Front. Comput. Sci., 2021, 15(2): 152802
https://doi.org/10.1007/s11704-020-9284-9

REVIEW ARTICLE

Ethereum smart contract security research: survey and future research opportunities

Zeli WANG[1,2], Hai JIN[1,2], Weiqi DAI (✉)[1,3,4], Kim-Kwang Raymond CHOO[5], Deqing ZOU[1,3,4]

1 National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Clusters and Grid Computing Lab, Hubei Engineering Research Center on Big Data Security, Wuhan 430074, China
2 School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China
3 School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China
4 Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, China
5 Department of Information Systems and Cyber Security, University of Texas at San Antonio, San Antonio, TX 78249-0631, USA

Abstract Blockchain has recently emerged as a research trend, with potential applications in a broad range of industries and context. One particular successful Blockchain technology is smart contract, which is widely used in commercial settings (e.g., high value financial transactions). This, however, has security implications due to the potential to financially benefit from a security incident (e.g., identification and exploitation of a vulnerability in the smart contract or its implementation). Among, Ethereum is the most active and arresting. Hence, in this paper, we systematically review existing research efforts on Ethereum smart contract security, published between 2015 and 2019. Specifically, we focus on how smart contracts can be maliciously exploited and targeted, such as security issues of contract program model, vulnerabilities in the program and safety consideration introduced by program execution environment. We also identify potential research opportunities and future research agenda.

Keywords smart contract, security, blockchain, vulnerability, unreliable data

1 Introduction and motivation

The "first wave" of cryptocurrency research probably takes place between the 1980's to early 2000's, such as "anonymous transactions" [1], "online shopping without bank" [2], DigiCash [3], and Peppercoin [4]. The Blockchain-based Bitcoin was proposed in late 2000s [5], and since the popularity of Bitcoin, a number of altcoins, including Blockchain-based altcoins, have been proposed in the literature and market. For example, there are approximately 2,169 cryptocurrencies been tracked by CoinMarketCap. It is, perhaps, the popularity of Bitcoin that the market recognizes the potential of Blockchain, for example its capability to achieve properties such as decentralization, tamper-proofing, transparency, and traceability. In the past few years, there have been a number of Blockchain-related studies focusing on a broad range of applications [6–20].

In addition to Bitcoin, another widely successful Blockchain application is Ethereum [21], which uses Turing-complete programing language to enable users to develop smart contracts on Blockchain. This is also referred to as the Blockchain 2.0 era, where different applications can be built on smart contracts (e.g., Internet of Things [22, 23], healthcare [24–27], commercial services [28, 29], secure data exchange [30] and so on [31–33]). First proposed by Szabo [34], smart contracts generally have the following features: (1) self-executing: triggered by transactions, without the need for manual interaction; (2) self-enforcing: once triggered, smart contracts cannot be prevented from executing; (3) transparency: smart contracts are known to each node in the Blockchain network, since their correctness must be verified by most nodes; and (4) flexibility: they can adjust to different scenario requirements. The popularity of smart contracts is also evidenced by the interest in smart contract programming languages and platforms. While there are a large number of challenges related to smart contracts, security is one key challenge [35, 36] and hence, the focus of this survey.

So why is the security of smart contracts important? First, such contracts are generally used in financial settings, and hence they are an attractive target for financially- and criminally-motivated cybercriminals. In addition, any successful breach, particularly those that are highly publicized, can impact on the community's belief in smart contracts, and hence its usage. There have been a small number of such incidents in recent years, such as the incidents involving the Decentralized Autonomous Organization (DAO) event and Parity. More recently in 2018, attackers hacked Fomo 3D, a game Dapp, by conducting front-running and injecting Blockchain network lots of transactions with high fees to prevent other transaction from being packed, finally stealing around 10469.66 ETH (see SECBIT report). A snapshot of recent attacks is presented in

Received August 6, 2019; accepted December 25, 2019

E-mail: wqdai@hust.edu.cn

Table 1 A snapshot of recent smart contract attacks

Year of event	Smart contract	Loss
2016/6	The DAO	3,600,000 ETH [37]
2017/10	SmartBillions	400 ETH [38]
2017/11	Parity	514,000 ETH [39]
2018/4	BeautyChain	1,000,000,000 USD [40]
2018/4	SmartMesh	140,000,000 USD [40]

Table 1.

Due to the wide diversity in Blockchain platforms and programming languages, we focus only on Ethereum smart contracts due to the following reasons: (1) Ethereum is the first Blockchain-based application to realize Turing-complete smart contracts; (2) it is one of the most popular and widely used smart contract platforms; (3) transaction throughput always ranks first; and (4) Solidity designed for Ethereum contracts is the most widely used programming language.

Main contributions A summary of our contributions in this paper is as follows:

- ***Systematic mapping of smart contract security challenges*** We perform a systematic and comprehensive literature review[1)] of existing research literature on smart contract security, and categorize these security challenges into abnormal contract, program vulnerability, and unsafe external data.
- ***Systematic mapping of potential solutions*** For each security challenge, we also describe the potential solutions.
- ***Future research agenda*** Based on the survey, we present a number of potential future research directions.

The remainder of this paper is structured as follows. In Section 2, we will briefly introduce the Blockchain infrastructure and Ethereum smart contracts. In Section 3, we will describe our literature review protocol. In Sections 4 to 6, we will describe the key security challenges and potential solutions relating to *Abnormal Contract*, *Program Vulnerabilities* and *Exploitable Habitat*. In Section 7, we summarize the existing state-of-play on smart contract security and discuss potential research directions. In the last section, we conclude this paper.

2 Overview of Blockchain and smart contract

2.1 Blockchain

Blockchain, in general, is a distributed ecological system, where all nodes are independent in view of interests or willings to execute tasks but meanwhile close-related in view of maintaining an identical ledger by competition / cooperation. Moreover, each transaction after being verified will be packed into the block, which links to the latest block by setting previous block hash as one of input in computing current block hash. Only when the majority of miners reach a consensus, can this transaction be recorded into Blockchain ledger. All the above behaviors introduce Blockchain several remarkable characters, such as decentralization, transparency and tamper-proofing.

Blockchain underpins many smart contract platforms, and its popularity is ever increasing. There have also been an increasing awareness on the importance of security on Blockchain-based platforms. For example, the OpenZeppelin is an open source architecture designed to enable secure, test-providing, and audited smart contract code development, and Quantstamp specializes in auditing projects for making smart contracts safer. Building on existing surveys such as [41], Blockchain platform can be classified into two main categories, permissionless and permissioned. In the former, everyone can join or exit arbitrarily, while permissioned Blockchain adds an extra member management modular designed to put restrictions on participants, where only predesignated members have rights to access and maintain the Blockchain ledger. So permissioned Blockchain is semi-decentralized, but it appeals much more attentions than public chain in business circle. Because its higher throughputs due to sharp shrink of node amount, and security due to access control mechanism are more corresponding to practical application scenarios. But in general, public Blockchain is most popular due to its nearly complete compliance to original Blockchain design goals.

To guarantee the same contracts held by different nodes to obtain the same input and output the identical result, the key is to ensure the determinacy of contracts. In other words, at the Blockchain level, we need to ensure the distributed ledger technology (DLT) is correct and consistent between all nodes. Thus, we need to have the consensus mechanism. Blockchain consensus should consider the presence of malicious users, who seek to maximize their benefits even at the cost of destroying the entire system. Existing consensus mechanisms can be categorized into deterministic and nondeterministic. In the former category, once the block is added on the main chain, all transactions in the block is determined and cannot be modified anymore – typically represented by PBFT [42, 43]. PBFT is suitable for permissioned Blockchain, where members (determined in advance) reach an agreement via the following three steps: pre-prepare, prepare and commit. Typical nondeterministic consensus mechanisms are Proof of Work (PoW) [5] and Proof of Stake (PoS) [44, 45]. In this kind of consensus, even if the block is appended on the main chain, it may be invalidated later. Since there are forks, two valid child blocks belonging to the same parent block may be mined simultaneously, one of which will be discarded if the chain in which the other block is located becomes the longest chain. We refer reader interested on Blockchain consensus to [10, 11], which plays a key role in guaranteeing smart contract consistence and determinacy.

2.2 Smart contract

As our paper focus on Ethereum smart contracts, so we will now revisit their underlying concept and execution context in this section. Specifically, we will describe three key components, namely: accounts, transactions, and EVM.

Accounts and storage Ethereum adopts an account mode similar to the account management mechanism in conventional banking system. There are two account types, namely: external owned (EOA) and contract. Both account types are uniquely identified by a 20-byte address (i.e., their identities in the Ethereum network). EOAs are controlled by public/private key pairs, mainly used to manage ethers and interact with contracts

[1)] In the literature and this paper, the terminologies "literature review" and "literature survey" are used interchangeably. Similarly, "review" and "survey" are also used interchangeably

by sending transactions. While contract accounts are controlled by codes without keys, and mainly used to implement diverse function requirements and record contract state changes such as executed transactions and balance modification. Unlike EOAs, contract accounts cannot send transactions but they can send message calls to call other contracts. In addition, contract accounts cannot interact with EOAs proactively; however, they can use some "radical" mechanism such as self-destruction (where all their holding ethers will be refunded to their creators, upon successful execution). The contract account also has a room to store code hash that can be used to find codes, whilst EOAs do not store this value. In terms of similarity, both account types will store nonce, asset balance and the root hash of all stored states. The nonce is a mechanism designed to mitigate reply attacks and double spending, which will be incremented by 1 once an account sends a transaction. In other words, if one broadcasts a transaction again and again hoping it will be executed by miners many times, the miners will identify it as a repeated transaction (due to the nonce) and discard it. Clearly, if an attacker attempts to use the same nonce to facilitate double spending, then the gas prices appended in the transaction will determine the winner, because they are more likely to be processed first.

Transactions and interaction Transactions will be broadcast to every miner for execution, and change the Blockchain storage state after reaching consensus. Each transaction will specify its account nonce (as discussed above), price per gas, the maximum gas payment for this particular transaction, transferred value, recipient, input data and signature. Of the information it contains, gas charging mechanism plays a key role in many aspects, for example: 1) compensation and incentive for miners executing and storing transactions; 2) countermeasure to prevent denial of service (DoS) attacks such as malicious users sending computation-complex transactions. Different combinations of recipient and payload have diverse functions, such as those listed in Table 2. Since we focus only on smart contracts, we will explain cases 2 and 3 only.

- In case 2, a transaction with null recipient will be regarded as a contract creation request. In such a case, the miners will execute bytecodes included in the payload. Therefore, the payload includes not only contract codes, but also initial codes and parameters in the constructor. Initial codes are responsible for creating the contract account, storing codes, initialing constructor functions, etc. They will then be discarded after completing the execution.
- In case 3, when the recipient is a contract address, miners will call the related contract. There are four ways to realize case 3. In order to better introduce them next, we will explain the "Call" function first, which is also a kind of method to call the contracts, with an external API and underlying API. Unlike transactions, "Call" will not propagate in network or obtain consensus, and they are local vocations by reading local state database without making any change, suitable to view, pure and constant functions in Solidity. Although they do not charge, they will fail if there is a lack of gas. This is because they need to be executed in the local EVM. Contracts can be called by both EOAs and other accounts using "Call". Clearly, they can be called by EOAs using transactions. How can they be called by contracts using transactions? Well, contracts can interact with each other, and they use "message call" rather than transactions. Theoretically speaking, "message call" is a also a kind of transaction, which allows callers to obtain the return values promptly.

