger Fabric | Unchecked input [29], Unhandled errors [29] | Global state [29], Field declarations [29], Blacklisted imports [29], Goroutines [29], Map range iterations [29], Reified object addresses [192], System timestamp [178], Generating random number [178] | Read your write [29], Range query risk [29] |

papers [118, 159, 178, 192] studying EOS and Fabric smart contract vulnerabilities, we searched Internet blogs and forums [28, 29, 45, 46] to collect information on their vulnerabilities.

Table 3 highlights the comparison, while considering C++ and Go as the respective programming language for EOS and Hyperledger Fabric, because they are widely used. We observe the following. (i) Ethereum has much more types of vulnerabilities (than EOS and Hyperledger Fabric) and most vulnerabilities are specific to the Solidity language, which might have imposed a new learning curve and caused many new vulnerabilities. (ii) Hyperledger Fabric has many types of vulnerabilities that can be attributed to the non-deterministic nature of some Go instructions (i.e., an operation may produce different results when executed at different peers). (iii) Each of the three platforms has some vulnerabilities that are specific to its architecture. This leads to:

Insight 15. *Blockchains using different programming languages and architectures have different vulnerabilities.*

## 7 FUTURE RESEARCH DIRECTIONS

**Eliminating known Ethereum vulnerabilities**. There are 13 Ethereum vulnerabilities that are largely unaddressed. First, 2 vulnerabilities at the application layer, *upgradable contract* ($\mathcal{V}_4$) and *type casts* ($\mathcal{V}_{18}$), needed to be addressed possibly by using new smart contract design pattern and type system. Second, 8 vulnerabilities inherent to the Ethereum blockchain design need to be addressed, including *under-priced opcodes* ($\mathcal{V}_{23}$), *generating randomness* ($\mathcal{V}_{26}$), *outsourceable puzzle* ($\mathcal{V}_{29}$), *probabilistic finality* ($\mathcal{V}_{30}$), *DoS with block stuffing* ($\mathcal{V}_{31}$), *honest mining assumption* ($\mathcal{V}_{32}$), *rewards for uncle blocks* ($\mathcal{V}_{33}$) and *verifier's dilemma* ($\mathcal{V}_{34}$). For these purposes, we need to design new gas cost mechanisms, fair randomness mechanisms, consensus mechanisms, and incentive mechanisms. Third, 3 vulnerabilities inherent to the implementation of Ethereum clients need to be addressed, including *unlimited nodes creation* ($\mathcal{V}_{35}$), *public peer selection* ($\mathcal{V}_{37}$) and *sole block synchronization* ($\mathcal{V}_{39}$). For these purposes, we need to design new P2P network protocols.

**Developing Ethereum test tools and environments**. For test tools, Table 1 highlights that no detector can detect all of the known Ethereum application-layer vulnerabilities, as the currently best detector, SmartCheck ($D_{32}$), can only detect 10 (out of the 26) types of vulnerabilities at

the price of high false-positives. This highlights the importance to develop better smart contract vulnerability detectors. For this purpose, deep learning might be useful as shown in References [135, 136]. Moreover, it is important to monitor smart contracts runtime behavior for detecting attacks, because vulnerability detectors cannot be perfect; this runtime verification approach is largely unexplored. For test environment, EVM currently only supports domain-specific languages (e.g., Solidity) that have imposed a new learning curve and imposed new vulnerabilities. Enhance EVM to support general-purpose programming languages might help avoid many vulnerabilities. Moreover, it is imperative to develop an Ethereum testbed for adequately testing smart contracts before deploying them in the real world. This lack of testbed has caused the deployment of smart contracts that are not well tested. To evaluate designs and implementations of Ethereum consensus protocols and P2P network protocols, a flexible Ethereum network emulator is very valuable.

**Formalizing, analyzing and quantifying Ethereum security**. First, there is a urgent need to understand the desirable security properties of Ethereum. There are only informal discussions on its security properties [62, 112, 113, 196], representing a very preliminary first step. Second, there is a urgent need to develop principled and rigorous methodologies to analyze that the desirable properties are indeed satisfied. Third, there is a urgent need to quantify Ethereum security (or risk), because attacks against Ethereum are likely inevitable. This calls for security metrics and analysis methodologies. For this purpose, one may adopt or adapt existing security metrics (e.g., [81, 83, 147, 151, 155, 158, 160]) and the general security quantification methodology known as *cybersecurity dynamics* [111, 134, 137, 187–191, 197, 198].

