This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

Challenges and Common Solutions in Smart Contract Development

N. Kannengießer, S. Lins, C. Sander, K. Winter, H. Frey, A. Sunyaev

Abstract—Smart contracts are a promising means of formalizing and reliably enforcing agreements between entities using distributed ledger technology (DLT). Research has revealed that a significant number of smart contracts are subject to programming flaws, making them vulnerable to attacks and leading to detrimental effects, such as asset loss. Researchers and developers call for a thorough analysis of challenges to identify their causes and propose solutions. To respond to these calls, we conducted two literature reviews and diverse expert interviews and synthesized scattered knowledge on challenges and solutions. We identified 29 challenges (e.g., code visibility, code updateability, and encapsulation) and 60 solutions (e.g., gas limit specification, off-ledger computations, and shadowing). Moreover, we developed 20 software design patterns (SDPs) in collaboration with smart contract developers. The SDPs help developers adjust their programming habits and thus support them in their daily development practices. Our results provide actionable knowledge for smart contract developers to overcome the identified challenges and offer support for comparing smart contract integration concepts across three fundamentally different DLT protocols (i.e., Ethereum, EOSIO, and Hyperledger Fabric). Moreover, we support developers in becoming aware of peculiarities in smart contract development and the resulting benefits and drawbacks.

Index Terms—Blockchain, Distributed Ledger Technology, Decentralized Applications (DApps), Patterns, Smart Contracts, Software Development

1 Introduction

SMART contracts are software programs that express logic formalized in code for the reliable enforcement of business agreements between defined entities (e.g., individuals, organizations, or machines) [1]. An early form of smart contracts is enabled by the primitive Script available for the Bitcoin blockchain to define conditional asset transfers [2]–[4]. In 2015, the Ethereum foundation went beyond Bitcoin Script's primitive capabilities by introducing the Ethereum Virtual Machine (EVM), which enables the execution of Turing complete smart contracts[1] in high-level programming languages, such as Obsidian, Solidity, or Vyper. Following the success of Ethereum, various DLT protocols (e.g., EOSIO or Hyperledger Fabric) have focused on enabling smart contracts. Through Turing completeness, smart contracts have become more expressive and better usable for manifold decentralized applications. However, the gain in expressiveness of smart contract code comes with its downsides because it can increase the complexity of smart contract code and favors the occurrence of programming flaws. Moreover, developers must anticipate the special characteristics of smart contracts, such as the public visibility of smart contract code [6], [7], the tamper resistance of deployed smart contracts [8], [9], and access management for the execution of smart contract functions [9]–[11].

Existing research has revealed that a significant number of smart contracts deployed on the Ethereum blockchain are subject to programming flaws [12]–[14] that make smart contracts vulnerable to attacks. The criticality of flaws became apparent in various incidents, such as the *DAO* hack and the *Parity Wallet* hack. Each incident led to a loss of USD $150 MM [15], [16]. Beyond Ethereum, it became clear that smart contract development is also challenging for other DLT protocols, including EOSIO (e.g., USD $58,000 was stolen using faked EOS tokens [17]) and Hyperledger Fabric (e.g., dealing with phantom reads [18]). Given the frequency and severity of flaws in smart contract code, researchers and developers call for a thorough analysis of the challenges that lead to flaws in identifying their causes and proposing corresponding solutions, ultimately improving development practices.

To reduce the challenges of smart contract development and improve the quality of smart contract code, prior research has identified several challenges (e.g., [10], [19]–[21]) and proposed appropriate solutions (e.g., [8], [10], [14]). These solutions can be largely distinguished into *automated verification* and *coding support*. For automated verification, existing research presents software tools (e.g., MadMax [14] or ReGuard [22]) for automatically identifying flaws in smart contract code (e.g., using static analysis [23], dynamic analysis [24], or machine learning [25]) and increasing code quality. Nevertheless, the applicability of automated verification to smart contract code is limited in terms of comprehensiveness because most formal verification tools apply static patterns

- *N. Kannengießer, S. Lins, and A. Sunyaev are with the Institute of Applied Informatics and Formal Description Methods, Karlsruhe Institute of Technology, Karlsruhe, Germany. E-Mail: {niclas.kannengiesser; lins; sunyaev}@kit.edu*
- *N. Kannengießer and A. Sunyaev are with the KASTEL Security Research Labs, Karlsruhe, Germany*
- *C. Sander, K. Winter, and H. Frey are with the EnBW Energie Baden-Württemberg AG, Karlsruhe, Germany. E-Mail: {c.sander; k.winter; h.frey}@enbw.com*

[1] We are aware of the discussion on the potential Turing completeness of Bitcoin's Script [5]. Since this discussion has not been finished, we align with the Bitcoin documentation [3] and find Bitcoin's smart contract capabilities not Turing-complete.

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

to identify code flaws, and mostly applies to a single type of DLT protocol, such as those using the EVM. Because of the undecidability of several computational problems (e.g., the halting problem), flaws such as infinite loops can often not be proven beyond technical boundaries (e.g., limited memory allocations).

While automated verification only applies to existing code, coding support aims to sensitize developers to smart contract challenges and respective. To this end, prior research has started tackling said challenges by developing software design patterns (SDPs; e.g., [8], [9], [26]). However, existing SDPs for smart contract development focus on only a few DLT protocols (foremost Ethereum) and are scattered across various sources (e.g., scientific papers [8], [9], [26], blogs [27], [28], and DLT-related documentation [29]), obfuscating the actual causes for existing challenges. Details of the proposed SDP and related solutions, such as the problem context, are often missing, which hinders their practical applicability for developers in day-to-day operations. It remains unclear which features of DLT protocols cause what challenges for smart contract development and how developers should effectively address these challenges.

To sensitize developers to the peculiarities of and resulting challenges in smart contract development for different DLT protocols and to help improve smart contract code quality, we ask the following research questions (RQ):

RQ1: What are the key challenges in smart contract development?

RQ2: How can developers tackle the identified challenges?

To answer these RQs, we applied a two-step research method. First, we conducted two complementary literature reviews [29], [30] and diverse expert interviews to synthesize scattered knowledge on challenges and corresponding solutions concerning smart contract development. In total, we identified 29 challenges, including code visibility, concurrency, and data type complexity, and 60 corresponding solutions, including off-ledger computations, synchronization, and array replacement. We further grouped these into three principal origins—*platform, programming language and execution environment,* and *coding practice*—according to the individual challenge's causes. Second, we iteratively derived and evaluated SDPs for smart contract development based on a selected set of identified challenges and solutions because the details of proposed solutions in general and SDPs in particular are often missing in extant research. We particularly applied a thorough pattern generation approach and a strict canonical structure for SDPs (e.g., [31]–[33]) to ease the understanding and usage of patterns for smart contract developers and overcome the limitations of prior research regarding pattern applicability.

This work contributes to practice, as we present challenges developers frequently face when developing smart contracts for Ethereum, EOSIO, and Hyperledger Fabric and corresponding solutions. Moreover, we derived 20 SDPs in collaboration with smart contract developers from solutions that became best practices. These help developers avoid frequent challenges in smart contract development and avoid common flaws in smart contract code. By developing a three-layered hierarchy of challenges that starts with the three principal origins for challenges, we support developers in separately assessing the possible drawbacks of DLT protocols and offered programming languages and execution environments. We thereby help developers select and configure a DLT protocol under consideration of particular use-case requirements and their personal preferences. For example, developers can better assess which DLT protocol to combine with which virtual machine as offered by upcoming middleware, such as Neutron [34] for the Ethereum-based Qtum framework.