Table 2 Function description under different combinations of recipient and payload

Case	Recipient	Payload	Function
1	EOA	Null	Value transfer
2	Null	Contract deploying codes	Contract account creation
3	Contract	Function parameters	Contract call

EVM and execution The EVM is an architecture based on stack rather than register, and provides a separated execution environment to protect contract execution from external attacks and avoid malicious contracts affecting the entire system. There are three kinds of storage structure to assist operations, namely: stack, memory and storage. Stack can be used to store local variables for free, and memory is used to store parameters and return values, which will be released after being called by functions. Stack and memory are volatile, which will be cleared after completing the transaction. Storage will store state variables persistently; therefore, they are expensive. Once calls to the contracts received (by message calls or transactions), the EVM will first search and load contract code from the local database. Unlike conventional software with only one entry point (main()), all public functions in smart contracts can be entry points. Therefore, the first four bytes of payload, function signature, will point out the calling function, followed by the function parameters. The EVM will find the corresponding function bytecodes and parse them to opcodes byte-by-byte. Moreover, each opcode is bound with a detailed operation definition. Based on these instructions, the EVM will process these parameters. The machine may throw exceptions for reasons, such as stack underflow or invalid instructions. Thus, contract vulnerabilities will be invoked in the EVM eventually. The execution results (including participants' balance change, execution logs and receipts) will be packed as "StorageObject" stored into the Blockchain ledger. Since this entire process is repeatedly operated by all nodes in the Blockchain system, as long as compromised nodes are less than majority, the Blockchain can still run stably. The compromise of several nodes will not impact the overall system's security. However, we must guarantee that the same smart contract on all nodes receives the same transaction, data and generates identical output. Interested reader is referred to Ethereum yellowpaper for more details about the contract principles.

3 Literature review protocol

There are a number of literature review protocols. For example, we can perform a systematic mapping study (SMS) or a systematic literature review (SLR). SMS allows one to perform a somewhat high-level / cursory review in order to obtain some visual classification, whilst an SLR is an in-depth, comprehensive analysis of a relatively established / mature field by search-

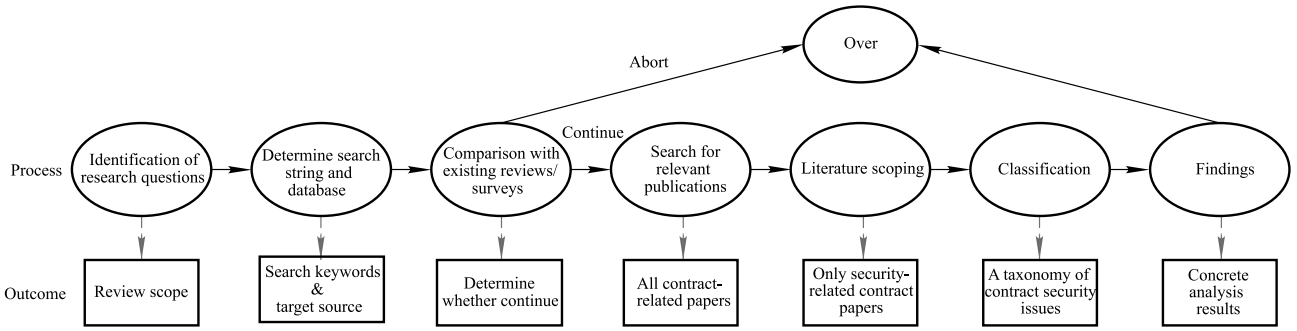

Fig. 1 Our literature review protocol

ing for and evaluating the existing literature. Since smart contract technology is still in its infancy, in this paper, we choose the former [46] as our literature review protocol – see also Fig. 1 and Sections 1 to 5.

3.1 Identification of research questions

Research questions should reflect the main goals of performing the literature review explicitly, guide us to locate the relevant publications, and provide an insight on the research field. In this paper, our research questions and sub-questions are as follows:

- **RQ1: How is the existing security state-of-play for smart contracts?** *RQ1.1: What are the security challenges in existing smart contracts? RQ1.2: Why are existing smart contracts prone to such attacks?* Only by having an in-depth understanding of the root source(s) of the security challenges will we be able to mitigate such challenges.
- **RQ2: How to design countermeasures to the identified security challenges?** *RQ2.1: What are the existing countermeasures to each the identified security challenge? RQ2.2: What are the benefits and limitations associated with the different countermeasures?*
- **RQ3: When and where are these studies about smart contract security published?** This allows us to understand the underpinning approaches (e.g. software engineering, distributed system, or network traffic analysis) and the recency of the challenges and countermeasures.
- **RQ4: What are potential future research challenges?**

3.2 Literature search

To facilitate the literature search, we will use the PICO principles generally used in the medical literature. Specifically, as shown in Table 3, we adopt the approach of Pahl et al. [47] to help us identify the relevant keywords and databases.

3.2.1 Comparison with existing reviews/surveys

We combine Population with Comparison to form our search string, since there are a number of existing literature review/survey articles on different aspects of Blockchain, cryptocurrencies and smart contracts. This will allow us to better refine our focus, to avoid having a significant overlap with existing literature review/survey articles. For example, Bonneau et al. [48] surveyed the cryptocurrency research, particularly focusing on Bitcoin. However, the focus is on Blockchain 1.0 without Turing-complete smart contracts. By contrast, we explore emerging smart contracts in the Blockchain 2.0 era. Tschorsch and Scheuermann [6] surveyed decentralized digital currencies, particularly focusing on Bitcoin. Specifically, the authors introduced the core protocol of Bitcoin, and discussed issues relating to security, network, privacy and consensus. Along a similar line, Conti et al. [7] examined the security and privacy challenges in Bitcoin, and discussed countermeasures and potential research directions; and Khalilov and Levi [8] reviewed anonymity and privacy challenges in Bitcoin-like digital cash systems. Ferrag et al. [9] reviewed existing research efforts and challenges when applying Blockchain to Internet of Things (IoT). The brief reviews of Alharb and van Moorsel [49] and Atzei et al. [50] are, perhaps, closer to the aim of this paper. Specifically, Alharb and van Moorsel [49] focused on codifying, security, privacy and performance issues relating to smart contracts, and Atzei et al. [50] presented a taxonomy of common programming pitfalls that results in Ethereum vulnerabilities. Wang et al. [51] presented a systematic and comprehensive overview of Blockchain-enabled smart contracts about architecture, application and future trends three levels. However, no review presents an in-depth treatment of the security issues underpinning smart contracts. Specifically, in this survey, we focus on abnormal contract, program vulnerability and unsafe external data issues in smart contracts – see proposed taxonomy in Fig. 2. The taxonomy is built on the premise that codes and data are central to (the security of) smart contracts, namely: entire contract function (*Abnormal Contract*), local code issue (*Program Vulnerability*), and malicious environment (*Exploitable Habitat*).

3.2.2 Search for relevant publications

To avoid bias and preference, we overlook ICO related key-

Table 3 PICO principles and applying on smart contract security

PICO entry	Original concept	Present meaning
Population	Objects affected by intervention	Smart contract OR Ethereum OR Blockchain OR cryptocurrency OR token OR Dapp OR EVM
Intervention	Diverse treatments or behaviors	Attack OR analy* OR exploit OR validate OR secur* OR safety OR crim*
Comparison	Contrast with other treatments	Survey OR review OR evaluation OR study
Outcome	The treatments will use on population	Design OR Strategy OR model OR language OR framework

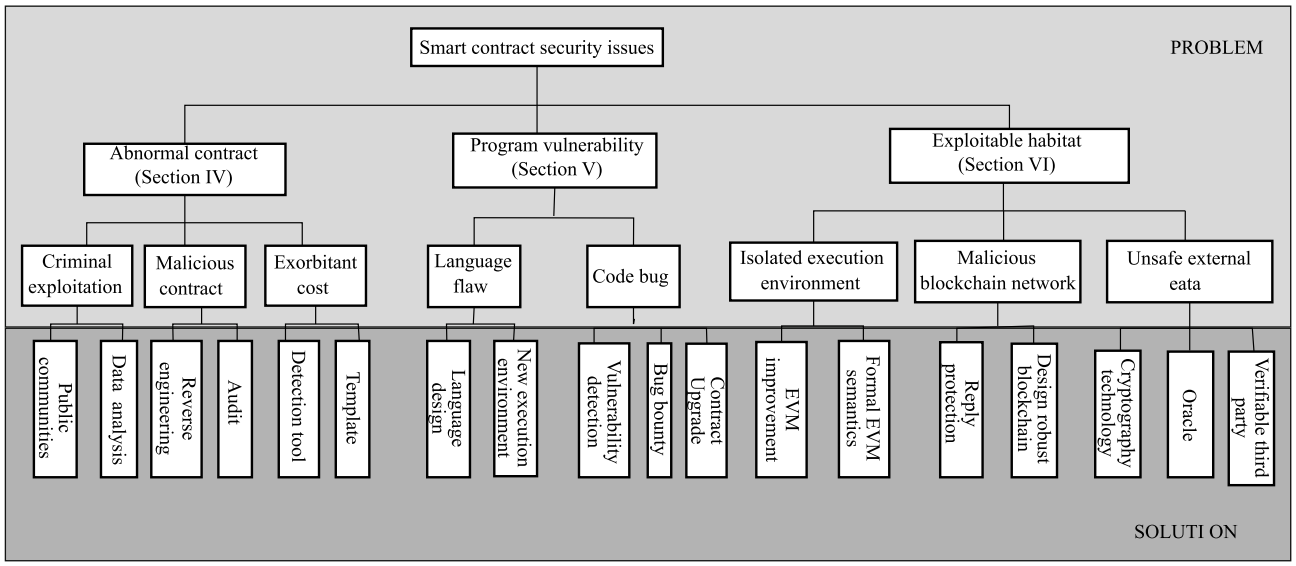

Fig. 2 Smart contract security: a taxonomy

words temporarily but left for our literature pruning in next section. So we just use population in Table 3 as our search strings. As to search databases, we do not limit it to one or several repositories or venues in security, because smart contract security covers lots of facets. Instead, Google Scholar is our key search platform since it indexes publications from major publishers. We also use DBLP, a computer science bibliography, wherever possible to obtain the BIBTEX entry. In our search, we also adopt the following strategy. Since the concept of smart contracts dates back to 1990s, but the Ethereum literature is mainly published after July 2015 as Ethereum client was launched successfully on July 30, 2015. In fact, we observe that there are only a handful of publications in 2015 during our search. In addition, to ensure that we capture recently accepted publications, we also scan the list of accepted papers at top relevant conferences. Besides, we will also scan the related referenced literatures in searched papers.

3.3 Literature scoping

After the prior steps, we obtain hundreds of publications, but many of these publications are not relevant. Our inclusion and exclusion criteria are described in Table 4.

3.4 Classification

We skim the publications concentrating on keywords, abstract, contribution and conclusions to identify the relevant features, in order to come up with a classification. If these parts are not enough to identify their features, we will scan their introduction even the main body to determine their belonged categories. In view of the relationships between issues and programs, all security issues can be classified from three facets: risks caused by program model (Abnormal Contracts), internal fragility of programs (Program Vulnerability), and dangers posed by their environment (Exploitable Habitat). So these three facets establish the first level of classification.

Then we make a big table to put each literature attached with keywords in categories. To further detailed division, based on these keywords and our knowledge of these scanning papers, we consider classifying each category from their goals or problem roots. For contract level, we divide them according to the first principle, in which some contracts are used to prompt criminal transactions, while some are specifically designed such as left a back door to scam money, by contrast, some contracts are apparently innocent but draw excess unnecessary cost. Finally, contract is classified into three sub-categories. For vulnerability level, we obey the second principle, some vulnerabilities result from programming languages, however, some are caused by programmers' careless. For habitat level, also we divide them according to different issue causes. Since the environment of smart contract has three hierarchies, from the inside out, that is, directly related EVM, Blockchain network, and external world respectively. At end, we get the second level categories.

After identifying all problems, we aim to conclude and classify all related countermeasures. Since some papers type is solution and proposal kind, they cover both problems and answers, in this way, we just need to put them under corresponding problems. To get a wider view, we will search specific problems again in Google scholar and browser to find more countermeasures. We classify these solutions mainly based on their involving technologies. Finally, we get our taxonomy of smart contract security issues, illustrated in Fig. 2. We will describe them at length in next three sections.