## 8 CONCLUSION

We have presented a systematic survey on the security of the Ethereum system, including its application, data, consensus, and network layers. The survey considered three perspectives, namely, vulnerabilities, attacks, and defenses, while correlating them. We discussed not only the locations of the vulnerabilities but also their root causes. We systematized the attacks against, and the defenses for, the Ethereum system. We further systematized the best practices proposed by industry into a small number of guiding principles, which might be easier to adopt by practitioners. We provide insights into the state-of-the-art and into future research directions.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments that guided us in improving the article. Approved for public release; distribution unlimited: 88ABW-2019-3890; dated 12 August 2019.

## REFERENCES

[1] Ittay Eyal and Emin Gün Sirer. 2014. How to disincentivize large Bitcoin mining pools. Retrieved from http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/.

[2] Fabian Vogelsteller and Vitalik Buterin. 2015. ERC-20 Token Standard|Ethereum Improvement Proposals. Retrieved from https://eips.ethereum.org/EIPS/eip-20.

[3] Least Authority. 2015. Ethereum Analysis: Gas Economics and Proof of Work. Retrieved from https://github.com/LeastAuthority/ethereum-analyses.

[4] Ethereum Community Forum. 2015. Formal Verification for Solidity Contracts. Retrieved from https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.

[5] Phil Daian. 2016. Analysis of the DAO exploit. Retrieved from http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[6] Vitalik Buterin. 2016. EIP-150, gas cost changes for IO-heavy operations. Retrieved from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md.

[7] Vitalik Buterin. 2016. EIP-155, simple replay attack protection. Retrieved from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md.

[8] Gavin Wood. 2016. EIP-161, state trie clearing. Retrieved from https://github.com/ethereum/EIPs/blob/master/EIPS/eip-161.md.
[9] Joris Bontje. 2016. How can I securely generate a random number in my smart contract? Retrieved from https://ethereum.stackexchange.com/questions/191/how-can-i-securely-generate-a-random-number-in-my-smart-contract.
[10] Alyssa Hertig. 2016. Rise of Replay Attacks Intensifies Ethereum Divide—CoinDesk. Retrieved from https://www.coindesk.com/rise-replay-attacks-ethereum-divide.
[11] Vitalik Buterin. 2016. Transaction spam attack: Next Steps. Retrieved from https://blog.ethereum.org/2016/09/22/transaction-spam-attack-next-steps/.
[12] Peter Vessenes. 2016. Tx.Origin And Ethereum Oh My! Retrieved from https://vessenes.com/tx-origin-and-ethereum-oh-my/.
[13] Matt Suiche. 2017. The $280M Ethereum's Parity bug—Comae Technologies. Retrieved from https://blog.comae.io/the-280m-ethereums-bug-f28e5de43513.
[14] Nooku. 2017. Exploit with ERC20 token transactions from exchanges. Retrieved from https://www.reddit.com/r/ethereum/comments/63s917/worrysome_bug_exploit_with_erc20_token/dfwmhc3/.
[15] Ethererik. 2017. GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas. Retrieved from https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/.
[16] Haseeb Qureshi. 2017. A hacker stole $31M of Ether—How it happened, and what it means for Ethereum. Retrieved from https://medium.freecodecamp.org/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce.
[17] Paweł Bylica. 2017. How to Find $10M Just by Reading the Blockchain. Retrieved from https://medium.com/golem-project/how-to-find-10m-by-just-reading-blockchain-6ae9d39fcd95.
[18] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2017. An In-Depth Look at the Parity Multisig Bug. Retrieved from http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.
[19] Santiago Palladino. 2017. The Parity Wallet Hack Explained. Retrieved from https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.
[20] Vbuterin. 2017. A state clearing FAQ. Retrieved from https://www.reddit.com/r/ethereum/comments/5es5g4/a_state_clearing_faq/?st=iw2e1mwo&sh=fa7768&depth=1.
[21] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen. 2020. A survey on the security of blockchain systems. *Future Gen. Comput. Syst.* 107 (2020), 841–853.
[22] Lorenz Breidenbach, Phil Daian, Ari Juels, and Florian Tramèr. 2017. To Sink Frontrunners, Send in the Submarines. Retrieved from http://hackingdistributed.com/2017/08/28/submarine-sends/.
[23] Crypto Panda. 2018. The $3 Million Winner of Fomo3D Is Still Playing to Win—Longhash. Retrieved from https://www.longhash.com/news/the-3-million-winner-of-fomo3d-is-still-playing-to-win.
[24] Cornell Blockchain. 2018. Bamboo. Retrieved from https://github.com/pirapira/bamboo.
[25] Common Vulnerabilities and Exposures. 2018. BatchOverflow. Retrieved from http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10299.
[26] Louis Poinsignon. 2018. BGP leaks and cryptocurrencies. Retrieved from https://blog.cloudflare.com/bgp-leaks-and-crypto-currencies/.
[27] SlowMist. 2018. Billions of Tokens Theft Case cause by ETH Ecological Defects. Retrieved from https://mp.weixin.qq.com/s/ia9nBhmqVEXiiQdFrjzmyg.
[28] Mihail Sotnichek. 2018. EOS Smart Contract Vulnerabilities in Detail. Retrieved from https://www.apriorit.com/dev-blog/553-eos-smart-contract-vulnerability.
[29] ChainSecurity AG. 2018. ChainSecurity Chaincode Scanner. Retrieved from https://chaincode.chainsecurity.com/.
[30] Adrian Manning. 2018. Comprehensive list of known attack vectors and common anti-patterns. Retrieved from https://github.com/sigp/solidity-security-blog.
[31] Vaibhav Saini. 2018. ContractPedia: An Encyclopedia of 40+ Smart Contract Platforms. Retrieved from https://hackernoon.com/contractpedia-an-encyclopedia-of-40-smart-contract-platforms-4867f66da1e5.
[32] Common Vulnerabilities and Exposures. 2018. CVE-2018-10299. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-10299.
[33] Block.one. 2018. EOS.IO Technical White Paper v2. Retrieved from https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.
[34] Georgios Konstantopoulos. 2018. How to Secure Your Smart Contracts: 6 Solidity Vulnerabilities and how to avoid them (Part 2). Retrieved from https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-2-730db0aa4834.
[35] Arseny Reutov. 2018. Predicting Random Numbers in Ethereum Smart Contracts. Retrieved from https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620.