We contribute to the research by synthesizing scattered knowledge on smart contract development challenges and solutions across three major DLT protocols—Ethereum, EOSIO, and Hyperledger Fabric. We highlight the implications of different design decisions for DLT protocols for smart contract development (e.g., regarding the characteristics of an execution environment and corresponding programming languages). Thereby, we support the understanding of the interplay between DLT protocols and their smart contract execution environments. By applying the canonical pattern structure proposed in prior research [31]–[33], our SDPs contain detailed descriptions of each challenge and its solution and a discussion on benefits and boundary conditions, thereby extending prior research that briefly outlined potential solutions to overcome challenges.

This work is structured as follows. First, we introduce the fundamentals of DLT, smart contract development, and SDPs. Second, we briefly explain the applied method for identifying smart contract development challenges and corresponding solutions and how we derived the 20 SDPs. Third, we present the derived challenges for smart contract development and depict how the identified solutions and our SDPs can overcome these challenges. Fourth, we discuss this study and our findings in the context of related works and describe our implications for research and practice. We conclude with a summary and our principal findings and describe the limitations of this work, as well as corresponding starting points for future research. To make the developed SDPs easy to use, we made them accessible in our public git repository.[2]

2 Background

2.1 Distributed Ledger Technology

DLT enables multiple individuals or organizations to collectively operate a digital platform in a decentralized manner. This decentralized digital platform is based on a highly available and tamper-resistant distributed database (i.e., distributed ledger), where various storage and computing devices (i.e., nodes) maintain local copies of stored data [35]. Nodes add data to their local ledger version in the form of transactions. In blockchains, these

[2] https://github.com/KITcii/smart-contract-dev-support

transactions are batched together in blocks. Transactions can contain digital representations of assets (e.g., coins) or the byte code of a smart contract (e.g., in Ethereum). When a node receives a new transaction, the node first validates the transaction [36]. Validated transactions remain in the node's local storage (i.e., mempool) until the nodes verify the transaction by appending it to its local copy of the ledger. The order in which transactions are verified is decided by each individual node, leading to a kind of concurrency in transaction processing [37].

Distributed ledgers operate in untrustworthy environments characterized by the arbitrary occurrence of Byzantine faults, comprising temporarily unreachable nodes, crashed nodes, or malicious behavior of nodes (e.g., double-spending of assets) [38], [39]. Byzantine faults and network delays can cause nodes in a DLT network to store different ledger versions and thus be in different states. Such inconsistencies in a distributed ledger can cause vulnerabilities in DLT systems. For example, inconsistencies across nodes cause network partitions that can make distributed ledgers vulnerable to tampering [35], [39]. To resolve inconsistencies between the versions of the ledger stored on different nodes, consensus mechanisms are used.

Within DLT networks, nodes can have different permissions for appending new data to the ledger (i.e., write permissions) [35]. In permissioned DLT protocols, only specified nodes are permitted to participate in consensus finding and commit new data to the distributed ledger. In permissionless DLT systems, all nodes in the DLT network can participate in consensus finding.

When interacting with a distributed ledger, entities (i.e., individuals, organizations, or devices) have individual digital identities with attributes, such as a unique pseudonym as an identifier (e.g., an account address in Ethereum). The pseudonym can be used to reference an account in the distributed ledger, and entities can send and receive transactions using the pseudonym.

2.2 Smart Contracts

Smart contacts offer reliable enforcement of agreements formalized in the program code between multiple parties. Depending on the DLT protocol, different concepts have been applied for the integration of smart contracts (see Table 1). These differences often originate from design decisions, especially concerning the consensus mechanism. For example, the Hyperledger Fabric protocol does not require deterministic smart contract execution to favor consensus finding, but applies Raft as a centralized consensus mechanism with only crash-fault tolerance and no Byzantine fault tolerance. In contrast, the Ethereum protocol requires determinism in smart contract execution to make the consensus mechanism more secure, which can be decentralized and Byzantine fault-tolerant. In the following sections, we explain the concepts of how smart contracts are integrated into three major open-source DLT protocols supporting smart contracts: Ethereum, EOSIO, and Hyperledger Fabric.

Ethereum. The Ethereum protocol natively uses a proof-of-work-based consensus mechanism and applies the concept of a Greedy Heaviest Observed Sub-Tree (GHOST) to resolve inconsistencies between nodes. Smart contracts are independently executed by all nodes in Ethereum-based DLT networks. Still, these nodes must eventually agree on a consistent state requiring the deterministic execution of smart contracts. To prevent nondeterministic smart contract execution, the EVM encapsulates smart contracts, hindering interaction with external information systems (i.e., oracles) and the use of real randomness (e.g., for random number generation).

Ethereum allows for the development of smart contract code in several high-level programming languages, including Solidity, Obsidian, or Vyper. After development, the smart contract code must be compiled to bytecode (e.g., using *solc* compiler for Solidity or *vyper* compiler for Vyper) and produce a corresponding application binary interface (ABI) file to specify application programming interface (API) for interactions with the smart contract. For deployment, the bytecode is included in the payload of a transaction issued to the Ethereum network. After the bytecode has been deployed, the smart contract is included in the blockchain and stored in a tamper-resistant manner.

Each Ethereum smart contract has an individual account with a unique address, similar to the externally owned

TABLE 1
COMPARISON BETWEEN SMART CONTRACT INTEGRATION CONCEPTS

	DLT Protocol		
Characteristic	**EOSIO**	**Ethereum**	**Hyperledger Fabric**
Distributed Computing	All nodes execute all smart contracts upon invocation	All nodes execute all smart contracts upon invocation	Only defined nodes execute smart contracts upon invocation
Deterministic Execution	Required for consensus finding	Required for consensus finding	Not required for consensus finding but potentially important to fulfilling the endorsement policy
Execution Regulation Mechanism	Execution bounded by time or by a maximum number of instructions	Execution bounded by gas consumption	Execution timeout
Execution Environment	EOS Virtual Machine (EOSVM)	Ethereum Virtual Machine (EVM)	Docker container
Programming Languages	C++	Solidity, Vyper	Go, Java, Node.js
Deployment Process	Bytecode included in the transaction	Bytecode included in the transaction	Manual installation by node controllers
Bytecode Storage	RAM and blockchain on disk	Blockchain on disk	Docker container on disk
Execution Process	Order-execute	Order-execute	Execute-order-validate

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

accounts used by entities external to the distributed ledger. Smart contracts can receive, store, and transfer assets and interact with other accounts. The execution of a particular smart contract function is triggered by a transaction sent to the smart contract address with the signature of the target function in the data field.

In Ethereum, each node maintains its own state *s*. Successfully processed transactions cause a state transition from s_t to s_{t+1} [37]. Although these nodes independently execute smart contracts, all nodes in the DLT network must eventually agree on a common state requiring the deterministic execution of smart contracts following a replicated state machine model. To prevent nondeterministic smart contract execution, the EVM largely encapsulates smart contracts, hindering interaction with external information systems (i.e., oracles) and the use of real randomness (e.g., for random number generation).

To counter denial of service in smart contract execution (e.g., through infinite loops) and reward nodes for the provision of computational resources for the execution of smart contract code, the Ethereum protocol applies a pricing scheme. The pricing scheme uses the unit *gas* to measure computational resource consumption associated with transaction processing. With each transaction, entities pass a maximum amount of gas they are willing to spend (i.e., gas limit) for the transaction processing (e.g., to execute a function or deploy a smart contract) and the corresponding amount of Ether to pay per consumed unit of gas. If the execution exceeds the gas limit, the execution is aborted and rolled back.

For function calls from one smart contract (A) to another (B), Solidity offers three ways to invoke functions [40], [41]: `call`, `delegatecall`, and `staticcall`. When using `call` (e.g., in direct calls like `ContractB.functionName(…)`), the target function provided by B is executed in a separate context from caller contract A and can access its own smart variable values. When using `delegatecall`, the target function is executed in the context of the caller contract and can also change variable values of the caller contract. `staticcall` can be used to call a smart contract but disallows any state changes during its execution. Attempts to make state modifications result in an exception, and no modifications are made.