Table 4 Inclusion and exclusion criteria

Action	Criteria
Inclusion	The title, abstract or keywords explicitly mention smart contracts or Dapps or tokens or their execution environment such as EVM and security. Papers published at security conferences are also included. Also, if publications relate to vulnerabilities or pitfalls published at software engineering conferences, they are also included.
Exclusion	Publications that do not refer to Ethereum smart contracts or security are excluded. Examples include publications that use smart contracts for IoT security.

3.5 Findings

Retrospecting the selected studies, we find that smart contract

research is very hot especially in 2018. The related researches frequently arise in top conferences, which account for nearly thirty percent in over a hundred papers, among security and software engineering fields are most popular. Moreover, during our strengthened search process in top conferences, we found that top four security vendors nearly every year have Blockchain or cryptocurrency special columns but very few smart contract, which implies that smart contract issues are less popular than Blockchain. But from a global perspective, smart contract security gets more and more attention. More findings about contract security issues, countermeasures, challenges and research agendas are illustrated in the remainder.

4 Abnormal contracts

Smart contracts are deployed on all miners in the Blockchain network, and once triggered, they will be executed by each node. From trigger to execution, this process does not require user interaction, and is self-enforcing (i.e., contracts cannot be canceled once conditions defined in the contracts are satisfied). Thus, this allows fair-exchange. As smart contracts are stored on Blockchain as bytecodes, they are not human readable. Although there are source codes on some websites such as Etherchain and Etherscan, uploading source codes is not enforced and consequently the code libraries are not complete. Also, smart contracts process significant volume of digital assets. Hence, they are attractive to malicious users, for example to deploy malicious codes on Blockchain without making their source codes public, in order to obtain illicit financial gains. Finally, to prevent distributed DOS attacks such as infinite programs, smart contract platforms introduce gas mechanism to charge transaction sender for computation and storage consumption. Although this is a good way to protect the system, smart contracts may be designed to consume more gas that is unfair to users. We will elaborate on the following three potential abuse / risks in Sections 1 to 3.

4.1 Criminal exploitation

Advantages of smart contracts (e.g., self-execution, self-enforcing, self-destruction) can also be criminally exploited. For example, self-enforcing and self-execution enable the smart contract to be considered as a trusted third-party. This not only reduces attack cost such as relying on the intermediate service, but also provides a higher level of trust than a traditional third-party. For example, funds paid to a cybercriminal (e.g., in a ransomware or other extortion incidents such as sextortion scams) via smart contracts cannot be terminated and reversed, once the contracts are triggered by transactions. So such fund transfer does not require criminals' interaction. Moreover, since all participants in Blockchain are pseudonymous, the criminals' identities will be challenging to trace. Also, some smart contracts support self-destruction (e.g., suicide method). Thus, when the transactions complete, the criminals can destroy the smart contracts; thus, further complicating (forensic) investigations.

The notion that smart contracts can be criminally exploited (or criminal smart contracts – CSCs) is not new. For example in [52], the authors pointed out that smart contracts can be used in the criminal ecosystem, such as leakage of sensitive information, theft of private keys, and calling-card criminal activities. To explain how smart contracts may promote or be abused to facilitate criminal activities, we take the "Key-Theft" as an example. In this activity, the initiator wants to find a person (partner) to help him/her steal C's private key and will pay for the information. However, participants do not trust each other as either party may be an undercover law enforcement officer or one party may rip the other party off. In addition, how can the responding individual guarantee that the initiator will pay him/her as agreed? Hence, the need for fair-exchange. If in traditional ways, criminals need to use a trusted third party or communicate directly, however, in this way, criminals may be discovered by law enforcement or cheated. Instead, this deal can be easily achieved without these risks by decentralized smart contracts. Moreover, in [52], the authors defined a more strict transaction rule (commission-fair) than fair-exchange, which assumes that contractual parties are arbitrarily malicious and are satisfied only when all participants get their initial fair expected results. And in this way, fair-exchange is only a precondition of commission-fair. For example, in the Key-Theft contract, although we can achieve fair-exchange by defining a smart contract for rewards to be paid once the key is delivered, the contract cannot guarantee that the key is valid. For example, the key may have been revoked, and this results in an unfair situation. As introduced in [52], even under such strict requirements, smart contracts can also successfully satisfy criminals.

Smart contracts can also be used as a launch pad to attack the Blockchain system. For instance, in a typical block-withholding attack [53, 54], miners will obtain the bonus not only from full proof of work (FPoW) but also partial proof of work (PPoW). However, the computation administrators can control is limited. With the help of smart contracts, attackers can reach agreements with miners in other pool(s) to facilitate the launching of attacks. Since all rules have been written in smart contracts, which will be definitely executed after specific requirements are satisfied, so there is no need for the miner conspirator to trust the initiator's intention or payment capacity. Hence, they can be in collusion with each other more easily. Under this situation, any malicious user even with little computation can breach a large mining pool by collaborating with other miners, and reduces its revenue to zero with minimal or no financial penalty [55]. In addition to the pool attack, smart contracts may have a significant influence on Nakamoto consensus. The authors in [56] used three cases to explain how smart contracts can be leveraged to compromise consensus security. For example, a briber A without having majority computations, wants to control transactions in the mined blocks. Instead of renting hardware power to become the majority (and pays the full cost), A deploys a smart contract to employ miners, compensating them only after they mine the required uncle blocks. Because the inherent uncle block reward in the Blockchain can subside these bribes partially, to some extent, reducing A's attack costs. Also, smart contracts can also be used to cause hard forks maliciously and reduce the utility of certain cryptocurrency.

In summary, the fair-exchange feature of smart contracts can be criminally exploited to help non-trusting individuals cooperate with each other in a criminal activity. There have been some attempts to mitigate criminal activities on Blockchain. Wang et al. demonstrate through simulation that CSCs are not as powerful as expected, further they can be alleviated by in-

troducing randomness [57]. Meanwhile, some communities are built to guard smart contract ecology secure. For example, the Blockchain Alliance is a platform dedicated to enable different stakeholder groups (e.g., law enforcement, and banking and financial institutions) worldwide to collaborate in their crime-fighting and investigation activities. Another example is Crystal developed by the Bitfury Group's software team, which is designed to monitor the state of public Blockchain ecosystem. Advanced data analytic technologies are used to detect abnormal transactions and map them to related accounts. However, such initiatives particularly for CSCs are limited.

4.2 Malicious contract

Different from CSCs (where the codes are technically sound), malicious contracts refer to vulnerabilities (e.g., back-doors) intentionally introduced to facilitate criminal activities such as stealing of contracts' digital assets. For example, in some smart contracts, creators may seek to maximize the utility of their smart contracts, for example by offering incentives or discounts. Once they have had some critical mass, these creators then make use of intentionally introduced back-doors to steal the digital assets. One real-world example is LastWinner, which is designed to mostly target the Chinese population. Honeypot is another an approach to scam users, which seems vulnerable but contains unconspicuous traps, [58] makes a taxonomy of smart contract honeypots. Moreover, it employs symbolic execution to collect program informations, cash flow analysis to exclude completely impossible honeypots that cannot receive or transfer funds, and honeypot analysis combining detection rules with heuristic knowledges to implement HoneyBadger, a honeypot detection tool.

Since it is not always easy to identify such malicious contracts as it is not mandatory to share the contract's source codes. For example, according to [59], as of January 2018, 26,000 (approximately 77%) smart contracts do not have readily available source codes, although not all such contracts are malicious. What's more, albeit contracts' bytecodes are transparent to anyone but bytecodes are hard to be understood by human. Hence, there have been focus on smart contract reverse engineering. Researchers from University of Illinois, for example, explained how one can reverse engineer Ethereum's smart contracts from their bytecodes [59] using Erays. Erays first transfers bytecodes into high-level pseudocodes suitable for manual analysis, in which conventional software technologies can then be used. For example, one can use linear sweep [60] to disassemble the hex string (bytecode) into EVM instructions. Finally, after obtaining the pseudocode, Erays can link bytecodes to publicly available source codes based on code similarity. But the reversed pseudocodes are not much read friendly and have some errors due to incomplete recovery and simulation of evm operations. Other similar reverse engineering tools include Etherscan that can reverse bytecodes into opcodes. However, opcodes are still relatively obscure and challenging to work with. To analyze contracts easier, Gigahorse [61] can transfer Ethereum bytecode into 3-address code representation making data- and control- flow dependencies explicit. There have also been efforts to convert EVM bytecodes to source codes.There have also been efforts to convert EVM bytecodes to source codes. However, as pointed out by Parizi et al. [62], "research on the empirical knowledge evaluation of security testing for smart contracts is scarce in the literature" and it is not trivial for even smart contract developers to security test their own product.

4.3 Exorbitant cost

Costly contracts can be either gas-costly or verification-costly. Ethereum supports Turing-complete languages to satisfy different requirements, but this can be exploited to write a transaction or contract that demands significant computation power to perform DoS attacks in the Blockchain system. Consequently, transactions will be blocked and transaction fees experience a steep increase; thus, affecting system stability. Therefore, smart contract platforms generally run a gas mechanism to prevent DoS attacks. Such platforms provide computation as a service, and users pay for their computation consumption on the Blockchain. Each operation will be charged, and there is a charge rule for gas cost of different operations. This discourages attackers (or in fact, most users) from creating complex contracts or transaction scripts to consume significant Blockchain resources. In [63], however, the authors revealed that some contracts cost too much gas relative to the normal. They identified seven under-optimized patterns, which were then grouped into two categories. Category 1 (useless code related patterns) refers to gas-costly codes with no real functions, and category 2 (loop related patterns) involves expensive or unnecessary operations in a loop that incur additional (unnecessary) costs. The authors also introduced a new tool, Gasper based on Oyente [64], to automatically discover gas-costly programming patterns based on bytecodes. Specifically, Gasper first recovers control flow graph (CFG) based on disassembled codes, then symbolically executes each basic block according to CFG to derive the necessary information for subsequent analysis. Gasper will identify dead codes by comparing whether all blocks in CFG are explored by symbolic execution. However, Gasper can only identify specified patterns with significant limitations. A probably better dynamic gas charge mechanism is also proposed to alleviate this problem [65]. Developers can also use developer libraries and templates to program gas-efficient contracts.

The high costs can also be attributed to bad design logic in the smart contracts, which result in additional unncessary overheads. Since each smart contract execution will be executed by all full nodes on the Blockchain, any complex computation will incur resource wastage. Luu et al. [66] explained that the attacker can construct customized script that requires nontrivial computation effort — *resource exhaustion attack*, and even the gas mechanism is not capable of preventing such an attack. Specifically speaking, focusing on a matrix operation (A*B) contract, the attacker may create an expensive transaction script involving two large size matrices with low gas prize, so other miners are not willing to pack this transaction into the block. Hence, the attacker will pack this himself/herself. In this way, although this transaction will cause huge gas consumption, all gas will be rewarded to the miner (attacker). Then, all nodes have to execute this transaction after the block is broadcasted, and decide whether to verify this transaction. If verifiers check the transaction, this will consume much of their computation and time resources; thus, affecting their probability to mine the

next block. If not, the attacker may involve an illegal transaction. Since gas charged in each transaction is credited to the attacker's account, (s)he will launch this attack successfully at minimal or no cost. Such costly contracts are more difficult to deal with, in comparison to the first type of costly contracts. Since this is not only a design issue but also one that concerns the underlying consensus mechanism (how to incentivize miners to verify transactions), a consensus-based computation model where individuals should only accept to verify the transaction that has a specified bounded gas limit is required.

5 Program vulnerability

Vulnerability analysis is an ongoing research topic. Unlike conventional software programs, smart contract codes are much more vulnerable. Also, since smart contracts run on a large-scale complex Blockchain system, they will be subjected to more complicated situations. For example, smart contracts must be executed by all nodes in the same order of transactions. During the execution process, there is a broad range of attack vectors. For example, the authors in [50] surveyed smart contract vulnerabilities, and made a taxonomy of existing popular pitfalls for Solidity, EVM, and Blockchain. [67] analyzes concurrency exploits, which are posed because contracts can be called simultaneously by multiple transactions, and different transactions execution order may significantly influence the final results. There are also other public resources relating to smart contract vulnerabilities, such as Ethereum blog and Consensys community. To better classify existing solutions, we propose categorizing program vulnerabilities into *language flaw* (1) and *code bug* (2). The *code bug* mainly indicates vulnerabilities due to the programmer's negligence. If the vulnerability is introduced intentionally, then this is considered *Malicious Contract* (see Section 2).