[36] Zhenxuan Bai. 2018. Replay Attacks on Ethereum Smart Contracts. Retrieved from https://github.com/nkbai/defcon26/tree/master/docs.
[37] OpenZeppelin. 2018. SafeMath. Retrieved from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol.
[38] Bernhard Mueller. 2018. Safety tips. Retrieved from https://github.com/ethereum/wiki/wiki/Safety#favor-pull-over-push-for-external-calls.
[39] Ethereum community. 2018. Solidity 0.5.0 documentation. Retrieved from https://solidity.readthedocs.io/en/v0.5.0/050-breaking-changes.html.
[40] Ethereum community. 2018. Solidity Version 0.4.22. Retrieved from https://github.com/ethereum/solidity/releases/tag/v0.4.22.
[41] Stefan Beyer. 2018. Storage Allocation Exploits in Ethereum Smart Contracts. Retrieved from https://medium.com/cryptronics/storage-allocation-exploits-in-ethereum-smart-contracts-16c2aa312743.
[42] Martin Derka. 2018. What We Learned from Fomo3D. Retrieved from https://medium.com/@martinderka.
[43] Zhenxuan Bai, Yuwei Zheng, Senhua Wang, and Kunzhe Chai. 2018. You may have paid more than you imagine. Retrieved from https://www.defcon.org/html/defcon-26/dc-26-speakers.html#Bai2.
[44] The Coq development team. 2019. The Coq Proof Assistant. Retrieved from https://coq.inria.fr/.
[45] SlowMist. 2019. EOS DApp hack events. Retrieved from https://hacked.slowmist.io/en/?c=EOS%20DApp.
[46] SlowMist. 2019. EOS smart contract development security best practices. Retrieved from https://github.com/slowmist/eos-smart-contract-security-best-practices.
[47] Alex Lielacher. 2019. ETC 51 % attack. Retrieved from https://bravenewcoin.com/insights/etc-51-attack-what-happened-and-how-it-was-stopped.
[48] Ethereum community. 2019. Ethereum 2.0 specifications. Retrieved from https://github.com/ethereum/eth2.0-specs.
[49] ConsenSys Diligence. 2019. Ethereum Smart Contract Best Practices. Retrieved from https://consensys.github.io/smart-contract-best-practices/.
[50] Felix Lange, Guillaume Ballet, and Antoine Toulme. 2019. Ethereum Wire Protocol (ETH). Retrieved from https://github.com/ethereum/devp2p/blob/master/caps/eth.md.
[51] MythX development team. 2019. Mythril. Retrieved from https://github.com/ConsenSys/mythril.
[52] Franz Volland and Florian Blum. 2019. Oracle. Retrieved from https://github.com/fravoll/solidity-patterns/blob/master/docs/oracle.md.
[53] Yaning Zhang and Youcai Qian. 2019. RANDAO: A DAO working as RNG of Ethereum. Retrieved from https://github.com/randao/randao.
[54] MythX development team. 2019. Smart Contract Weakness Classification and Test Cases. Retrieved from https://smartcontractsecurity.github.io/SWC-registry/.
[55] Vyper development team. 2019. Vyper documentation. Retrieved from https://vyper.readthedocs.io/en/latest/?badge=latest#.
[56] Etherscan development team. 2020. Ethereum (ETH) Blockchain Explorer. Retrieved from https://etherscan.io/.
[57] OpenEthereum. 2020. Fast and feature-rich multi-network Ethereum client. Retrieved from https://github.com/paritytech/parity-ethereum.
[58] The go-ethereum authors. 2020. Official Go implementation of the Ethereum protocol. Retrieved from https://github.com/ethereum/go-ethereum.
[59] State of The DApps development team. 2020. State of the DApps—DApp Statistics. Retrieved from https://www.stateofthedapps.com/stats.
[60] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania. 2018. Astraea: A decentralized blockchain oracle. *arXiv:1808.00528*.
[61] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey. 2018. EthIR: A framework for high-level analysis of Ethereum bytecode. *arXiv:1805.07208*.
[62] R. Almadhoun, M. Kadadha, M. Alhemeiri, M. Alshehhi, and K. Salah. 2018. A user authentication scheme of iot devices using blockchain-enabled fog nodes. In *Proceedings of the IEEE/ACS AICCSA*. IEEE, 1–8.
[63] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the ACM SIGPLAN CPP*. ACM, 66–77.
[64] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, and Y. Manevich. 2018. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the EuroSys*. 30.
[65] N. Atzei, M. Bartoletti, and T. Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Proceedings of the POST*. 164–186.
[66] Arati Baliga. 2017. Understanding blockchain consensus models. In *Persistent*, Vol. 4. 1–14.
[67] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. 2017. Consensus in the age of blockchains. *CoRR* abs/1711.03936.