EOSIO. Blockchains that build on the EOSIO protocol (e.g., the EOS blockchain) natively use a consensus mechanism consisting of two components: Delegated Proof-of-Stake (DPoS) to elect block producers and asynchronous Byzantine fault tolerance (aBFT) to finalize blocks [42]. The EOSIO consensus mechanism requires all nodes to agree on the same state of the main chain. Thus, EOSIO imposes a deterministic execution of actions to favor consensus finding [43].

EOSIO smart contracts are programmed in C++ and are compiled into WebAssembly (WASM) formatted bytecode using the *eosio-cpp* compiler. The compilation process also produces an ABI file to derive the smart contract API for interactions. For deployment, a smart contract bytecode is put into a transaction that is sent to call the *eosio.system contract*. Executable bytecodes of current EOSIO smart contracts are hosted in the random-access memory (RAM) of nodes. The blockchain records all transactions and events on the disks of nodes in the DLT network [44], [45].

In EOSIO, each account is identified by a unique name with a length of one to twelve characters [46]. Each smart contract has a unique account and exhibits actions invoked by accounts. Actions are invoked through action instances, defining the target account, the name of the action to be executed, a list of authorizations to prove permissions for action execution, and action data (e.g., function arguments). Action instances are included in transactions and are executed by validating nodes in sequential order.

Upon transaction receival, nodes check whether the authorizations included in action instances fulfill the permissions defined for the smart contract actions to be called. Permissions are linked to an authority table where the individual permissions for the execution of actions are defined by the respective smart contract owners [46]. If the permission check fails for at least one account in an action instance, the entire transaction processing is aborted and no smart contract action is executed. Otherwise, the node invokes the actions defined in the action instances [47]. Before executing the called actions, the *nodeos* daemon running on each node makes a local snapshot of the state history and loads the WASM bytecode of the smart contract into the EOS Virtual Machine (EOSVM) for execution [43]. During the execution of actions, EOSIO smart contracts can invoke other contract actions by using inline actions [48], [49]. Inline actions synchronously execute an action in the context of the original transaction. If an action execution is aborted, all changes made in the transaction context are rolled back using the snapshot.

Analogous to Ethereum, EOSIO applies a mechanism to prevent infinite loops. Developers can define a maximum number of instructions or use a watchdog timer with a maximum runtime for the sequential execution of actions in a transaction [43]. If one of the defined thresholds is exceeded, the execution of actions in the transaction is aborted, and all changes are rolled back.

Hyperledger Fabric. Hyperledger Fabric is used to set up permissioned blockchains, with Raft as the recommended consensus mechanism [50]. Unlike Ethereum and EOSIO, Hyperledger Fabric does not use a native cryptocurrency and was designed for business use cases where known individuals and organizations form a consortium that operates and uses the blockchain.

To keep data confidential within consortia, Hyperledger Fabric allows for setting up channels on top of the consortium's infrastructure. Channels are private subnetworks between specific consortium members that use granular access control based on their identities [51]. Consortium members that are part of a channel (i.e., channel members) operate a blockchain and a world-state database on their peer nodes isolated from other channels. The blockchain records all transactions and determines the world state, storing current values related to defined business objects. Within channels, peer nodes enforce endorsement policies for transactions, and ordering nodes execute the consensus mechanism and

commit transactions to the blockchain. Data stored on the blockchain are only visible to members of the corresponding channel [51].

In Hyperledger Fabric, there are chain codes that can include multiple smart contracts [52]. To make smart contracts available to applications [53], developers manually install chain codes on peer nodes in the channel that are specified in the policy to endorse transactions. Smart contracts contain the actual transaction logic, which can be expressed in Go, Java, or Node.js [52].

To call a smart contract function, an entity sends a transaction proposal via its software client to peer nodes hosting the chaincode. The transaction proposal includes the chaincode identifier and input parameters for the function call. Before function execution, the peer nodes check whether the transaction proposal matches the required format and whether the issuer is authorized to call the smart contract according to the chaincode endorsement policy [54]. If the transaction proposal succeeds in these checks, the corresponding peer node executes the smart contract function. Otherwise, the transaction is marked invalid. Valid and invalid transactions are stored on the blockchain, but only valid transactions can update the world state database. After the smart contract function is executed, the smart contract produces a new transaction (i.e., transaction response) that includes updates on the world state. The peer nodes send their transaction responses to the client. The client compares the transaction responses to check whether the smart contract execution fulfills the endorsement policy defined for the chaincode. For example, if the endorsement policy requires two of three peer nodes to have an equal outcome upon transaction execution, the client compares the transaction responses from the three peer nodes for consistency. If at least two peer nodes calculate an equal output, the client creates a new transaction, including the transaction proposal, the transaction response, and the digital signatures of the peers. The client sends the new transaction to the channel's ordering nodes to be committed to the blockchain [53].

Unlike Ethereum and EOSIO, Hyperledger Fabric does not strictly bind smart contract execution to resources (e.g., by gas in Ethereum or time in EOSIO). Still, developers can define a maximum execution time per chaincode (i.e., `ExecuteTimeout`).

Blockchains based on Hyperledger Fabric do not strictly require determinism in the execution of smart contracts. Inconsistent results from smart contract execution are filtered out to avoid contradictions in consensus findings after endorsing peer nodes have executed the smart contract. Following the order-execute-validate approach in Hyperledger Fabric instead of the execute-order approach applied in Ethereum and EOSIO, only consistent results will be forwarded to ordering nodes to be committed to the blockchain after consensus finding [55].

2.3 Software Design Patterns

A pattern is an abstraction from a concrete design that keeps recurring in specific nonarbitrary contexts [31], [56], [57]. Patterns usually refer to the architecture or structure of several parts in a superordinate system. They comprise a general description of a recurring problem and an associated solution with defined objectives and constraints. SDPs form a special class of patterns that describe objects and classes and their communication and customization to solve a general software design problem in a particular context [56]. SDPs can be further distinguished into three abstraction levels [31]: *architectural patterns*, *design patterns,* and *idioms*. *Architectural patterns* describe "[…] a fundamental structural organization or scheme for software systems and provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them" [31, p. 12]. *Design patterns* provide "[…] a scheme for refining the subsystems or components of a software system, or the relationships between them" [31, p. 13] to solve a general design problem within a certain context. *Idioms* are patterns on the lowest level of abstraction and "describe how to implement particular aspects of components or the relationships between them using the features of the given language" [31, p. 14].

3 Methods

To answer our research questions, we applied a two-step research approach. First, we conducted extensive literature reviews and expert interviews following established methodological guidelines [30], [58], [59] to identify challenges and solutions in smart contract development. Second, we iteratively derived and evaluated SDPs for smart contract development based on identified challenges and solutions, while considering extant research, gray literature, and practitioners' knowledge.

3.1 Identifying and Synthesizing Smart Contract Challenges and Solutions

We applied a mixed-method approach to identify and synthesize preliminary challenges and corresponding solutions in smart contract development, comprising different types of descriptive literature reviews [60] augmented by expert interviews. We performed four iterations of data gathering accompanied by iterative data analyses to achieve theoretical saturation. Table 2 summarizes the objectives, applied methods, and outcomes of each iteration. In the following sections, we briefly summarize each iteration. Appendix A provides detailed information on each iteration to enhance method transparency and reproducibility.