5.1 Language flaw

Programming language limitation is one of prime culprits for smart contract vulnerabilities. For example, Solidity is an object-oriented high-level programming language and currently one of the most popular programming languages. However, it has many constrains. For example, Solidity does not have a standard library that supports arrays and strings, and it does not allows one to compare two strings directly. Solidity has more drawbacks than other established languages like Python, C++, and Javascript. For instance, "assert" may use up all remaining gas, date suffixes cannot be applied to variables (only up to three parameters can be received as the attribute indexed for event parameters), and so on.

There have also been attempts to create new languages for programming more robust contracts or easier verifying correctness. For example, various alternative languages for Solidity such as Bamboo, Obsidian [68], Vyper, Flint [69], Yul, Babbage have been proposed to enhance code security. Bamboo aims to enforce programmed codes execute in an expected order eliminating nondeterminacy, moreover, by modeling contracts as state machines and making state transition explicit, Bamboo perfectly circumvents reentrancy attack. Similar to Bamboo, Obsidian [68] is another language modeling contracts as state machines, but more advanced than Bamboo, adds additional functions to track contract states statically, finally facilitating developers programming correct codes. However, Obsidian focuses only on mitigating three sources of vulnerabilities, namely: high-level state dependence addressed by moving dynamical state information into static types (thus, allowing one to analyze them statically); monetary loss addressed by tracking financial changes using a linear type system; and re-entrancy attacks. Seeking to construct more secure, easily implementable and human-readable smart contracts, Vyper, extends Solidity to achieve enhanced security. However, Vyper deliberately restricts several actions that are allowed in Solidity (e.g., modifiers, class inheritance and function overloading); thus, limiting the smart contracts' functionalities. Moreover, it is still in a beta stage at the time of this research. Thus, its effectiveness remains to be verified. Besides these high-level programming languages, there are also some languages specially designed to verify contract security conveniently. Flint [69, 70], for example, is a language for better contract ecosystem, which can be more easily verified and rectified by introducing features such as asset types and type states and setting different restrictions on them. Taking asset types as an example, Flint only allows limited operations in predefined patterns, restricting some dangerous operations like creation or duplication. However, at the time of this research this language still in alpha version and not ready to be used in production yet. Rather than focus on high-level programming language, Ethereum community also develops Yul intermediate language able to be transferred to bytecodes for various Ethereum backends (EVM1.0, EVM1.5 and eWASM). While Babbage is a visual programming language, aimed at revealing how different parts interact and fit together. In this way, users can better understand smart contracts.

Of these programming languages, functional may be preferred over other imperative programming languages. This is because the other imperative programming languages tell software how to do to achieve what we want, which is clumsy and complex. Functional, on the other hand, follows a declarative programming model, supports parallel programming, and is more easily audited; thus, it is suitable for smart contracts running on Blockchain. In addition, it treats computation as the evaluation mathematical functions without changing state and mutable data (see Wikipedia), and appears to introduce fewer vulnerabilities. Due to space limitation, we cannot enumerate each language concretely. Based on existing classification methods [101], Table 5 tries to cover existing popular languages for Ethereum as much as possible from three different level, high-level, intermediate and low-level.

5.2 Code bug

It is impossible to claim that a program does not have any single bug, even if the developing team is very experienced and the program has been audited multiple times. The authors in [50], for example, presented a detailed summary of smart contract bugs. Vulnerabilities in smart contracts are constantly been discovered. For example, a simple keyword search for smart contracts on the website of Common Vulnerabilities & Exposures (CVE) revealed 100 or more vulnerabilities in smart contracts published in 2018 (e.g., CVE-2018-13746, CVE-2018-13661, CVE-2018-12702, CVE-2018-12079, CVE-2018-12079, and CVE-2018-13782).

Table 5 Features of several representative smart contract languages

Level	Language	Paradigm	Turing-complete?	Goal(s)
High-level	Solidity	Contract-oriented	Y(yes)	Realize smart contracts on Ethereum virtual machine
	Bamboo	Procedure	Y	Provide polymorphic contracts that can be executed in predefined order with enhanced auditability
	Obsidian	Object-oriented	-	Make it easier for programmers to write correct programs while leveraging a type system to provide strong safety guarantees
	Vyper	Procedural	N(no)	Provide more secure,simple and auditable smart contracts on EVM
	Flint	Procedure	Y	Write robust smart contracts on Ethereum
	Babbage	Mechanical	-	Expose how different parts interact and fit together
	Pyramid	Functional	Y	Scheme programming language targets at EVM
	DAML	Functional	N	Build composable applications on an abstract DA Ledger Model
Intermediate language	Yul	Procedure	Y	Designed to be a usable common denominator of various different backends in Ethereum((EVM 1.0, EVM 1.5 and eWASM)
	IELE	Rigister-based	Y	Designed to provide smart contracts security, formal verification, human-readability, and portability
Low-level	Ewasm	Stack-based	Y	Redesign of the Ethereum smart contract execution layer to facilitate smart contract ecology such as execution speed and supporting more programming languages
	Huff	Stack-based	Y	Write highly optimized EVM smart contracts enabling construction of EVM assembly macros and working of EVM explicitly

Vulnerability detection methods for smart contracts are somewhat similar to conventional softwares. But existing tools of traditional programs cannot be applied on contracts directly. Compared with traditional codes, the most remarkable feature of smart contracts is decentralized, which are stored and executed by lots of peers. So there are many unpredicted risks. For example, when a user sends a transaction to the network to invoke some contracts, he cannot ensure that the transaction will be run in the same state as the time of sending that transaction. Because, in the meanwhile, other transactions may have changed the contract state. Moreover, Ethereum contracts currently only support single thread, while mostly traditional codes support multi-threads. Therefore, there are many domain specific vulnerabilities for contracts, such as transaction order dependence and reentracy. Furthermore, smart contracts are programmed by customized language like solidity, which has particular features relative to traditional languages like C and C++. Hence it is impossible to leverage existing traditional program analysis tools straightly. Nevertheless, we can refer existing vulnerability detection knowledges and meanwhile consider contract features to design satisfied tools suitable to contracts. But many challenges need to be overcome, for example, getting master of contracts' features, finding an efficient way to figure out risks caused by these features, and guarding against these risks based on existing knowledges or their improvements.

Many solutions about contract vulnerability detection have been proposed. However, due to the complicated execution environment in smart contracts, dynamic code analysis may not be as effective as static code analysis, since dynamic code analysis requires more complex configuration to simulate the execution environment. Hence, most existing vulnerability detection methods are based on static analysis technologies (e.g., symbolic execution, data- or control- flow analysis and formal verification) [64, 71, 72]. But to reduce false positive or protect smart contracts in runtime, several dynamical methods are proposed, represented by fuzzing and code instrumentation [73–75]. What's more, some works adopt text comparison approaches of natural language processing or code similarity methods to detect known vulnerabilities [76–78]. Since Angelo and Salzer made a thorough review and comparison of existing tools for analyzing Ethereum smart contracts [79], and we concentrate more on security issues and exposing diverse solutions, consequently, we just list a few typical tools to demonstrate the fundamental principles of these methods.

- *Static analysis* can evaluate the potential risks of codes without execution, among symbolic execution is most popular for smart contracts. **Symbolic execution** substitutes program variable values with symbols, and performs symbolic computations when traversing codes. A symbolic expression will be produced when the judgment statement is satisfied, and finally a path condition will be determined for each path, which is a key factor to confirm path feasibility and vulnerability existence. Due to the brevity of smart contracts and the completeness of symbolic execution, this technology is very suitable to smart contracts. Oyente [64], for example, is designed to detect particular pitfalls within contracts (e.g., transaction-ordering dependence (TOD) and timestamp dependence). Specifically, after feeding both bytecodes and Blockchain global state, Oyente recovers CFG and symbolically executes all instructions recursively to derive each trace information with path constraint and some additional auxiliary data. Finally, Oyente uses such information to determine the existence of bugs. Oyente will also check whether two different traces have discrepant Ether flows. Other similar works chooses to complement symbolic execution with additional analysis technologies [72, 80–82] or only focus on a kind of bugs to enhance accuracy [71]. However, because of unsound completion of the system and the limitation of symbolic execution technology such as path explosion, no tool can guarantee totally accuracy. **Formal verification** is a promising technology to improve correctness by using mathematical language to comprehensively define the expected behavior of systems. Given a systematic formal specification, formal verification can determine whether each step of the implementation is consistent with the specification. Formal verification can com-

plement vulnerability detection methods, but it is complex and resource-intensive. Generally, formal verification is used only in security critical areas, such as spacecraft and operating systems. Increasingly, there has been focus on the use of formal verification for Blockchain applications, as evidenced in the literature [83–88], and the increasing number of companies providing verification-as-a-service (VaaS). Examples of VaaS include the CertiK platform, Secbit.io, and Securify.ch. The latter (Securify) [89], designed by researchers from ETH Zurich, is a security analyzer for Ethereum contracts to determine whether the contract behavior is safe or unsafe. Specifically, the tool can extract accurate semantic information based on contract's dependency relationships and then judges whether a property holds according to predetermined patterns involving both compliance and violation. Other approaches, such as those of [90] are designed to determine whether the contract templates satisfy correctness and the necessary properties. Clearly, this is an area of ongoing research.

- *Dynamical analysis* needs to execute codes but it has very low false positives relative to static analysis. ContractFuzzer [73] and Reguard [91] are typical fuzzing tools to detect vulnerabilities, which execute smart contracts by feeding enormous randomly generated inputs and then detect vulnerabilities based on execution logs. Due to the randomness of inputs, some extreme bug locations may consume too much time even bypass the detection. Although several researches focus on improving the quality of the inputs [92], this field is still in infancy. These tools can merely be used on pre-tests, unable to protect deployed contracts. Sereum [74] instruments EVM with vulnerability detection codes, which can dynamically check correctness. Once an execution violates predefined rules, the transaction will be aborted in real time. EVM* [75] is another similar detection tool focus on overflow and timestamp bugs. However, due to distributed execution, this method may introduce excessive overheads.
- *Code similarity* regards the features of existing vulnerability code base as the reference, and checks whether another program has the same bugs by assessing the similarity of their features. Liu et. al. conduct a serial of researches. They extract contract birthmarks involving both semantic and syntactic features [76] and then implement a tool - EClone [78] - to identify clones by comparing similarity between two contract birthmarks. They also attempt to use S-gram technology similar to N-gram to capture characters of contracts, providing basic elements for evaluating similarity [77].

  There are also other vulnerability approaches, for example, there are commercial providers that offer vulnerability detection as a service, similar to VaaS. Smart contract users or designers can submit their contracts to such providers to verify the contracts' security. However, the audit service may not be totally transparent and the customer may not have an easy way to determine how in-depth the analysis is. Also a few works try to apply machine learning into smart contract bug detection [93], but due to lack of vulnerability libraries, this method has many limitations until now. In a nutshell, there is no one-for-all-bug approach.