[68] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia. 2017. Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. *arXiv:1703.03779*.
[69] I. Bentov, R. Pass, and E. Shi. 2016. Snow white: Provably secure proofs of stake. *IACR ePrint Arch.* 2016 (2016), 919.
[70] K. Bhargavan, A. Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Pinote, N. Swamy et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the ACM PLAS*. 91–96.
[71] F. Bobot, J. C. Filliâtre, C. Marché, and A. Paskevich. 2011. Why3: Shepherd your herd of provers. *First International Workshop on Intermediate Verification Languages*, pp. 53–64.
[72] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Proceedings of the CRYPTO*. Springer, 757–788.
[73] D. Boneh, B. Bünz, and B. Fisch. 2018. A survey of two verifiable delay functions. *IACR ePrint Arch.* 2018 (2018), 712.
[74] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. 2015. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Proceedings of the IEEE SP*. 104–121.
[75] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz. 2018. Vandal: A scalable security analysis framework for smart contracts. *arXiv:1809.03981*.
[76] Vitalik Buterin. 2014. Slasher: A punitive proof-of-stake algorithm. Ethereum Blog. Retrieved from https://blog.ethereum. org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm.
[77] Vitalik Buterin and Virgil Griffith. 2017. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*.
[78] Christian C. and Marko V.2017. Blockchain consensus protocols in the wild. *CoRR* abs/1707.01873.
[79] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang. 2018. sCompile: Critical path identification and analysis for smart contracts. *arXiv:1808.00624*.
[80] D. Chaum. 1982. Blind signatures for untraceable payments. In *Proceedings of the CRYPTO*.199–203.
[81] H. Chen, J. Cho, and S. Xu. 2018. Quantifying the security effectiveness of firewalls and DMZs. In *Proceedings of the HoTSoS*. 9:1–9:11.
[82] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. Au, and X. Zhang. 2017. An adaptive gas cost mechanism for ethereum to defend against under-priced DoS attacks. In *Proceedings of the ISPEC*. Springer, 3–24.
[83] Jin-Hee Cho, Shouhuai Xu, Patrick M. Hurley, Matthew Mackay, Trevor Benjamin, and Mark Beaumont. 2019. STRAM: Measuring the trustworthiness of computer-based systems. *ACM Comput. Surv.* 51, 6 (2019), 128:1–128:47.
[84] Michael Coblenz. 2017. Obsidian: A safer blockchain programming language. In *Proceedings of the ICSE*. 97–99.
[85] M. Conti, E. Kumar, C. Lal, and S. Ruj. 2018. A survey on security and privacy issues of bitcoin. *IEEE Communications Surveys Tutorials* 20, 4 (2018), 3416–3452.
[86] T. Cook, A. Latham, and J. Lee. 2017. Dappguard: Active monitoring and defense for solidity smart contracts. Retrieved from https://pdfs.semanticscholar.org/7438/ffd4c3b45a6d239815df377a453adfa890fa.pdf.
[87] P. Cousot and R. Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the PoPL*. 238–252.
[88] P. Daian, I. Eyal, A. Juels, and E. Sirer. 2017. Piecework: Generalized outsourcing control for proofs of work. In *Proceedings of the FC*. 182–190.
[89] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. 2019. Flash Boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv:1904.05234*.
[90] B. David, P. Gaži, A. Kiayias, and A. Russell. 2018. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Proceedings of the EUROCRYPT*. Springer, 66–98.
[91] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis. 2019. A survey on long-range attacks for proof of stake protocols. *IEEE Access* 7 (2019), 28712–28725.
[92] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. 2016. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Proceedings of the FinancialCRYPTO*. 79–94.
[93] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons. 2018. Smart contracts vulnerabilities: A call for blockchain software engineering? In *Proceedings of the IEEE IWBOSE*. 19–25.
[94] Monika Di Angelo and Gernot Salzer. 2019. A survey of tools for analyzing ethereum smart contracts. In *Proceedings of the DAPPCON*.
[95] Cynthia Dwork and Moni Naor. 1992. Pricing via processing or combatting junk mail. In *Proceedings of the CRYPTO*. 139–147.
[96] Paul Dworzanski. A note on committee random number generation, commit-reveal, and last-revealer attacks. Retrieved from http://paul.oemm.org/commit_reveal_subcommittees.pdf.
[97] P. Ekparinya, V. Gramoli, and G. Jourjon. 2018. Impact of man-in-the-middle attacks on ethereum. In *Proceedings of the IEEE SRDS*. 11–20.
[98] Joshua Ellul and Gordon J Pace. 2018. Runtime verification of ethereum smart contracts. In *Proceedings of the IEEE EDCC*. 158–163.