TABLE 2
SUMMARY OF ITERATIONS TO IDENTIFY AND SYNTHESIZE SMART CONTRACT CHALLENGES AND SOLUTIONS

	Iteration 1	Iteration 2	Iteration 3	Iteration 4
Objectives	• Understand the problem domain • Identify an initial set of smart contract challenges and solutions	• Verify and extend preliminary themes • Incorporate practice knowledge	• Extend themes • Strive for theoretical saturation	• Verify themes • Discuss the relevancy and applicability of themes for different ledgers
Data Source	Snowballing literature review [58]	Focus group interview	Database literature review [29]	Semi-structured expert interviews
Data	21 documents on smart contract challenges and solutions (incl. grey literature)	Interview transcript on a six-hour focus group with five DLT experts	86 research documents on smart contract challenges and solutions	Nine interviews with experts on EOSIO, Ethereum, and Hyperledger
Data Analyses	Thematic analysis [30]	Selective, open, and axial coding of interview findings	Thematic analysis [30] and synthesizing prior iteration findings	Joint extension of themes and field notes analyses
Outcomes	146 text segments assigned and aggregated into seven preliminary themes of challenges and solutions	One additional challenge theme, refined and enriched themes of challenges and solutions	13 themes of smart contract challenges and respective solutions, relating to 1018 text segments; three principal origins	29 challenges and 60 corresponding solutions relating to three ledgers

3.1.1 First Iteration: Snowballing Literature Review

To generate an initial set of smart contract challenges and solutions as a foundation for our study, we conducted a literature search utilizing the snowballing approach proposed by Wohlin [58]. Snowballing starts with a core set of relevant documents and then uses these to identify further relevant documents in a specific domain through multiple iterations of forward- and backward-snowballing.

We first applied the search string *"Smart Contract*" AND ("Challeng*" OR "Vulnerabilit*")* to Google Scholar, as suggested by Wohlin [58], yielding 879 documents as of July 4, 2019. We analyzed documents' meta-information (i.e., title, abstract, etc.), read potentially relevant documents, and applied inclusion and exclusion criteria (see Table 3) to identify the documents relevant to answer our research question. This relevancy check led to a set of ten starting documents on which we then conducted three rounds of backward and forward searching [58], resulting in 21 documents, including grey literature.

For the literature analysis, we applied thematic analysis [30] to identify themes for challenges and related solutions apparent in smart contract development. During this thematic analysis, we performed multiple rounds of data-driven coding. Afterward, we compared our codes and the respective text segments to form overarching themes [30]. We were able to identify 18 candidate themes, including *exception handling* and *event order*. We revised these themes by applying Patton's [61] dual criteria of internal homogeneity (i.e., data within themes should cohere together meaningfully) and external heterogeneity (i.e., there should be clear and identifiable distinctions between themes), leading to seven themes as a result.

3.1.2 Second Iteration: Focus Group Interview

We decided to conduct a second iteration for two reasons. First, we wanted to validate our literature findings because qualitative coding techniques bear the risk of interpretation and other biases. Second, we strove to incorporate knowledge from experts in the field to extend and enrich our themes. We, therefore, conducted a focus group interview [59] in July 2019 using a convenience sample of five DLT experts (see Table 4). Three researchers participated in and moderated the workshop.

We conducted the focus group interview based on an interview guide [62] comprising a brainstorming phase about potential challenges in smart contract development, a discussion about potential solutions, and specific questions to validate and gather additional data surrounding the seven themes identified in the first iteration. The interview lasted six hours and was recorded and then transcribed. We applied scientific coding techniques to analyze the interview data, especially selective (i.e., assigning prior themes to interview data), open (i.e., labeling new challenges and solutions discussed in the interviews), and axial coding (i.e., identifying the causes and consequences of each challenge) [63]. We identified one new challenge theme—code discoverability—and a corresponding solution, refined existing themes by enriching their cause and consequence descriptions, identified minor inconsistencies, and resolved these accordingly (e.g., we unified the levels of abstraction of the solutions).

TABLE 3
INCLUSION AND EXCLUSION CRITERIA FOR LITERATURE SEARCHES IN ITERATION 1 AND ITERATION 3

Type	Name	Description
Inclusion	Challenge	Naming, proposing, discussing, or revealing smart contract challenges, vulnerabilities, or related issues
	Solution	Naming, proposing, discussing, or revealing solutions to tackle smart contract challenges, vulnerabilities, or related issues
	Concrete SDP	Naming, proposing, discussing, or revealing SDPs relating to smart contracts
	Transferrable SDP	Naming, proposing, discussing, or revealing SDPs that are applicable or transferable to smart contracts
Exclusion	Off-topic	Not dealing with DLT and smart contracts
	Books	Books on smart contracts
	Not in English	Publications in a non-English language
	Duplicates	Multiple identical occurrences of a document

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

TABLE 4
OVERVIEW OF INTERVIEWEES' DEMOGRAPHICS

	Iteration 2	Iteration 4
Number of Experts	5	9
DLT Expertise	Ethereum	Ethereum, EOSIO, Hyperledger Fabric
Average Software Engineering Experience, in years	3.9	9.1
Average Smart Contract Development Experience, in years	2.9	3.2
Industry	Energy, IT	Automotive, Energy, IT

3.1.3 Third Iteration: Database Literature Review

We decided to return to scientific literature in the next iteration for two reasons. First, new documents may have been published since our first literature search. Second, we might have overlooked relevant documents given our snowballing research approach. We, therefore, performed a descriptive literature review [60] in scientific databases, trying to achieve theoretical saturation.

To cover a broad set of documents, we applied the search string *("smart contract*" OR "chaincode*") AND ("challenge*" OR "pattern*" OR "issue*" OR "develop*" OR "programming*")* in the title, abstract, or keywords (TAK) of prominent databases, including ACM Digital Library, AISel, EBSCOHost, IEEEXplore, Proquest, ScienceDirect, and Web of Science. Our search yielded a total of 1,774 potentially relevant documents as of November 6, 2020. We once again performed a comprehensive relevancy check by applying the same inclusion and exclusion criteria from our first iteration (see Table 3), resulting in 86 novel relevant documents.

We again applied thematic analysis [30] to refine our existing themes and identify novel ones for challenges and solutions, comprising two rounds of data-driven coding, constant comparison of coded challenges and solutions to identify subthemes, and frequent theme refinement. This analysis process resulted in 13 themes, including *code efficiency, confidentiality*, and *determinism*. To further group these themes, we compared their different origins and strived to identify a core set of common ones. Our comparative analysis revealed three principal challenge origins that can make smart contracts prone to programming flaws (see **Error! Reference source not found.**): *platform*, *programming language and execution environment*, and *coding practice*.

Compared to our first snowballing literature research, we were not only able to identify six novel high-level themes, but also to refine and enrich existing themes, and create a hierarchy of themes raging from text segments and initial labels for challenges and solutions up to aggregated challenge themes that were assigned to principal challenge origins.

TABLE 5
PRINCIPAL ORIGINS THAT CAN CAUSE CHALLENGES IN SMART CONTRACT DEVELOPMENT

Component	Description
Platform	The protocol put in place to manage the interactions between nodes and define the procedures for the issuance, verification, and storage of transactions
Programming Language & Execution Environment	The capabilities offered to develop smart contracts and execute them via the distributed ledger
Coding Practice	The development activities to achieve a specified outcome in the form of a smart contract

3.1.4 Fourth Iteration: Expert Interviews

While we already gathered rich information on various challenges and were able to identify solutions for many of them, we decided to conduct a fourth iteration due to two reasons. First, we strove to validate our literature review findings with further knowledge from DLT experts. Second, most of the literature focuses on Ethereum, and the EVM and Solidity, respectively. Hence, we were eager to reflect the applicability of our findings to other DLT protocols. We particularly focused on EOSIO and Hyperledger Fabric in addition to Ethereum because these DLT protocols are frequently used in organizational contexts and allow for insights into three distinct smart contract integration concepts.