Bug bounty program is another way where vulnerabilities can be discovered and reported, for a financial reward. The bug bounty program is widely used by organizations such as Mozilla, Google, Facebook and Microsoft – see also [94]. Official bug bounty programs are not the only forum where vulnerabilities are being sold or traded. For example, there are underground or gray markets where such vulnerabilities are sold, in order to maximize the vulnerability identifier's profits. One key challenge in bug bounty is for the initiator to set a "fair" price that satisfies all parties. Seeking to address this, the Hydra project [95], for example, proposes an approach to model and administer bug bounties that incentives bug disclosure. Specifically, the approach is to exploit the price gap based on N-version programming, in order to motivate bug report even the reward is much lower than the asking price. By mathematically modeling and analyzing the market trend, one could arrive at a "fair" or reasonable price range. In the context of smart contracts, a bug in one version may not affect all the remaining versions. However, only when all versions have the same bug can then the bug be exploited. So there is an exploit gap between bug discovery and bug exploit, and this will also affect the pricing of the bug. In addition, there is a risk of holding on to the vulnerability for longer than required, as the value of the bug decreases significantly if it has already been reported or known. Unsurprisingly, designing efficient incentive mechanism in bug bounty and how to guarantee fair-exchange in bug bounty are ongoing research areas [96, 97]. For example, when a participant reports a bug, the bug bounty program must ensure that the participant will be financially rewarded as agreed. Tramer et al. [97], for example, considered using smart contracts for bug bounties, based on Sealed-Glass Proof (SGP). Such an approach permits safe verifiable computing using zero-knowledge. In other words, a reporter can prove to the buyer that he found bugs without leaking key information. However, the approach needs to utilize smart contracts and does not consider that the contract may also have vulnerabilities.

Contract upgrade is considered impossible in a Blockchain, due to its immutability property. This in itself is also a limitation for public Blockchain-based applications, for example it prevents the updating / patching of an incorrect or removal of a malicious / vulnerable contract. In permissioned Blockchain-based smart contracts (e.g., "fabric" series), however, the contract can be upgraded by authorized federated nodes. To make the public chain more flexibly, several work attempts to design new programming models or Blcockhain architectures to support contract code modification. Thus, contracts can be repaired after being found bugs.

Two "families" of patterns proposed by Zeppelin have emerged for upgradable contracts' design models, namely: data separation and delegatecall-based proxies – see also Fig. 3. Data separation pattern splits a contract into a data contract managing storage and a logic contract implementing special functions. The data contract is responsible for manipulating database, which cannot be upgraded. Therefore, it is designed

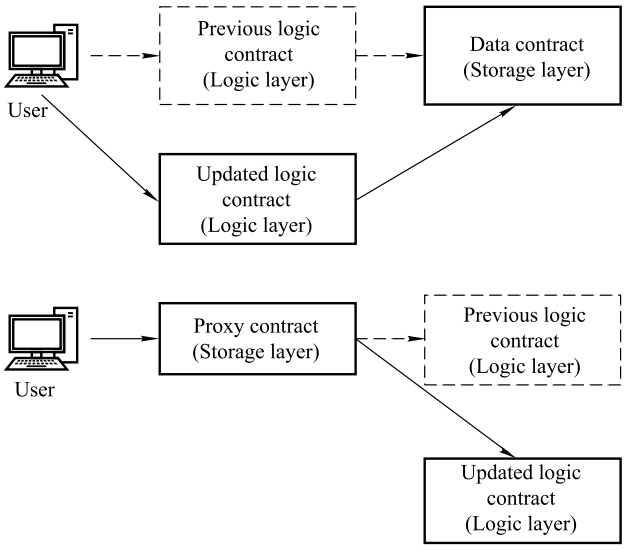

Fig. 3 Control flow of two design patterns (data separation pattern (top) and delegatecall-based proxy pattern (bottom))

as simple as possible. The logic contract can only access data indirectly by the data contract, and responsible for processing all required complex business logics, hence are more prone to bugs. However, it is designed to be upgradeable. Essentially, "upgrade" does not mean patching. Instead, it is replaced using a new logic contract. But, the Blockchain states can still be utilized again for the new contract by calling the data contract.

The delegatecall-based proxies model comprises a proxy contract holding data and a logic contract. The former code (without implementing concrete functionality) is made as simple as possible to prevent loophole. For each call, the proxy contract will send the logic contract context involving the original data using the "delegatecall" instruction. Once the current logic contract is determined to be limited or vulnerable, we can deploy a new contract by replacing the limited / vulnerable contract by changing the proxy contract pointer. When using proxy pattern, three storage models can be used to realize the separation of data and logic, namely: inherited storage, eternal storage and unstructured storage. All these models rely on low-level "delegatecall" (managing the returned data different from high-level delegatecall instruction). In inherited storage, the storage structure is built into the proxy contract, and the logic contract will inherit from it. In eternal storage, one uses a separate contract for the storage schema, and both proxy and logic contract are inherited from it. Unstructured storage is somewhat similar to inherited storage, but it does not require the logic contract to inherit any state variables associated with upgradeability. To sum up, both models have their own drawbacks, for instance, data separation's complexity makes it hard to ensure sound realization and proxy contract's old state data is difficult to be migrated.

Another alternative is the voting mechanism. Arbitrum [98], for example, designs a new blockchain architecture, that requires participants to reach an agreement off-chain so that miners only need to verify digital signatures on final results. In Arbitrum, contracts are realized as VMs that encode rules of corresponding contracts, which will be assigned a set of managers. The group of managers will be the regulator responsible for managing and executing of the contracts. Once a contract is found to be vulnerable, it can be upgraded as long as all honest managers approve during a dispute phase. Nebulas also supports contract upgrade by voting. Specifically, Nebulas allows state variables defined as "shared" to be accessed and modified directly by another contract. Thus, new contracts can inherit some necessary states of the old contract without additional data migration. It is worth noting that contract upgrade is not the same as contract modification. While deployed contracts on existing Blockchain platforms such as Ethereum or Nebulas cannot be changed, we can use some special design models to allow new contracts to use old data. This is somewhat equivalent to upgrading the contract. However, approaches undertaken in systems such as Arbitrum need to be closely examined to determine whether they conflict the tamper-proof property that underpins Blockchain.

Other alternative approaches include designing new contract secure development libraries and templates, such as several secure and gas-efficient libraries proposed by CryptoFin, to allow contract developers to create more secure and standard contracts. Fault-tolerant is another post-contract deployment remedial measure. For example, Hydra [95] improves upon conventional software fault-tolerant (N version programming (NVP)) to address contract bugs during runtime, as described earlier. One potential future research agenda is to explore other fault-tolerant approaches to enhance contract security. Allowing Blockchain to be modified [99] especially with fine-grained access control [100] is another underway novel research field, then the vulnerable deployed contracts may be patched timely.

6 Exploitable habitat

Located in a complicated environment, smart contract security is closely related to them. First, Ethereum contracts run in an isolated stack-based virtual machine (EVM), whose limitations can seriously affect the execution process. For example, the stack depth is limited to l024 bits, if the number of call exceeds this range, then the transaction will terminate abnormally. Extending outward a layer, transactions and contracts are propagated or executed both through the whole Blockchain network, attackers can leverage the Blockchain network or miners' mining strategies to hack contracts such as chain fork, transaction congestion, front-running, etc. Moreover, sometimes, smart contracts need to interact with the outside world, such as data feeding, so the trustworthy of data inputs may significantly influence the final results. To sum up, we will introduce the contract security issues introduced by execution environment (Section 1), Blockchain network (Section 2), and untrustworthy oracle (Section 3) three levels.

6.1 Isolated execution environment

EVM is specifically designed for executing smart contracts in a sandbox, providing two-way protection between physical hosts and smart contracts. However, some features of EVM can introduce potential risks. For example, EVM is actually semi-Turing complete, because its execution is limited by transaction gas, and gas-costly calculations such as the loop and recursion may be aborted due to out of gas (OOG) exception; EVM does not necessarily throw an exception when dealing with bugs such as arithmetic overflows, instead, it will quietly wrap it around

by modulo 2**256; besides, EVM with the Wei as the smallest unit does not support floating point numbers, resulting in imprecise computation results; unlike JVM that loads the executing function individually, EVM loads the entire smart contract as an opaque block and blindly executes from the first instruction, causing that calling an external contract is complicated; lack of standard libraries makes contract develop, deployment and execution more energy-intensive; moreover, since EVM parses the contract bytecode based on the definition of the contract Application Binary Interface (ABI), when attackers leverage the function [function transfer(address to, uint tokens) public returns (bool success)] in ERC-20 token contracts to transfer tokens with the length of first parameter less than 20 bytes, EVM will automatically regards significant bits of the next parameter as the supplement, moving the value of the second parameter to the right, resulting in an increase of the actual value of money transfer, namely, short address attack. If the contract has enough balances, it will be deducted more money than the expected.

To provide a more satisfiable execution environment, Ethereum community currently is dedicated into developing Ethereum WebAssembly (ewasm) targeted at reconstructing execution layer. Meanwhile, aimed at better describing EVM and automatically generating smart contract analysis tools easily, Hildenbrandt et al. provide a formal executable semantics of the EVM [101] - KEVM - based on K framework [102]. KEVM can find some contract defects that are untraceable to general detection tools, such as the recent Gridlock bug in Edgeware's lockdrop smart contracts, which is posed by programmers' wrong intuitive that the balance of a newly created account is always zero.

6.2 Malicious Blockchain network

All participants in Blockchain try to maximize their own profits. Hence, they are eager to leverage any shortcoming of Blockchain such as chain fork to hack smart contracts. A typical example is replay attack, namely, when a Blockchain forks into two isolated chains, at then, no matter EOAs or contracts hold same states on both chains. For example, to make up the loss caused by the DAO attack, after getting the majority votes, Ethereum is forked into ETH and ETC. So if a transaction extracting money from the victim contract is successfully executed in a fork chain, attackers can propagate the same transaction to another chain, then the contract will lose assets on both chains. To prevent this attack, Ethereum realizes reply attack protection by adopting different hash rules, see EIP-155 proposed by Vitalik Buterin.

Besides Blockchain changes caused by human can affect contract security, blockchain constrain rules can also be maliciously exploited. Ethereum uses "Block Gaslimit" to limit block size, but this value may be leveraged to launch smart contract DoS attack. For instance, if a contract iterates through an array to pay users, the attacker can invoke it to pay a bunch of receivers, but accumulated gas consumption of all transfer transactions may exceed block gas limit, resulting in the failure of all transfers. Contract owners can choose pull payments rather than push to mitigate this problem. Similarly, preferences of miners can also be an attack surface. Specifically, miners will first consider transactions with higher gas price. Given this policy, the attacker can send a computation-intensive transaction with extremely high gas price to achieve various kinds of attacks. On the one hand, he can perform front-running and his transaction will be preferentially processed by miners; on the other hand, due to complicated computation requirements, the miner cannot spare any energy to deal with other transactions, thus forming DoS attack again.

6.3 Unsafe external data

As smart contracts are applied in commercial setting, external data can be input into the contracts. As contracts are executed independently by each node on the Blockchain, any request to retrieve from an external data source is also independently executed. Since the external world is dynamical and potentially unsafe, it is challenging to ensure every node distributed in different countries will get a consistent response. Stock prices and weather forecast are two typical examples, where their values or readings may vary (e.g., between platforms / sources). If consistence cannot be enforced, then such inconsistent data can cause a Blockchain system to crash. Unsafe external data sources can be due to non-random random number or unreliable online data source.

6.3.1 Non-random random number

The security and stability of Blockchain rely heavily on random numbers, for example in leader selection, airdrop reward, and gambling decentralized applications (dApps). For most conventional centralized applications, a pseudo-random number generator (PRNG) may be adequate. However, Blockchain as a distributed system needs an approach to generate consistent and publicly verifiable random numbers, in order for fairness to be assured. However, determinacy of smart contracts contradicts indeterminacy of randomness. Several popular adopted solutions that are based on Blockchain inherent data such as Block hash, timestamp, and block number, can be easily manipulated or predicted by miners [103, 104].

The commit-and-reveal approach [105] is typically used in practice, where each participant will commit a commitment in the first step. Then, all commitments will be published and the random number will be constructed from all these committed numbers. This can be easily verified whether the process is fair. Examples where this approach is used include Randao, where anyone can participate and the random number is generated collaboratively by all participants (see Randao whitepaper). However, it involves too many participants and hence is expensive to manage and run. In fact, Randao has not been used in certain dApps until now. Another alternative approach is to extract information in a future block, in order to provide uniform randomness. Only at the pre-specified time will the random number be computed. In other words, when malicious user attempts to tamper with it, the result is already known by then. Examples of usage include gambling contracts [50] and Bitcoin-based protocols [106, 107]. However, the randomness can still be manipulated by miners through withholding valid blocks or reordering transactions. Countermeasures to prevent these attacks include the use of (efficient) verifiable delay functions (VDF) [108, 109], which is a hard-to-compute inherently-sequential function in a specified time. Thus, this prevents malicious miners from computing the random outputs in a timely

fashion.