[99] Ittay Eyal and Emin Gün Sirer. 2018. Majority is not enough: Bitcoin mining is vulnerable. *Commun. ACM* 61, 7 (2018), 95–102.
[100] M. Fischer, N. Lynch, and M. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM* 32, 2, 374–382.
[101] P. Gaži, A. Kiayias, and A. Russell. 2018. Stake-bleeding attacks on proof-of-stake blockchains. In *Proceedings of the CVCBT*. 85–92.
[102] A. Gervais, G. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. 2016. On the security and performance of proof of work blockchains. In *Proceedings of the ACM CCS*. 3–16.
[103] Vincent Gramoli. 2020. From blockchain consensus back to byzantine consensus. *Future Gen. Comput. Syst.* 107 (2020), 760–769.
[104] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. In *Proceedings of the OOPSLA*. 116.
[105] I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. *EtherTrust: Sound Static Analysis of Ethereum Bytecode*. Technical Report. Retrieved from https://pdfs.semanticscholar.org/26c2/b7e7479336d44891aadda6b5eaae2ca2ee91.pdf.
[106] I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. Foundations and tools for the static analysis of ethereum smart contracts. In *Proceedings of the ICCAV*. Springer, 51–78.
[107] I. Grishchenko, M. Maffei, and C. Schneidewind. 2018. A semantic framework for the security analysis of ethereum smart contracts. In *Proceedings of the POST*. Springer, 243–269.
[108] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *In Proceedings of the PoPL*. 48.
[109] C. Grunspan and R. Pérez-Marco. 2019. Selfish mining and Dyck words in Bitcoin and Ethereum networks. *arXiv:1904.07675*.
[110] Cyril Grunspan and Ricardo Pérez-Marco. 2019. Selfish mining in ethereum. *arXiv:1904.13330*.
[111] Y. Han, W. Lu, and S. Xu. 2014. Characterizing the power of moving target defense via cyber epidemic dynamics. In *Proceedings of the HotSoS'14*, Vol. 10. 1–12.
[112] D. Harz and W. Knottenbelt. 2018. Towards safer smart contracts: A survey of languages and verification methods. *arXiv:1809.09805*.
[113] H. Hasan and K. Salah. 2018. Proof of delivery of digital assets using blockchain and smart contracts. *IEEE Access* 6, 65439–65448.
[114] H. Hasan and K. Salah. 2019. Combating deepfake videos using blockchain and smart contracts. *IEEE Access* 7, 41596–41606.
[115] S. Henningsen, D. Teunis, M. Florian, and B. Scheuermann. 2019. Eclipsing ethereum peers with false friends. In *Proceedings of the EuroS&P*. 300–309.
[116] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, and A. Stefanescu. 2018. KEVM: A complete formal semantics of the ethereum virtual machine. In *Proceedings of the CSF*. 204–217.
[117] Yoichi Hirai. 2017. Defining the ethereum virtual machine for interactive theorem provers. In *Proceedings of the FinancialCRYPTO*. 520–535.
[118] Y. Huang, Y. Bian, R. Li, J. Zhao, and P. Shi. 2019. Smart contract security: A software lifecycle perspective. *IEEE Access* 7, 150184–150202.
[119] B. Jiang, Y. Liu, and W. Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the ASE*. 259–269.
[120] A. Judmayer, N. Stifter, A. Zamyatin, I. Tsabary, I. Eyal, P. Gazi, S. Meiklejohn, and E. Weippl. 2019. *Pay-To-Win: Incentive Attacks on Proof-of-Work Cryptocurrencies*. Technical Report. Cryptology ePrint Archive, Report 2019/775.
[121] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing safety of smart contracts. In *Proceedings of theNDSS*.
[122] M. Khan and K. Salah. 2018. IoT security: Review, blockchain solutions, and open challenges. *Future Gen. Comput. Syst.* 82, 395–411.
[123] A. Kiayias, A. Russell, B. David, and R. Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Proceedings of the CRYPTO*. 357–388.
[124] L. Kiffer, D. Levin, and A. Mislove. 2017. Stick a fork in it: Analyzing the Ethereum network partition. In *Proceedings of the ACM HotNets*. 94–100.
[125] Simon Kim. 2017. *Measuring Ethereum's Peer-to-peer Network*. Ph.D. Dissertation.
[126] S. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey. 2018. Measuring ethereum network peers. In *Proceedings of the ACM IMC*. 91–104.
[127] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.
[128] Sunny King and Scott Nadal. 2012. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *Self-published Paper*. Retrieved from https://www.chainwhy.com/upload/default/20180619/126a057fef926dc286accb372da46955.pdf.