In the fourth iteration, we conducted nine semi-structured interviews [59] with DLT experts (see Table 4 for an overview). We again prepared a guide [62] to structure the interviews. In particular, we asked whether the identified challenges and solutions are relevant and applicable to other ledgers. All interviews were performed via video conference tools between December 2020 and February 2021 and were recorded with the permission of participants. The average interview time was 63 minutes. The researcher who carried out the interview took notes on each challenge and solution discussed with the interview partner during the interview.

After each interview, we analyzed the field notes and compared them to our intermediate findings from the third iteration. We enriched our theme descriptions, renamed themes and challenges suggested by the interviewees, added novel challenges and solutions, and added ledger-specifics to each challenge and solution description. The analyses results confirmed the three principal challenge origins and led to refinements of the associated sub-themes. More importantly, we were able to identify novel challenges and solutions by comparing challenges and solutions across ledgers and by considering ledger specifics. For example, we identified the challenge *of nondeterministic behavior* that only applies to smart contracts developed in Go. By finishing the fourth iteration, we identified 29 challenges and 60 solutions.

To ensure that we identified a reliable set of challenges and solutions, we followed researchers stressing that an important goal is to reach theoretical saturation in qualitative research [64]–[66]. Theoretical saturation is often taken to indicate that further data collection or analysis is unnecessary based on the data analyzed hitherto because it is unlikely that further data collection will generate new findings [67], [68].

We first looked at our literature review protocols, revealing that our literature analysis did not reveal new challenges or solutions since the last twelve analyzed documents. Similarly, we asked our interviewees during our fourth iteration if they knew of any further challenges or

solutions that had not been discussed so far during the interviews. We sent our manuscript to the interviewees after finishing the writing to ask them once more if they had any challenge or solution to add. In both cases, the interviewees agreed that they were not aware of further challenges and solutions to the best of their knowledge. Consequently, we are confident that we have reached at least a sufficient degree of theoretical saturation after completing our fourth iteration.

3.2 Generating and Evaluating Software Design Patterns

3.2.1 Generation of SDP

By performing multiple iterations of data gathering and analysis, we were able to identify 29 challenges and 60 solutions. However, details of proposed solutions in general and SDPs in particular are often missing, such as the problem context and resulting context, which have been requested by existing research (e.g., [31]–[33]). This lack of information not only hinders practical applicability by developers in day-to-day operations, but also hampers the adjustment of their programming habits. To counteract these issues, we next transferred the identified solutions into SDP as actionable means.

Selection of Solutions as a Base for SDPs. Given the high quantity and diversity of solutions, we first selected a set of solutions as a base for developing SDPs. On the one hand, we derived an SDP for a solution if (1) an SDP had been proposed by prior research; (2) a problem kept recurring in specific nonarbitrary contexts; (3) sufficient information was available to describe the SDP in detail (e.g., information gained from interviewees or studies); or (4) interviewees called for the development of SDPs and stressed their relevance and potential contribution. On the other hand, we particularly refrained from developing SDPs for a solution if they were (1) trivial (e.g., solution *S.10.1 Read the Documentations*); (2) only applicable on a limited scale (e.g., challenge *C.13 Non-deterministic Behavior* applies to smart contracts developed in Go only); or (3) on a very low abstraction degree that prevents generalization (e.g., solution *S.21.1* recommending the usage of data type `bytes` over `byte[]`). After applying these selection criteria, we decided to develop 20 SDPs related to various challenges and solutions (see Table 6).

SDP Generation. To generate SDPs, we followed existing research providing common pattern structures [31]–[33], comprising a name, context, problem, forces, solution, examples, resulting context, rationale, related patterns, and known uses of a pattern. For each pattern, we carefully specified each structural dimension based on extant research, interview findings, our own experiences, and prototypical instantiations, as outlined below.

First, for each SDP we defined a meaningful name [33]. If it was suitable, we aligned the naming of the derived smart contract SDP with the naming of common SDPs in traditional software engineering (e.g., *Façade Pattern* or *Proxy Pattern*). In the results section, we refer to these names but cite the original documents in which we found a similar solution. We also adapted our SDP names based on the feedback gained throughout the interviews to increase their comprehensibility and align our wording with the terms used in the software engineering community. For example, we renamed the *Register Contract Pattern* [9] the *Observer Pattern*.

We next elaborated on the context that suggests SDPs' applicability and in which a problem and its solution recur [33]. For each SDP, we discussed to which challenges and solutions the pattern relates. For example, we mapped the *Checks-Effects-Interactions Pattern* with challenge *C.15 Cross-Account Interactions* and solution *S.15.4 Instruction Order* because the pattern can prevent reentrancy attacks. This mapping helped us to ground our SDP descriptions on extant research and interviewees' opinions. In addition, we discussed the applicability of the pattern to Ethereum, EOSIO, and Hyperledger Fabric.

We then defined a problem that described the objectives to be achieved within the contexts by applying the SDP [33]. While typical pattern objectives relate to the mitigation of risks, such as those associated with the removal or deactivation of smart contracts with the *Deactivation Pattern*, we particularly considered problem specifics, such as pre-conditions and boundary conditions. We also reflected on whether the problem may appear in each DLT protocol because they differ in their smart contract integration concepts (see Table 1). For example, problems leading to challenge *C.21 Data Type Complexity* only relate to smart contracts based on Ethereum and EOSIO because they bind smart contract execution to resources (e.g., gas in Ethereum or time in EOSIO).

Afterward, we specified forces that reveal the details of a problem and define the kinds of trade-offs that must be considered in the presence of the tension or dissonance they create [33], [69]. Forces commonly relate to the characteristics of an application, such as maintainability or response time. For example, the *Token Pattern* improves maintainability but comes at the cost of code efficiency as the number of required interactions increases. When describing the forces and constraints and how they interact, we considered the objectives to be achieved when using the SDP. Analyzing potential forces also supported us in comparing different solutions and their appropriateness for use in the pattern.

Next, we focused on describing a solution that includes static relationships and dynamic rules to realize the desired outcome [33]. To define an appropriate solution (i.e., fulfilling the SDPs' objectives while considering the forces), we went back to the data gathered through identifying and synthesizing challenges and solutions and compared proposed solutions for a given challenge. For example, prior research proposes using the *Oracle Pattern* whenever external data or real-world data is required by a smart contract [70], [71]. We synthesized information about oracles to come up with a solution for our SDP. We also coped with opposing views and research findings. For example, prior research provides different means to tackle the challenge *C.4 Randomness*, whereas some of these means have been later proven to expose flaws (e.g., dependence on blockchain properties to generate

random numbers, such as block hash values). If needed, we implemented solutions and tested them to ensure their correctness and to better describe their inner functioning.

To increase the understandability and applicability, we added an *example* to each SDP [33]. The example shows a possible implementation of the solution. We took examples from extant research, GitHub and related smart contract repositories, interviewees' suggestions, and developed and tested our own examples. To further support developers, we also provided antipattern examples to emphasize what typically goes wrong in smart contract development.

As the next step, we defined the resulting context that describes the system state after applying a pattern [33]. For explaining the resulting context, we reflected on benefits (e.g., problems solved) and drawbacks (e.g., further challenges caused by the pattern utilization).

To substantiate the solution, we provided a rationale concerned with justifying how and why the solution resolves its forces to align with the desired objectives and why it is suitable [33]. Since we built our SDPs on justificatory knowledge from extant research and practitioners' expertise, we aimed to summarize the assumptions of why the SDP works as a solution.