There have also been attempts to use other cryptographic approaches to create randomness. For example, Algorand [110] uses a cipher to choose the leader that is tasked with block generation. In general, Algorand executes two major parts (i.e., block proposal and Byzantine Agreement (BA*) for reaching consensus on the proposal) iteratively. In the proposal step, Algorand will first select committee members randomly in a private and non-interactive way using the verified random function (VRF) [111] to protect their security, prior to selecting a proposer using the same way to mine a block. The entire committee will decide whether to grant it consent using the BA* consensus mechanism. Specifically, for member selection in the first step, each verifier computes a seed and a proof for round r that has been accepted in round r-1. The seed can be verified by the corresponding proof. A leader will be chosen based on these seeds. However, this mechanism relies on advanced cryptography technologies, which imposes a significant requirement on the hardware and is not easily applied in practice. Dfinity [112] adopts a bias-resistant threshold scheme to guarantee the adversary cannot abort the protocol. Specifically, it ensures that an attacker can neither hinder its creation nor predict the outcome. This mechanism can guarantee security in any Boneh-Lynn-Shacham (BLS)-based scheme as long as the attacker does not have more than a half of stakes (i.e., the majority attack). Both Algorand and Dfinity adopt the VRF method. Another example is Ouroboros [113], a composable proof-of-stake Blockchain with dynamic availability. Ouroboros also utilizes VRF to generate the slot leader. In summary, VRF is a relatively promising technology to create random number in distributed system. In Thunderella [114], a nonce is set by the miner for next r rounds. Combined with the nonce and other parameters, the proposer's public key will be hashed and then the hash value will be confirmed whether it is less than a difficulty value. The proposer will be chosen as a leader as long as it satisfies this inequation. However in this design, the same nonce is reused for subsequent rounds. Thus, an attacker may be able to predict the proposers in advance.

6.3.2 Unreliable online data source

A smart contract works as an arbitrary institution, in the sense that when it is asked to make a decision, relevant data is required. However, it is challenging to ensure that data is correct and consistent. For instance, if the contract needs static data from a specific source, one cannot guarantee that the data at the point of access has not been tainted / compromised. If dynamical data is required, due to time and geographical differences in unstable / uncertain networks, it is also challenging to ensure all nodes obtain consistent and correct data. In general, data can be sent directly by the transaction senders such as third-party providers, but there is a risk that they may attempt to influence the decision by manipulating critical data. Therefore, we require a trusted relayer, who transfers data that is reliable and correct. Trusted relayers are also referred to as oracles, which are in the business of providing external data for smart contracts.

Currently, oracles can be broadly categorized into trusted third-party oracles and decentralized oracles. Oraclize and Town Crier [115] are two examples of trusted third-party oracles. Oraclize is widely used as a data carrier to help contracts fetch external data by leveraging Amazon with TLS notary-based proofs to guarantee data security. Specifically, Amazon is responsible for storing the proofs. Thus, verifying the signature / proof AWS provides, smart contracts can determine authenticity of the retrieved data. Although it provides secure web contents, it suffers from a number of drawbacks. For example, attackers may attempt to tamper the proofs by compromising the AWS oracle. Town Crier focuses on the utilization of Intel Software Guard Extensions (SGX) to verify data correctness. SGX will attach the data a signature if it passes verification, then will send it to the smart contract. The contract only needs to check the signature to judge correctness of the data. Such an approach, however, defrays from the essence of Blockchain as we should rely on a third-party.

On the other end of the spectrum, we have decentralized oracles such as ChainLink. ChainLink is a decentralized oracle network [116], where anyone can apply to become a data provider once the requirements and application conditions have been posted. Data purchasers can manually sort, filter and select oracles via off-chain listing service, which runs based on the reputation maintained on-chain along with a set of data gathered from prior contracts logs. ChainLink avoids failure of single point, but to some extent we need to rely on the architecture of ChainLink system running normally. However, once this system crashes, the reliability of relayed data cannot be ensured.

7 Challenges and future work

7.1 Challenges and potential solutions

In this paper, we systematically surveyed existing smart contract security research, next, we will introduce related challenges and potential solutions. For example, in abnormal contracts, it is challenging to distinguish between CSCs, normal smart contracts, and contracts that are been criminally exploited. This is partly because CSCs are essentially the same as normal smart contracts, including how they are triggered, invoked, stored, etc. In addition, there are many different programming languages and compilers, resulting in complex internal call relationships. Therefore, it is not realistic to design a one-size-fit-all reverse engineering tool to transfer bytecodes to source codes or human readable form. It is also challenging to recreate the intent from the bytecodes alone due to a lack of contextual information. Even in the event that we identify the intention of a smart contract, it is also nontrivial to determine whether it is associated with some illegal activities. This is also partly due to lack of a clear definition for contracts intention legality. A potential solution may be to *use machine learning to learn the differences between CSCs, normal smart contracts, and contracts that are been criminally exploited, so that we can better detect abnormal contracts*.

Instances relating to exorbitant costs are also challenging to detect. For example, there are many low-level instructions and applications, and defining or figuring out costly patterns can be tricky. Existing gas-costly detection tools are not entirely effective; thus, *designing gas-efficient templates* may help reduce the potential for costly contracts.

Another potential research agenda is to *design a more secure*

programming language, without affecting the performance of the smart contracts. For example, in a smart contract context, since the language will be executed by a VM, the design should therefore take into consideration the compiler. However, there will be instances where vulnerabilities exist due to implementation or design flaws.

Research on smart contract vulnerability is also crucial, particularly as the usage of smart contracts broadens. There is, therefore, the need to *design automated tools to identify and exploit vulnerabilities in existing smart contracts, for the diverse smart contract languages and platforms*.

In the event that a vulnerability is identified, we also need a secure way of performing upgrade / patching. Therefore, we should explore *designing ways to upgrade the smart contract that holds the data state, without affecting the security and trust of the underpinning Blockchain*.

Due to the complexity of EVM and blockchain network, it is tricky to define the semantics of EVM, simulate the whole system or consider comprehensive attack surface of the survive environment. Given the lessons of EVM 1.0, maybe a better blockchain ecology from technical improvement to incentive mechanism constrains can mitigate this problem. As discussed earlier, there is also a need to *design secure random number generator or PRNG, which is suitable for Blockchain applications* (e.g., sufficiently lightweight).

Recall that smart contracts are required to be stored by all full nodes and each execution is repeated across the total Blockchain system. Therefore, a potential threat exists that codes, inputs, and outputs are exposed to the public. Even if the variable in smart contracts are private, it is only so in the Blockchain. Therefore, there is a need to *design effective approaches, such as homomorphic encryption techniques, to protect private data yet allowing the data to be publicly verified*.

7.2 Future work

Besides future research areas about addressing hot problems mentioned above, some overlooked but important security aspects are worth considering as well. *Smart contract issues on other platforms* owns much less attentions than Ethereum. Other Blockchain platforms of interest include Hyperledger Fabric and EOS. For example, in fabric, smart contracts (chaincodes) running in Docker may escape from the container; thus, resulting in malicious network access. The associated programming language, go, can also be the cause of vulnerabilities in the smart contracts, and EOS is also frequently targeted by malicious actor (see "Eos smart contract security best practices" proposed by Slow Mist team).

Exploring vulnerabilities beyond contract codes. Future research should extend beyond the existing that only focus on contract codes. On the one hand, most times, smart contracts do not merely mean codes and involve some other meanings such as decentralized applications (DAPPs) or ERC tokens. Not only do they involve codes, but also external front-end applications. Under this context, new issues may generate. For instance, the interaction process between the font-end and the contract codes may also be controlled and can be an attack vector. As the number of various DAPPs soars, *researches on DAPP security are extremely significant*. On the other hand, smart contracts are different from traditional programs, which have specific business logics, *it is important to guarantee their security by ensure the safety of the logics*, namely making contracts behaviors identical to the expected. But now contract logic defects such as improper access control are few touched. *Combining additional informations provided by the front-end, detecting logic vulnerabilities may be easier*. In addition, *smart contracts support reuse, and hence flaws in the reused codes can have a far-reaching consequence. Hence, this necessitates the ongoing need to design new (automated) tools and approaches (e.g., using machine learning) to identify previously unknown vulnerabilities*.

To improve performance, there have been attempts to facilitate smart contract concurrency by changing its original execution modes [117–119]. While smart contract concurrency enhances effectiveness to some extent, we have to closely examine the security implications. For example, running on complex distributed Blockchain network, *we need to consider race condition bugs such as reentrancy vulnerability, when supporting concurrency*.

Machine learning has been shown to be relatively efficient in identifying vulnerability problems in conventional software [120–123]. Similarly, *we posit its potential in learning normal semantics of smart contracts and identifying previously unknown bugs / vulnerabilities, or analyzing existing bug patterns to check contract security. Moreover, it can also be used to check complex logic bugs and verify whether smart contract behaviors are expected*. One key challenge in using machine learning is the lack of training datasets.

Formal verification on smart contracts should be attached importance. Until now, there are only a handful of works in this area. But comparing with traditional program analysis technologies, formal verification is more suitable to check whether smart contract codes satisfy expected functions. Although getting master of this technology is difficult, it will benefit contract security significantly.

Generating smart contracts that can escape from common vulnerability detection tools is also an interesting area. Although hiding bugs is not a good way to secure contracts, but it someway somehow can enhance them security. In turn, the accumulated knowledges can facilitate designing more robust program analysis tools.

Performance is an issue that has yet to be discussed in this paper so far, however it is critical. For example, unlike a typical payment system such as Visa (that reportedly processes 1,200 to 56,000 transactions per second (TPS)), Blockchain's throughput does not scale well (e.g., Bitcoin 7 TPS, and Ethereum 25 TPS). While in theory Ethereum can pack unlimited transactions due to unbounded block size, in practice the throughput is far less due to network performance and the block gas limit (GasLimit). However, improving performance shouldn't sacrifice security. For example, hot researches about execution in SGX off chain, state channel, and cross chain focus more on performance. But it is noted that, any change may introduce potential risks. *Digging out contract issues or proving its safety in these new contexts is also a meaningful and necessary work*.

Because all smart contracts are deployed on a decentralized and public environment, so contracts are supported to reuse

or invoke each other. Developers can copy on-chain contracts in the development stage, or directly call them in Blockchain. Hence, to reduce the development or deployment cost, the phenomenon of contract code reuse becomes more and more popular. So *contract quality assessment is also a significant research area*. Currently, most state-of-art work focuses on vulnerability prevention and detection, but vulnerability is just one of metrics of the contract quality. For example, gas usage, code size, design mode etc, are both key metrics. More comprehensive and precise metrics are waited to be explored. Also efficient assessment technologies of contract quality are expected. Further, software defect detection, an almost neglected direction, is somewhat analogous to quality assessment, which can find more potential improper places.

Last but not least, as Blockchain and distributed ledger networks are exploding by the day, interaction between multiple Blockchains is necessary, so there is a need to develop cross-chain smart contracts. There has been merely a little work until now, but it is definitely a popular area in the future. It is noted that, under this kind of scenario, the attack surface of smart contracts is more enlarged, because it involves more different Blockchain ecologies rather than just one; therefore, when developing cross-chain contracts, security issues should be attached much more importance.

8 Conclusion

Blockchain-based smart contracts are moving beyond hype to real-world deployment. Due to its usage to support high value financial transactions, ensuring its security is crucial. Therefore, in this paper, we surveyed existing literature on Blockchain-based smart contracts from its birth (July, 2015) to this manuscript writing (July, 2019) and presented a taxonomy focusing on the various security challenges and potential mitigation strategies. Moreover, some promising research areas are provided to promote contract security.