[129] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Proceedings of the IEEE SP*. 839–858.
[130] J. Krupp and C. Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *Proceedings of the UsenixSecurity*. 1317–1333.
[131] Ao Li and Fan Long. 2018. Detecting standard violation errors in smart contracts. *arXiv:1812.07702*.
[132] W. Li, S. Andreina, J. Bohli, and G. Karame. 2017. Securing proof-of-stake blockchain protocols. In *Proceedings of the DPM CBT*. 297–315.
[133] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen. 2017. A survey on the security of blockchain systems. *Future Gen. Comput. Syst.* 107 (2020), 841–853.
[134] X. Li, P. Parker, and S. Xu. 2011. A stochastic model for quantitative security analyses of networked systems. *IEEE TDSC* 8, 1, 28–43.
[135] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, Z. Chen, S. Wang, and J. Wang. 2018. SySeVR: A framework for using deep learning to detect software vulnerabilities. *CoRR* abs/1807.06756.
[136] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In *Proceedings of the NDSS*.
[137] Z. Lin, W. Lu, and S. Xu. 2019. Unified preventive and reactive cyber defense dynamics is still globally convergent. *IEEE/ACM Trans. Netw.* 27, 3 (2019), 1098–1111.
[138] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. 2018. ReGuard: Finding reentrancy bugs in smart contracts. In *Proceedings of the ICSE*. 65–68.
[139] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making smart contracts smarter. In *Proceedings of the ACM CCS*. 254–269.
[140] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. 2015. Demystifying incentives in the consensus computer. In *Proceedings of the ACM CCS*. 706–719.
[141] L. Luu, Y. Velner, J. Teutsch, and P. Saxena. 2017. Smartpool: Practical decentralized pooled mining. In *Proceedings of the UsenixSecurity*. 1409–1426.
[142] Y. Marcus, E. Heilman, and S. Goldberg. 2018. Low-resource eclipse attacks on Ethereum's peer-to-peer network. Retrieved from http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/Enseignements/ProjetsCrypto/Ethereum/236.pdf.
[143] A. Mavridou and A. Laszka. 2017. Designing secure ethereum smart contracts: A finite state machine based approach. *arXiv:1711.09327*.
[144] Patrick McCorry, Alexander Hicks, and Sarah Meiklejohn. 2018. Smart contracts for bribing miners. In *Proceedings of the FinancialCRYPTO*. 3–18.
[145] Silvio Micali. 2016. Algorand: The efficient and democratic ledger. *arXiv preprint arXiv:1607.01341* (2016).
[146] A. Miller, A. Kosba, J. Katz, and E. Shi. 2015. Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. In *Proceedings of the ACM CCS*. 680–691.
[147] J. Mireles, E. Ficke, J. Cho, P. Hurley, and S. Xu. 2019. Metrics towards measuring cyber agility. *IEEE TIFS* 14, 12 (2019), 3217–3232.
[148] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. Retrieved from https://bitcoin.org/bitcoin.pdf.
[149] Ryuya Nakamura, Takayuki Jimba, and Dominik Harz. 2019. Refinement and verification of CBC casper. *Networks* 2 (2019), 4.
[150] C. Natoli and V. Gramoli. 2017. The balance attack or why forkable blockchains are ill-suited for consortium. In *Proceedings of the IEEE/IFIP DSN*. 579–590.
[151] D. Nicol, W. Sanders, and K. Trivedi. 2004. Model-based evaluation: From dependability to security. *IEEE TDSC* 1, 1 (2004), 48–65.
[152] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the ACSAC*. 653–663.
[153] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Vol. 2283. Springer.
[154] Jianyu Niu and Chen Feng. 2019. Selfish mining in Ethereum. *arXiv:1901.04620*.
[155] S. Noel and S. Jajodia. 2017. *A Suite of Metrics for Network Attack Graph Analytics*. Springer International Publishing, Cham, 141–176.
[156] Russell O'Connor. 2017. Simplicity: A new language for blockchains. In *Proceedings of the PLAS*. 107–120.
[157] D. Park, Y. Zhang, M. Saxena, P. Daian, and G. Roşu. 2018. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the of ACM ESEC/FSE*. ACM, 912–915.
[158] M. Pendleton, R. Garcia-Lebron, J. Cho, and S. Xu. 2016. A survey on systems security metrics. *ACM Comput. Surv.* 49, 4, 62:1–62:35.
[159] L. Quan, L. Wu, and H. Wang. 2019. EVulHunter: Detecting fake transfer vulnerabilities for EOSIO's smart contracts at webassembly-level. *arXiv:1906.10362*.