Since patterns often share common forces and a compatible initial or resulting context, we defined related patterns [33]. Related "patterns might be predecessor patterns whose application leads to this pattern; successor patterns whose application follows from this pattern; alternative patterns that describe a different solution to the same problem but under different forces and constraints; and codependent SDPs that may (or must) be applied simultaneously with this pattern" [33, p. 6]. Highlighting relations between patterns supports developers in selecting alternative solutions to a problem. For example, the *Mutex Pattern* can be used as an alternative to the *Checks-Effects-Interactions Pattern* to protect smart contracts from reentrancy attacks.

To show that the SDP is an approved solution for a problem, we finally listed the known uses of an SDP in existing systems [33]. We searched smart contract databases (e.g., etherscan.io), developer repositories (e.g., GitHub), websites (e.g., Ethereum Name Service), and whitepapers and foundation blogs (e.g., Ethereum foundation blog) to identify known uses of SDPs.

By applying this canonical structure to SDPs, we want to ease the understanding and usage of patterns for smart contract developers and overcome issues regarding pattern applicability in prior research.

3.2.2 Derived Software Design Pattern Evaluation

Whereas we built the SDPs on the data and findings from identifying and synthesizing challenges and solutions, we aimed to evaluate them to ensure correctness, comprehensibility, and practical applicability.

SDP Evaluation Criteria Derivation. To evaluate our SDPs, we first defined a set of evaluation criteria. Therefore, we conducted a scoping literature review to identify quality criteria for SDPs to consider in the evaluation (see Appendix B). We particularly focused on the most cited scientific works on software design patterns in the English language. Eventually, we identified 12 particularly relevant documents on quality criteria for software design patterns (e.g., [72]–[75]). To synthesize quality criteria across these documents, two researchers read the relevant documents and independently analyzed their content following the open coding approach [63]. In total, the analysis revealed 23 criteria for the evaluation of the SDPs. Based on the identified evaluation criteria, we created five groups to which we assigned the evaluation criteria: *flexibility*, *outcome*, *pattern design*, *perception*, and *utilization*. Flexibility refers to the range of applicability of an SDP. Outcome is about the results when a software design is applied. The structure of an SDP is discussed in the group pattern design. Perception considers the characteristics of users' perceptions of an SDP. Utilization refers to the applicability of an SDP. Among the identified evaluation criteria, we found a subset of 12 evaluation criteria suitable for our evaluation.

Evaluation Interviews. To evaluate the SDPs, we conducted a focus group workshop with four smart contract developers. First, we wrote a handout that included the 12 suitable evaluation criteria and the 20 SDPs, which should be discussed considering these twelve evaluation criteria. For the evaluation, we organized two events: an introductory event and a focus group interview. In the introductory event, we discussed the handout with the four smart contract developers to familiarize them with the SDPs and let them share their initial thoughts. The developers had an average experience in developing smart contracts of 4.5 years. During the following week, the four smart contract developers individually familiarized themselves with the SDPs and took notes in their handouts. The participants sent us their notes on the SDPs before the second event to consolidate their feedback. Based on the participants' feedback, we developed an interview guide for the semi-structured focus group workshop, as recommended in existing works [76], [77]. Next, we carried out a focus group workshop with three of the four smart contract developers. We discussed each SDP in detail during the workshop, elaborated on the feedback, and jointly improved the SDPs. We recorded the focus group workshop, subsequently transcribed the recordings, and analyzed the transcriptions by extracting improvements for the SDPs. We refined the SDPs accordingly and sent the revised SDPs to the fourth participant of the introductory event for an additional interview. We revised the SDPs again and sent the revised version to all four participants to gather additional comments. This final round of feedback comprised only minor issues, such as wording and description improvements.

4 Smart Contract Development

We identified three principal challenge origins in smart contract development that can make smart contracts prone to programming flaws (see **Error! Reference source not found.**): *platform*, *programming language and execution*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

environment, and *coding practice.* Associated with the principal challenge origins, we revealed 11 sub-themes (e.g., confidentiality challenges and interoperability challenges), including 29 specific challenges and 60 corresponding solutions (see Table 6), including 20 SDPs. In the following, we describe the three principal challenge origins and their subordinate 11 challenge themes. For each challenge theme, we describe the identified challenges (*C*) and discuss the corresponding solutions (*S*). If not mentioned, the identified solutions apply to blockchains based on Ethereum, EOSIO, and Hyperledger Fabric. If an SDP relates to a solution, we also briefly describe the SDP. For the complete description of the SDPs, please refer to Appendix C, D, and E or our GitHub repository.[2]

4.1 Challenges Caused by the Platform

The principal challenge of the origin platform refers to the protocol put in place to manage the interactions between nodes and to define the procedures for the issuance, verification, and storage of transactions.

4.1.1 Confidentiality Challenges

Challenges related to confidentiality can decrease the degree to which unauthorized access to information is prevented.

(C.1) Code Visibility: *The protection of the deployed smart contract code from being visible to entities with access to the distributed ledger.*

In DLT protocols, where multiple nodes execute smart contract code (e.g., Ethereum-based and EOSIO-based blockchains), smart contract logic is usually exposed to all entities operating these nodes. The visibility of code to these entities is particularly challenging for companies that have smart contract logic at the core of their business models and must keep this sensitive logic confidential. In addition, visibility of tamper-resistant code facilitates the identification of vulnerabilities and their exploitation. Challenges related to code confidentiality preservation apply to blockchains based on Ethereum or EOSIO. In Hyperledger Fabric, only nodes that must endorse a transaction store the respective chaincode. Still the following solutions also apply to blockchains based on Hyperledger Fabric.

(S.1.1) Off-Ledger Computations: A solution to protect smart contract logic from being visible to all entities with access to the distributed ledger represents the deployment and execution of logic external to the distributed ledger (i.e., off-ledger) using an oracle (see *Oracle Pattern*) [70], [71]. Upon the invocation of a smart contract function, the smart contract can initiate a call to a service provided by the oracle. Before the called service invokes the callback function of the smart contract, the oracle must convert its response to a compatible data type. Otherwise, it can lead to asset loss. For example, Ethereum does not support decimal data types. Thus, oracles provide integer values to the smart contract to avoid truncation errors (see C.26 *Appropriate Data Type Use*). The integration of oracles into smart contracts is especially challenging for DLT protocols that encapsulate smart contract execution, for example, Ethereum-based and EOSIO-based blockchains (see *Encapsulation* in Section 4.2.3). Using Hyperledger Fabric, oracles are accessible directly from the Docker container, where the chain code is executed.

(C.2) Data Visibility: *The protection of transaction data stored on a distributed ledger from being visible without authorization.*

In addition to smart contract bytecode, other transaction data (e.g., number of transferred assets) are commonly visible to entities that operate nodes. The broad visibility of data can violate data confidentiality. Still, visibility of data representing verifiable proofs for the happenings of events is important for the secure functioning of the DLT system. Protection of data visibility is challenging on blockchains based on Ethereum, EOSIO, and Hyperledger Fabric.

(S.2.1) Data Encryption: A solution to protecting data from unauthorized reads is to encrypt the data prior to its submission to the DLT network [78], [79]. However, not all transaction data can be encrypted in transactions but only the payload data or even only parts of the payload data—for example, if the transaction is to invoke a smart contract.

(S.2.2) Commitment Pattern: To keep data secret for a particular time while binding an entity to that data (e.g., binding an entity to their bid value in a bet), the *Commitment Pattern* can be used [9]. The *Commitment Pattern* comprises a commitment phase and a reveal phase. In the commitment phase, each entity first individually specifies data (e.g., a result in a lottery) and a random nonce, concatenates these two values, and sends the hash values of the concatenated value and the nonce to a smart contract. The smart contract stores the hash values of the concatenation and the nonce in a tamper-resistant way. In the reveal phase, the entities send the plain values of the data and nonce to the smart contract. The smart contract checks whether the plain data and nonce match the corresponding hash values stored in the commit phase. If the check succeeds, the contract is executed and the plan values are visible to any other entity with access to the distributed ledger.