Acknowledgements This work was supported by the National Key Research and Development (R&D) Plan of China (2019YFB2101700), the Science and Technology Program of Guangzhou (201902020016), the Shenzhen Fundamental Research Program (JCYJ20170413114215614), the Guangdong Provincial Science and Technology Plan Project (2017B010124001), and the Guangdong Provincial Key R&D Plan Project (2019B010139001).

References

1. Chaum D. Blind signatures for untraceable payments. In: Proceedings of the 2nd Annual International Cryptology Conference. 1982, 199–203
2. Chaum D, Fiat A, Naor M. Untraceable electronic cash. In: Proceedings of the 8th Annual International Cryptology Conference. 1988, 319–327
3. Schoenmakers B. Security aspects of the ecash^{tm} payment system. In: State of the Art in Applied Cryptography. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1998, 338–352
4. Rivest R L. Peppercoin micropayments. In: Proceedings of the 8th International Conference on Financial Cryptography. 2004, 2–8
5. Satoshi N. Bitcoin: a peer-to-peer electronic cash system. 2008
6. Tschorsch F, Scheuermann B. Bitcoin and beyond: a technical survey on decentralized digital currencies. IEEE Communications Surveys Tutorials, 2016, 18(3): 2084–2123
7. Conti M, Kumar E S, Lal C, Ruj S. A survey on security and privacy issues of bitcoin. IEEE Communications Surveys Tutorials, 2018, 20(4): 3416–3452
8. Khalilov M C K, Levi A. A survey on anonymity and privacy in bitcoin-like digital cash systems. IEEE Communications Surveys Tutorials, 2018, 20(3): 2543–2585
9. Ferrag M A, Derdour M, Mukherjee M, Derhab A, Maglaras L, Janicke H. Blockchain technologies for the internet of things: research issues and challenges. IEEE Internet of Things Journal, 2018, 6(2): 2188–2204
10. Sankar L S, Sindhu M, Sethumadhavan M. Survey of consensus protocols on blockchain applications. In: Proceedings of the 4th IEEE International Conference on Advanced Computing and Communication Systems. 2017, 1–5
11. Nguyen G T, Kim K. A survey about consensus algorithms used in blockchain. Journal of Information Processing Systems, 2018, 14(1): 101–128
12. Zhu L, Wu Y, Gai K, Choo K R. Controllable and trustworthy blockchain-based cloud data management. Future Generation Computer system, 2019, 91: 527–535
13. Esposito C, Santis A D, Tortora G, Chang H, Choo K R. Blockchain: a panacea for healthcare cloud-based data security and privacy. IEEE Cloud Computing, 2018, 5(1): 31–37
14. Gai K, Choo K R, Zhu L. Blockchain-enabled reengineering of cloud Datacenters. IEEE Cloud Computing, 2018, 5(6): 21–25
15. Lin C, He D, Huang X, Choo K R, Vasilakos A V. Bsein: a blockchain-based secure mutual authentication with fine-grained access control system for industry 4.0. Journel of Network and Computer Applications, 2018, 116: 42–52
16. Conoscenti M, Vetro A, De Martin J C. Blockchain for the internet of things: a systematic literature review. In: Proceedings of the 13th IEEE/ACS International Conference of Computer Systems and Applications. 2016, 1–6
17. Hassan M U, Rehmani M H, Chen J. Privacy preservation in blockchain based iot systems: integration issues, prospects, challenges, and future research directions. Future Generation Computer Systems, 2019, 97: 512–529
18. Taylor P J, Dargahi T, Dehghantanha A, Parizi R M, Choo K R. A systematic literature review of blockchain cyber security. Digital Communications and Networks, 2020, 6(2): 147–156
19. Xie J, Tang H, Huang T, Yu F R, Xie R, Liu J, Liu Y. A survey of blockchain technology applied to smart cities: research issues and challenges. IEEE Communications Surveys & Tutorials, 2019, 21(3): 2794–2830
20. Yang R, Yu F R, Si P, Yang Z, Zhang Y. Integrated blockchain and edge computing systems: a survey, some research issues and challenges. IEEE Communications Surveys & Tutorials, 2019, 21(2): 1508–1532
21. Buterin V. A next-generation smart contract and decentralized application platform. White Paper, 2014, 3(37): 1–36
22. Ronen E, Shamir A, Weingarten A, O'Flynn C. IoT goes nuclear: creating a zigbee chain reaction. In: Proceedings of the 38th IEEE Symposium on Security and Privacy. 2017, 195–212
23. Vasisht D, Kapetanovic Z, Won J, Jin X, Chandra R, Sinha S N, Kapoor A, Sudarshan M, Stratman S. Farmbeats: an IoT platform for data-driven agriculture. In: Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation. 2017, 515–529
24. Azaria A, Ekblaw A, Vieira T, Lippman A. Medrec: using blockchain for medical data access and permission management. In: Proceedings of the 2nd International Conference on Open and Big Data. 2016, 25–30
25. Yue X, Wang H, Jin D, Li M, Jiang W. Healthcare data gateways: found healthcare intelligence on blockchain with novel privacy risk control. Journal of Medical Systems, 2016, 40(10): 218
26. Chen L, Lee W K, Chang C, Choo K R, Zhang N. Blockchain based searchable encryption for electronic health record sharing. Future Generation Computer Systems, 2019, 95: 420–429
27. McGhin T, Choo K R, Liu C Z, He D. Blockchain in healthcare appli-

cations: research challenges and opportunities. Journal of Network and Computer Applications, 2019, 135(1): 62–75

28. Huckle S, Bhattacharya R, White M, Beloff N. Internet of things, blockchain and shared economy applications. In: Proceedings of the 7th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2016)/The 6th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2016)/Affiliated Workshops. 2016, 461–466
29. Yao Q. A systematic framework to understand central bank digital currency. Science China Information Sciences, 2018, 61(3): 033101
30. Liang J, Han W, Guo Z, Chen Y, Cao C, Wang X S, Li F. DESC: enabling secure data exchange based on smart contracts. Science China Information Sciences, 2018, 61(4): 049102
31. Matsumoto S, Reischuk R M. IKP: turning a PKI around with decentralized automated incentives. In: Proceedings of the 38th IEEE Symposium on Security and Privacy. 2017, 410–426
32. Chen J, Yao S, Yuan Q, He K, Ji S, Du R. Certchain: public and efficient certificate audit based on blockchain for TLS connections. In: Proceedings of the 2018 IEEE International Conference on Computer Communications. 2018, 2060–2068
33. Chase M, Meiklejohn S. Transparency overlays and applications. In: Proceedings of the 23th ACM SIGSAC Conference on Computer and Communications Security. 2016, 168–179
34. Szabo N. Formalizing and securing relationships on public networks. First Monday, 1997, 2(9): 1–21
35. Paul A, Ahmad A, Khan M, Jeon G. Smart contract's interface for user centric business model in blockchain. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing. 2019, 709–714
36. Siano P, Marco G De, Rolán A, Loia V. A survey and evaluation of the potentials of distributed ledger technology for peer-to-peer transactive energy exchanges in local energy markets. IEEE Systems Journal, 2019, 13(3): 3454–3466
37. Castillo M. The dao attacked: code issue leads to 60 million ether theft. see Coindesk Website, 2020
38. Reddit. Smartbillions lottery contract just got hacked. see Reddit Website, 2020
39. Petrov S. Another parity wallet hack explained. see Medium Website, 2020
40. Slow Mist. Eth dapp hack events. see Slow Mist Hacked Website, 2020
41. Bartoletti M, Pompianu L. An empirical analysis of smart contracts: platforms, applications, and design patterns. In: Proceedings of the 21st International Conference on Financial Cryptography and Data Security. 2017, 494–509
42. Castro M, Liskov B. Practical byzantine fault tolerance. In: Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation. 1999, 173–186
43. Sukhwani H, Martínez J M, Chang X, Trivedi K S, Rindos A. Performance modeling of PBFT consensus process for permissioned blockchain network (hyperledger fabric). In: Proceedings of the 36th IEEE Symposium on Reliable Distributed Systems. 2017, 253–255
44. David B, Gazi P, Kiayias A, Russell A. Ouroboros praos: an adaptively-secure, semi-synchronous proof-of-stake blockchain. In: Proceedings of the 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques. 2018, 66–98
45. Badertscher C, Gazi P, Kiayias A, Russell A, Zikas V. Ouroboros genesis: composable proof-of-stake blockchains with dynamic availability. In: Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security. 2018, 913–930
46. Petersen K, Feldt R, Mujtaba S, Mattsson M. Systematic mapping studies in software engineering. In: Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering. 2008
47. Pahl C, Brogi A, Soldani J, Jamshidi P. Cloud container technologies: a state-of-the-art review. IEEE Transactions on Cloud Computing, 2019, 7(3): 677–692
48. Bonneau J, Miller A, Clark J, Narayanan A, Kroll J A, Felten E W. Sok: research perspectives and challenges for bitcoin and cryptocurrencies. In: Proceedings of the 36th IEEE Symposium on Security and Privacy. 2015, 104–121
49. Alharby M, van Moorsel A. Blockchain-based smart contracts: a systematic mapping study. 2017, arXiv preprint arXiv:1710.06372
50. Atzei N, Bartoletti M, Cimoli T. A survey of attacks on ethereum smart contracts (sok). In: Proceedings of the 6th International Conference on Principles of Security and Trust. 2017, 164–186
51. Wang S, Ouyang L, Yuan Y, Ni X, Han X, Wang F Y. Blockchain-enabled smart contracts: architecture, applications, and future trends. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2019, 49(11): 2266–2277
52. Juels A, Kosba A E, Shi E. The ring of gyges: investigating the future of criminal smart contracts. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security. 2016, 283–295
53. Kwon Y, Kim D, Son Y, Vasserman E Y, Kim Y. Be selfish and avoid dilemmas: fork after withholding (FAW) attacks on bitcoin. In: Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security. 2017, 195–209
54. Eyal I. The miner's dilemma. In: Proceedings of the 36th IEEE Symposium on Security and Privacy. 2015, 89–103
55. Velner Y, Teutsch J, Luu L, Smart contracts make bitcoin mining pools vulnerable. In: Proceedings of the 21st International Conference on Financial Cryptography and Data Security. 2017, 298–316
56. McCorry P, Hicks A, Meiklejohn S. Smart contracts for bribing Miners. IACR Cryptology ePrint Archive, 2018, 2018: 581
57. Wang Y, Bracciali A, Li T, Li F, Cui X, Zhao M. Randomness invalidates criminal smart contracts. Information Science, 2019, 477: 291–301
58. Torres C F, Steichen M. The art of the scam: demystifying honeypots in ethereum smart contracts. In: Proceedings of the 28th USENIX Security Symposium. 2019
59. Zhou Y, Kumar D, Bakshi S, Mason J, Miller A, Bailey M. Erays: reverse engineering ethereum's opaque smart contracts. In: Proceedings of the 27th USENIX Security Symposium. 2018, 1371–1385
60. Schwarz B, Debray S K, Andrews G R. Disassembly of executable code revisited. In: Proceedings of the 9th Working Conference on Reverse Engineering. 2012, 45–54
61. Grech N, Brent L, Scholz B, Smaragdakis Y. Gigahorse: thorough, declarative decompilation of smart contracts. In: Proceedings of the 41st International Conference on Software Engineering. 2019, 1176–1186
62. Parizi R M, Dehghantanha A, Choo R. A singh, empirical vulnerability analysis of automated smart contracts security testing on blockchains. In: Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering. 2018, 103–113
63. Chen T, Li X, Luo X, Zhang X. Under-optimized smart contracts devour your money. In: Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering. 2017, 442–446
64. Luu L, Chu D, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security. 2016, 254–269
65. Chen T, Li X, Wang Y, Chen J, Li Z, Luo X, Au M H, Zhang X. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In: Proceedings of the 13th International Conference on Information Security Practice and Experience. 2017, 3–24
66. Luu L, Teutsch J, Kulkarni R, Saxena P. Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015, 706–719
67. Li Y. Finding concurrency exploits on smart contracts. In: Proceedings