[160] A. Ramos, M. Lazar, R. H. Filho, and J. J. P. C. Rodrigues. 2017. Model-based quantitative network security metrics: A survey. *IEEE Commun. Surveys Tutor.* 19, 4 (2017), 2704–2734.
[161] F. Ritz and A. Zugenmaier. 2018. The impact of uncle rewards on selfish mining in ethereum. In *Proceedings of the IEEE EuroS&P*. 50–57.
[162] M. Rodler, W. Li, G. Karame, and L. Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv:1812.05934*.
[163] G. Roşu and T. Şerbănuţă. 2010. An overview of the K semantic framework. *J. Logic Algebra. Program.* 79, 6 (2010), 397–434.
[164] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen. 2019. Exploring the attack surface of blockchain: A systematic overview. *arXiv:1904.03487*.
[165] K. Salah, M. Rehman, N. Nizamuddin, and A. Fuqaha. 2019. Blockchain for AI: Review and open research challenges. *IEEE Access* 7 (2019), 10127–10149.
[166] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308.
[167] F. Schrans, S. Eisenbach, and S. Drossopoulou. 2018. Writing safe smart contracts in Flint. In *Proceedings of the ACM on Programming Languages*. ACM, 218–219.
[168] Robert W. Sebesta. 2012. *Concepts of Programming Languages*. Pearson, Boston.
[169] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: A smart contract intermediate-level language. *arXiv:1801.00687*.
[170] Yonatan Sompolinsky and Aviv Zohar. 2015. Secure high-rate transaction processing in bitcoin. In *Proceedings of the FinancialCRYPTO*. 507–527.
[171] Matt Suiche. 2017. Porosity: A decompiler for blockchain-based smart contracts bytecode. In *Proceedings of the DEF CON*. 11.
[172] A. Suliman, Z. Husain, M. Abououf, M. Alblooshi, and K. Salah. 2018. Monetization of IoT data using smart contracts. *IET Netw.* 8, 1 (2018), 32–37.
[173] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss et al. 2016. Dependent types and multi-monadic effects in F. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 256–270.
[174] A. Tann, X. Han, S. Gupta, and Y. Ong. 2018. Towards safer smart contracts: A sequence learning approach to detecting vulnerabilities. *arXiv:1811.06632*.
[175] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the IEEE/ACM WETSEB*. 9–16.
[176] P. Tsankov, A. Dan, D. Cohen, A. Gervais, F. Buenzli, and M. Vechev. 2018. Securify: Practical security analysis of smart contracts. *arXiv:1806.01143*.
[177] F. Tschorsch and B. Scheuermann. 2016. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Commun. Surveys Tutor.* 18, 3 (2016), 2084–2123.
[178] Marko Vukolić. 2017. Rethinking permissioned blockchains. In *Proceedings of the ACM BCC*. 3–7.
[179] Wenbo Wang, Dinh Thai Hoang, Peizhao Hu, Zehui Xiong, Dusit Niyato, Ping Wang, Yonggang Wen, and Dong In Kim. 2019. A survey on consensus mechanisms and mining strategy management in blockchain networks. *IEEE Access* 7 (2019), 22328–22370.
[180] X. Wang, X. Zha, G. Yu, W. Ni, R. Liu, Y. Guo, X. Niu, and K. Zheng. 2018. Attack and defence of ethereum remote apis. In *Proceedings of the GC*. IEEE, 1–6.
[181] Benjamin Wesolowski. 2019. Efficient verifiable delay functions. In *Proceedings of the EUROCRYPT*. 379–407.
[182] F. Winzer, B. Herd, and S. Faust. 2019. Temporary censorship attacks in the presence of rational miners. In *Proceedings of the IEEE EuroS&PW*. 357–366.
[183] M. Wohrer and U. Zdun. 2018. Smart contracts: Security patterns in the ethereum ecosystem and solidity. In *Proceedings of the IEEE IWBOSE*. 2–8.
[184] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
[185] Karl Wüst and Arthur Gervais. 2016. *Ethereum Eclipse Attacks*. Technical Report. ETH Zurich.
[186] Y. Xiao, N. Zhang, W. Lou, and Y. Hou. 2019. A survey of distributed consensus protocols for blockchain networks. arxiv:1904.04098
[187] M. Xu, G. Da, and S. Xu. 2015. Cyber epidemic models with dependences. *Internet Math.* 11, 1 (2015), 62–92.
[188] Shouhuai Xu. 2014. Cybersecurity dynamics. In *Proceedings of the HotSoS*. 14:1–14:2.
[189] Shouhuai Xu. 2014. Emergent behavior in cybersecurity. In *Proceedings of the HotSoS*. 13:1–13:2.
[190] Shouhuai Xu. 2019. Cybersecurity dynamics: A foundation for the science of cybersecurity. In *Proactive and Dynamic Network Defense*, Zhuo Lu and Cliff Wang (Eds.). Vol. 74. Springer International Publishing, Cham, 1–31.
[191] Shouhuai Xu, Wenlian Lu, and Li Xu. 2012. Push- and pull-based epidemic spreading in arbitrary networks: Thresholds and deeper insights. *ACM Trans. Auton. Adapt. Syst.* 7, 3 (2012), 32:1–32:26.