(S.2.3) Off-Ledger Data Storage: Another solution to control data visibility is to store the data off-ledger using oracles (see *Oracle Pattern*). Sensitive data are managed by the oracle and are not stored in the distributed ledger. Smart contracts can request information related to data from the services offered by the oracle. On oracles, data confidentiality can be improved by using trusted execution environments (e.g., Intel SGX in Town Crier [80]). Although keeping data off-ledger is most effective for protecting data confidentiality, the use of oracles in smart contracts of distributed ledgers with strong requirements for determinism (e.g., EOS and Ethereum) can be challenging because of the typically encapsulated smart contract execution environment. Moreover, oracles can represent a cause of nondeterminism when providing different data to smart contracts.

(S.2.4) Multi-Ledger Network: To keep data confidentially stored in a distributed ledger, multiple private blockchains can be operated on the same infrastructure. For this purpose, Hyperledger Fabric offers to set up channels between specified consortium members. Only channel members can interact with the corresponding blockchain. However, interactions between smart contracts of different channels are hardly possible in Hyperledger Fabric [52] and thus can exhibit a hurdle for applications. Ethereum and EOSIO do not offer channels like Hyperledger Fabric.

(S.2.5) Front-Running Prevention: Transaction payload visibility can cause vulnerabilities to front-running in Ethereum [81]. In front-running, a transaction T_1 is sent; an

TABLE 6
OVERVIEW OF IDENTIFIED CHALLENGES AND CORRESPONDING SOLUTIONS IN SMART CONTRACT DEVELOPMENT

Origin	Challenge	Solution	Software Design Pattern[2]	DLT Protocol		
				EOSIO	*Ethereum*	*HLF*
Platform	C.1: *Code Visibility*	S.1.1: *Off-Ledger Computations*	Oracle Pattern	X	X	X
	C.2: *Data Visibility*	S.2.1: *Data Encryption*	-	X	X	X
		S.2.2: *Commitment Scheme*	Commitment Pattern	X	X	X
		S.2.3: *Off-Ledger Data Storage*	Oracle Pattern	X	X	X
		S.2.4: *Multi-Ledger Network*	-			X
		S.2.5: *Front-Running Prevention*	Commitment Pattern		X	
		S.2.6: *Private Data Collections*	-			X
	C.3: *Pseudonymity*	S.3.1: *Identity Service*	Identity-Service Pattern	X	X	
	C.4: *Randomness*	S.4.1: *Centralized Randomness Generator*	Oracle Pattern	X	X	X
		S.4.2: *Decentralized Randomness Generator*	Commitment Pattern	X	X	X
	C.5: *Transaction-Ordering Dependence*	S.5.1: *Target-State Definition*	Event-Ordering Pattern	X	X	X
	C.6: *Code Discoverability*	S.6.1: *Name Service*	Name-Service Pattern		X	
	C.7: *Code Updatability*	S.7.1: *Separation of Concerns*	Token Pattern		X	
		S.7.2: *Observation of Addresses*	Observer Pattern		X	
		S.7.3: *Static Entry Point*	Proxy Pattern		X	
		S.7.4: *Static Entry Point with Additional Logic*	Façade Pattern		X	
	C.8: *Execution Restriction*	S.8.1: *Visibility Declaration*	-	X	X	X
		S.8.2: *Account-based Authorization*	Guarding Pattern	X	X	X
		S.8.3: *State-based Authorization*	Event-Ordering Pattern	X	X	X
		S.8.4: *Provisional Authorization*	-	X	X	X
		S.8.5: *Time-based Authorization*	-	X	X	X
		S.8.6: *Smart Contract Deactivation*	Deactivation Pattern		X	
	C.9: *Resource Management*	S.9.1: *Pull-over-Push*	Pull Pattern		X	
		S.9.2: *Continuable Loop*	Indexed-Loop Pattern	X	X	
Programming Language & Execution Environment	C.10: *Undefined Behavior*	S.10.1: *Read the Documentations*	-	X	X	X
	C.11: *Arithmetic Operations*	S.11.1: *Fixed-Point Arithmetic*	-		X	
	C.12: *Concurrency*	S.12.1: *Synchronization*	-			X
	C.13: *Non-deterministic Behavior*	S.13.1: *Cautious Use of Range Iterations*	-			X
	C.14: *Conformity to Expectations*	S.14.1: *Data Type Selection*	-	X	X	
		S.14.2: *Data Type Conversions*	-	X	X	
	C.15: *Cross-Account Interactions*	S.15.1: *Contract Availability Check*	External Call Pattern		X	
		S.15.2: *Gas Limit Specification*	-		X	
		S.15.3: *Check Return Values*	Error-Handling Pattern		X	
		S.15.4: *Instruction Order*	Checks-Effects-Interactions Pattern		X	
		S.15.5: *Execution Locking*	Mutex Pattern		X	
	C.16: *Encapsulation*	S.16.1: *Push Oracle*	-	X	X	
		S.16.2: *Pull Oracle*	Oracle Pattern	X	X	
		S.16.3: *Decentralized Pull Oracle*	Oracle Pattern	X	X	X
	C.17: *Error Handling*	S.17.1: *Isolate Calls*	Façade Pattern		X	
	C.18: *Programming Language Concept Compliance*	S.18.1: *Cautious Use*	-	X	X	X
	C.19: *Iteration through Data Structures*	S.19.1: *Auxiliary Data Structures*	-	X	X	X

X: Challenge and solution apply to DLT protocol HLF: Hyperledger Fabric

TABLE 6 (continued)
OVERVIEW OF IDENTIFIED CHALLENGES AND CORRESPONDING SOLUTIONS IN SMART CONTRACT DEVELOPMENT

Origin	Challenge	Solution	Software Design Pattern	DLT Protocol		
				EOSIO	*Ethereum*	*HLF*
Coding Practice	C20: *Data Storage*	S.20.1: *Off-Ledger Storage*	Oracle Pattern	X	X	
		S.20.2: *Store Data in Logs*	-		X	
	C21: *Data Type Complexity*	S.21.1: `bytes` *over* `byte[]`	-		X	
		S.21.2: *Array Replacement*	-		X	
		S.21.3: `string` *Avoidance*	-	X		
	C22: *Under-optimized Code*	S.22.1: *Constants*	-	X	X	X
		S.22.2: *Code Optimization*	-	X	X	X
		S.22.3: *Shadowing*	-		X	
	C23: *Required Interactions*	S.23.1: *Automated Deployment*	Factory Pattern		X	
	C24: *Readability*	S.24.1: *Style Guide Conformity*	-	X	X	X
	C25: *Ease of Code Reuse*	S.25.1: *Documentation*	-	X	X	X
	C26: *Appropriate Data Type Use*	S.26.1: *Integer Overflow and Underflow Handling*	Overflow/Underflow Pattern		X	
	C27: *Semantic Soundness*	S.27.1: *Argument Sanitization*	-	X	X	X
		S.27.2: *Protection from Replay Attacks*	Replay-Protection Pattern	X	X	
		S.27.3: *Fake-EOS Transfer Protection*	-	X		
		S.27.4: *Fake-EOS Notice Protection*	-	X		
		S.27.5: *Read-Your-Writes (RYW) Consistency*	-			X
	C28: *Technical Soundness*	S.28.1: *Fixed Compiler Version*	-	X	X	X
	C29: *Smart Contract API Conformity*	S.29.1: *Ethereum Request for Comments*	-		X	
			∑Challenges;∑Solutions	22;33	26;52	16;27

X: Challenge and solution apply to DLT protocol HLF: Hyperledger Fabric

adversary reads T_1, creates a concurrent transaction T_2, and sends T_2 with the goal that T_2 executes the smart contract before T_1 to realize a particular benefit. Countering front-running is highly specific to smart contract logic [82]. Still, there are different ways to mitigate front-running. First, developers should minimize the profitability of front-running. Second, a pre-commitment scheme similar to the *Commitment Pattern* can be used, where entities first announce the use of a functionality before calling it [82].