of the 41st International Conference on Software Engineering: Companion Proceedings. 2019, 144–146
68. Coblenz M J. Obsidian: a safer blockchain programming language. In: Proceedings of the 39th International Conference on Software Engineering. 2017, 97–99
69. Schrans F, Eisenbach S, Drossopoulou S. Writing safe smart contracts in flint. In: Proceedings of the 2nd International Conference on Art, Science, and Engineering of Programming. 2018, 218–219
70. Schrans F, Hails D, Harkness A, Drossopoulou S, Eisenbach S. Flint for safer smart contracts. 2019, arXiv preprint arXiv:1904.06534
71. Torres C F, SchĀijtte J, State R. Osiris: hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference. 2018
72. Nikolic I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Conference on Computer Security Applications. 2018
73. Jiang B, Liu Y, Chan W K. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018, 259–269
74. Rodler M, Li W, Karame G O, Davi L. Sereum: protecting existing smart contracts against re-entrancy attacks. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium. 2019
75. Ma F, Fu Y, Ren M, Wang M, Jiang Y, Zhang K, Li H, Shi X. EVM*: from offline detection to online reinforcement for ethereum virtual machine. In: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering. 2019, 554–558
76. Liu H, Yang Z, Jiang Y, Zhao W, Sun J. Enabling clone detection for ethereum via smart contract birthmarks. In: Proceedings of the 27th International Conference on Program Comprehension. 2019, 105–115
77. Liu H, Liu C, Zhao W, Jiang Y, Sun J. S-gram: towards semantic-aware security auditing for ethereum smart contracts. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018, 814–819
78. Liu H, Yang Z, Liu C, Jiang Y, Zhao W, Sun J, Eclone: detect semantic clones in ethereum via symbolic transaction sketch. In: Proceedings of the 26th 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018, 900–903
79. Angelo M D, Salzer G. A survey of tools for analyzing ethereum smart Contracts. In: Proceedings of IEEE International Conference on Decentralized Applications and Infrastructures. 2019
80. Krupp J, Rossow C. teEther: gnawing at ethereum to automatically exploit smart contracts. In: Proceedings of the 27th USENIX Security Symposium. 2018, 1317–1333
81. Mossberg M, Manzano F, Hennenfent E, Groce A, Grieco G, Feist J, Brunson T, Dinaburg A. Manticore: a user-friendly symbolic execution framework for binaries and smart contracts. In: Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering. 2019
82. Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: analyzing safety of smart contracts. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium. 2018
83. Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N, Béguelin S Z. Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. 2016, 91–96
84. Idelberger F, Governatori G, Riveret R, Sartor G. Evaluation of logic-based smart contracts for blockchain systems. In: Proceedings of the 10th International Symposium on Rule Technologies, Research, Tools, and Applications. 2016, 167–183
85. Hildenbrandt E, Saxena M, Rodrigues N, Zhu X, Daian P, Guth D, Moore B M , Park D, Zhang Y, Stefanescu A, Rosu G. KEVM: a complete formal semantics of the ethereum virtual machine. In: Proceedings of the 31st IEEE Computer Security Foundations Symposium. 2018, 204–217
86. Park D, Zhang Y, Saxena M, Daian P, Rosu G. A formal verification tool for ethereum VM bytecode. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018, 912–915
87. Ahrendt W, Pace G J, Schneider G. Smart contracts: a killer application for deductive source code verification. In: Müller P, Schaefer I. eds. Principled Software Development. Springer, Cham, 2018, 1–18
88. Ellul J, Pace G J. Runtime verification of ethereum smart contracts. In: Proceedings of the 14th European Dependable Computing Conference. 2018, 158–163.
89. Tsankov P, Dan A M, Drachsler-Cohen D, Gervais A, Bünzli F, Vechev M T. Securify: practical security analysis of smart contracts. In: Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security. 2018, 67–82
90. Bai X, Cheng Z, Duan Z, Hu K. Formal modeling and verification of smart contracts. In: Proceedings of the 7th International Conference on Software and Computer Applications. 2018, 322–326
91. Liu C, Liu H, Cao Z, Chen Z, Chen B, Roscoe B. Reguard: finding reentrancy bugs in smart contracts. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. 2018, 65–68
92. Wüstholz V, Christakis M. Harvey: a greybox fuzzer for smart contracts. 2019, arXiv preprint arXiv:1905.06944
93. Tann W J, Han X J, Gupta S S, Ong Y. Towards safer smart contracts: a sequence learning approach to detecting vulnerabilities. 2018, arXiv preprint arXiv:1811.06632
94. Finifter M, Akhawe D, Wagner D A. An empirical study of vulnerability rewards programs. In: Proceedings of the 22nd USENIX Security Symposium. 2013, 273–288
95. Breidenbach L, Daian P, Tramèr F, Juels A. Enter the hydra: towards principled bug bounties and exploit-resistant smart contracts. In: Proceedings of the 27th USENIX Security Symposium. 2018, 1335–1352
96. Banasik W, Dziembowski S, Malinowski D. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: Proceedings of the 21st European Symposium on Research in Computer Security. 2016, 261–280
97. Tramèr F, Zhang F, Lin H, Hubaux J, Juels A, Shi E. Sealed-glass proofs: using transparent enclaves to prove and sell knowledge. In: Proceedings of the 2nd IEEE European Symposium on Security and Privacy. 2017, 19–34
98. Kalodner H A, Goldfeder S, Chen X, Weinberg S M, Felten E W. Arbitrum: scalable, private smart contracts. In: Proceedings of the 27th USENIX Security Symposium. 2018, 1353–1370
99. Ateniese G, Magri B, Venturi D, Andrade E R. Redactable blockchain - or - rewriting history in bitcoin and friends. In: Proceedings of 2017 IEEE European Symposium on Security and Privacy. 2017, 111–126
100. Derler D, Samelin K, Slamanig D, Striecks C. Fine-grained and controlled rewriting in blockchains: chameleon-hashing gone attribute-based. In: Proceedings of the 26th Annual Network and Distributed System Security Symposium. 2019
101. Hildenbrandt E, Saxena M, Rodrigues N, Zhu X, Daian P, Guth D, Moore B M, Park D, Zhang Y, Stefanescu A, Rosu G. KEVM: a complete formal semantics of the ethereum virtual machine. In: Proceedings of the 31st IEEE Computer Security Foundations Symposium. 2018, 204–217
102. Rosu G, Serbanuta T. An overview of the K semantic framework. Journal of Logic and Algebraic Programming, 2010, 79(6): 397–434
103. Chatterjee K, Goharshady A K, Pourdamghani A. Probabilistic smart

contracts: secure randomness on the blockchain. In: Proceedings of IEEE International Conference on Blockchain and Cryptocurrency. 2019
104. Pierrot C, Wesolowski B. Malleability of the blockchain's entropy. Cryptography and Communications, 2018, 10(1): 211–233
105. Cachin C, Kursawe K, Shoup V. Random oracles in constantinople: practical asynchronous byzantine agreement using cryptography. Journal of Cryptology, 2005, 18(3): 219–246
106. Bonneau J, Narayanan A, Miller A, Clark J, Kroll J A, Felten E W. Mixcoin: anonymity for bitcoin with accountable mixes. In: Proceedings of the 18th International Conference on Financial Cryptography and Data Security. 2014, 486–504
107. Garman C, Green M, Miers I, Rubin A D. Rational zero: economic security for zerocoin with everlasting anonymity. In: Proceedings of the 18th International Conference on Financial Cryptography and Data Security. 2014, 140–155
108. Bünz B, Goldfeder S, Bonneau J. Proofs-of-delay and randomness beacons in ethereum. IEEE Security and Privacy on the Blockchain (IEEE S&B), 2017
109. Lenstra A K, Wesolowski B. A random zoo: sloth, unicorn, and trx IACR Cryptology ePrint Archive, 2015, 2015: 366
110. Gilad Y, Hemo R, Micali S, Vlachos G, Zeldovich N. Algorand: scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles. 2017, 51–68
111. Micali S, Rabin M O, Vadhan S P. Verifiable random functions. In: Proceedings of the 40th Annual Symposium on Foundations of Computer Science. 1999, 120–130
112. Hanke T, Movahedi M, Williams D. DFINITY technology overview series, consensus system. 2018, arXiv preprint arXiv:1805.04548
113. Badertscher C, Gazi P, Kiayias A, Russell A, Zikas V. Ouroboros genesis: composable proof-of-stake blockchains with dynamic availability. In: Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security. 2018, 913–930
114. Pass R, Shi E. Thunderella: Blockchains with optimistic instant confirmation. In: Proceedings of the 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques. 2018, 3–33
115. Zhang F, Cecchetti E, Croman K, Juels A, Shi E. Town crier: an authenticated data feed for smart contracts. In: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security. 2016, 270–282
116. Ellis S, Juels A, Nazarov S. Chainlink—a decentralized oracle Network. 2017
117. Sergey I, Hobor A. A concurrent perspective on smart contracts. In: Proceedings of the 21st International Conference on Financial Cryptography and Data Security. 2017, 478–493
118. Dickerson T D, Gazzillo P, Herlihy M, Koskinen E. Adding concurrency to smart contracts. In: Proceedings of the 36th ACM Symposium on Principles of Distributed Computing. 2017, 303–312
119. Zhang A, Zhang K. Enabling concurrency on smart contracts using multiversion ordering. In: Proceedings of the 2nd International Joint Conference on Web and Big Data. 2018, 425–439
120. Li Z, Zou D, Xu S, Ou X, Jin H, Wang S, Deng Z, Zhong Y. Vuldeepecker: a deep learning-based system for vulnerability detection. In: Proceedings of the 25th Annual Network and Distributed System Security Symposium. 2018
121. Russell R, Kim L, Hamilton L, Lazovich T, Harer J, Ozdemir O, Ellingwood P, McConley M. Automated vulnerability detection in source code using deep representation learning. In: Proceedings of the 17th IEEE International Conference on Machine Learning and Applications. 2018, 757–762
122. Liu B, Huo W, Zhang C, Li W, Li F, Piao A, Zou W. αdiff: cross-version binary code similarity detection with DNN. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018, 667–678
123. White M, Tufano M, Vendome C, Poshyvanyk D. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016, 87–98

Zeli Wang is a PhD candidate at Huazhong University of Science and Technology (HUST), China. Her main research topics are blockchain and smart contract security.

Hai Jin received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), China in 1994. He is currently Chair Professor of computer science and engineering with HUST, China. He is the also Chief Scientist of the National 973 Basic Research Program Project of Virtualization Technology of Computing System. He has co-authored 22 books and published over 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security. He is a fellow of CCF and a member of the ACM.

Weiqi Dai received the PhD degree in computer science and technology from Huazhong University of Science and Technology (HUST), China. He is an assistant professor in school of cyber science and engineering at HUST, China. His expertise and research interests include blockchain, cloud computing security, trusted computing, virtualization technology, and trusted SDN.

Kim-Kwang Raymond Choo received his PhD in Information Security from Queensland University of Technology, Australia. He currently holds the Cloud Technology Endowed Professorship at the University of Texas, USA at San Antonio and is an associate professor at the University of South Australia, Australia. He was named one of 10 Emerging Leaders in the Innovation category of The Weekend Australian Magazine/Microsoft's Next 100 series in 2009, and is the recipient of various awards including the British Computer Society's Wilkes Award and the Fulbright Scholarship. He is a fellow of the Australian Computer Society, and a Senior Member of IEEE.

Deqing Zou received the PhD degree from the Huazhong University of Science and Technology (HUST), China in 2004. He is currently a professor in school of cyber science and engineering at HUST, China. He has applied almost 20 patents, published two books, one is theXen virtualization Technologies and the other is Trusted Computing Technologies and Principles, and published over 50 high-quality papers. His main research interests include system security, trusted computing, virtualization, and cloud security. He has served as the PC member/PC chair of over 40 international conferences.