[192] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun. 2019. Potential risks of hyperledger fabric smart contracts. In *Proceedings of the IEEE IWBOSE*. 1–10.
[193] V. Zamfir, N. Rush, A. Asgaonkar, and G. Piliouras. 2018. Introducing the "Minimal CBC Casper" Family of Consensus Protocols. Retrieved from https://github.com/cbc-casper/cbc-casper-paper/blob/master/cbc-casper-paper-draft.pdf.
[194] G. Zeng, S. Yiu, J. Zhang, H. Kuzuno, and M. Au. 2017. A nonoutsourceable puzzle under GHOST rule. In *Proceedings of the IEEE PST*. 35–358.
[195] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the ACM CCS*. 270–282.
[196] R. Zhang, R. Xue, and L. Liu. 2019. Security and privacy on blockchain. *CoRR* abs/1903.07602.
[197] R. Zheng, W. Lu, and S. Xu. 2015. Active cyber defense dynamics exhibiting rich phenomena. In *Proceedings of the HotSoS*. 2:1–2:12.
[198] R. Zheng, W. Lu, and S. Xu. 2018. Preventive and reactive cyber defense dynamics is globally stable. *IEEE Trans. Netw. Sci. Eng*. 5, 2 (2018), 156–170.
[199] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey. 2018. Erays: Reverse engineering ethereum's opaque smart contracts. In *Proceedings of the USENIXSecurity*.
[200] L. Zhu, B. Zheng, M. Shen, S. Yu, F. Gao, H. Li, K. Shi, and K. Gai. 2018. Research on the security of blockchain data: A survey. *CoRR* abs/1812.02009.

Received August 2019; revised February 2020; accepted March 2020