(S.2.6) Private Data Collections: To keep data secret while allowing nodes in a channel to see that a transaction happened, Hyperledger Fabric offers private data collections [83], [84]. Only a defined subset of nodes in a channel can endorse, commit, or query the data of private collections. Private data are stored in a separate and private state database on authorized peers. The state database can be accessed via chaincode. Transactions involving private data store the hash value of the used data on the blockchain so that nodes can check if a state between members exists.

(C.3) Pseudonymity: *The hurdles related to the verification of identity attributes of real-world entities.*

Pseudonymity can cause challenges related to accountability and liability because the actual entities remain unknown [7]. Pseudonyms are hard to associate with corresponding real-world entities, especially in public instances of EOSIO-based or Ethereum-based blockchains. In Hyperledger Fabric, the membership service enables the identification of entities associated with pseudonyms [85].

(S.3.1) Identity Service: To manage entities' digital identities and their associated pseudonyms, prior research has proposed implementations for decentralized identity management (e.g., [86]–[88]). In these implementations, an identity publishes personal information about itself in a decentralized identifier (DID) document and stores the hash value of the DID document on a distributed ledger so that the integrity of the DID document is provable. Real entities can confirm or deny the information contained in the DID document by issuing transactions with verifiable claims that reference the associated DID. Verifiable claims consist of an assertion to express an affirmation or denial of the information in the DID document and an attestation to make the claim verifiable. The more verifiable claims that exist per DID document, the likelier it is that the information contained is accurate [86], [88].

4.1.2 Determinism Challenges

Challenges related to determinism hinder nodes in a DLT network from computing consistent results by following the same protocol.

(C.4) Randomness: *The difficulties of using secure random values in smart contracts.*

Random value generation is challenging in blockchains based on Ethereum, EOSIO, and Hyperledger Fabric due to two main causes. First, nodes in DLT networks independently execute smart contracts in a distributed manner. Nonetheless, all nodes must generate equal random values to preserve determinism [21], [89]. In blockchains based on Hyperledger Fabric, the generation of equal random numbers can be relevant to fulfilling the

```
pragma solidity >=0.6.6 <0.7.0;

contract InsecureRandomness1 {

    function random() public view returns (uint256) {
       bytes32 hash = blockhash(block.number - 5);
       uint256 random_number = uint(hash) % 10 + 1;
       return random_number;
    }
}
```

```
pragma solidity >=0.5.0 <0.7.0;
import "Math.sol";

contract InsecureRandomness2 {
    uint256 seed = 1;

    function random() private view returns (uint8) {
       uint256 x = Math.sin(seed++) * 10000;
       return x - Math.floor(x);
    }
}
```

Fig. 1: ***Insecure examples*** *of implementation for random number generation on a distributed ledger. Both examples allow us to predict and bias random number generation.* ***Do not use these examples in your productive smart contracts.***

individual endorsement policy [90], [91]. In blockchains based on Ethereum and EOSIO that use an order-execute architecture, nondeterminism may cause a consensus to be unreachable [55]. Second, high entropy (i.e., unpredictability and bias-resistance) regarding random value generation is fundamental to achieving a high level of security but challenging to achieve. Environmental variables (e.g., nodes' local timestamps and block hash values; see Fig. 1) should not be used for random value generation because they are predictable or biasable by nodes [89], [92], [93]. Seeds cannot be stored in smart contracts deployed to EOSIO-based or Ethereum-based blockchains because they cannot be kept secret (see *C.1 Data Visibility*). Reaching high entropy in random value generation is also challenging in Hyperledger Fabric.

(S.4.1) Centralized Randomness Generator: To enable randomness in distributed systems while achieving determinism, developers can use oracles (e.g., [94], [95]) like beacons [92], [96] (i.e., services that emit new random data called beacon records at a regular rate) or other distributed ledgers [92]. Beacons (e.g., the NIST beacon service) offer a simple way to integrate random number generation into smart contracts, using the *Oracle Pattern* for implementation. Nonetheless, beacons can centralize DLT applications and can be prone to manipulation. Moreover, values of beacons that periodically change the delivered random values can be reused by multiple smart contracts, and can be exploited by adversaries that first retrieve the random value and use it in an attack until the next random value is generated by the beacon [97].

(S.4.2) Decentralized Randomness Generator: For decentralized randomness generation, the *Commitment Pattern* can be used. In the commit phase, multiple entities send hash values $h(s_e)$ of secretly generated random values s to the Randomness Contract. The Randomness Contract stores $h(s_e)$ of authorized entities $e \in E$, where E is the set of entities registered with the Randomness Contract. In the reveal phase, the entities submit the preimage s to the Randomness Contract. To generate a random number, the Randomness Contract can calculate the XOR result of all submitted preimages as a random value [98]. This approach can be modified by requiring each entity to send coordinates of a point in a 2D-matrix instead of random numbers. Then, the Randomness Contract calculates the polynomial $f(x)$ from all coordinates using barycentric Lagrange interpolation. The Y-axis value in $f(x)$ represents the random number [99]. The decentralized randomness solution avoids single points of failure but requires each entity to interact with the Randomness Contract two times. Thus, the decentralized randomness solution increases the cost and time required for random value generation. Moreover, the last entity sending the plain value can already predict the random number, which can cause vulnerabilities.

To the best of our knowledge, there are still no established best practices for randomness generation in Ethereum-based and EOSIO-based blockchains. When choosing a solution, developers should estimate the cost (e.g., computational resources) of predicting or biasing the outcome of random number generation, and in parallel consider the gains to an attacker.

(C.5) Transaction-Ordering Dependence: *The dependence of smart contract logic on the processing order of transactions.*

In blockchains based on Ethereum, EOSIO, and Hyperledger Fabric, transactions have counters per address so that all transactions issued by an account are processed in a defined order. Transactions issued by multiple accounts can be imagined as concurrent processes [37], making smart contracts vulnerable when relying on a particular transaction order. This class of vulnerabilities is caused by transaction order dependence [7], [100], [101]. Since nodes individually determine the order in which transactions from different accounts are processed, the state in which a smart contract is executed by a particular transaction is unpredictable [102]. Moreover, transaction-ordering dependence favors successful replay attacks (see *S.27.2 Protection from Replay Attacks*). This challenge also applies to Hyperledger Fabric [50], [103].

(S.5.1) Target-State Definition: To counter transaction-ordering vulnerabilities, linearizability and synchronization need to be ensured to guarantee that either the invocation of a function fails or terminates successfully [104]. In accordance with the finite state machine model, function calls can be represented as state transitions. To allow for state transitions only in an intended order, the *Event-Ordering Pattern* recommends implementing checks that only allow for the execution of functions from specified states [102]. In the *Event-Ordering Pattern*, transactions sent to a smart contract carry a nonce that represents the state in which the contract should be executed. Functions of the target smart contract are guarded by checks that deny function execution if the nonce carried in the transaction does not match the current nonce stored in the contract. After each successful function execution, the nonce is changed by the smart contract.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

4.1.3 Maintainability Challenges

Maintainability challenges deteriorate the ease with which deployed smart contract code can be updated, for example, to add functionality, correct flaws, or improve code efficiency.

(C.6) Code Discoverability: *The difficulty of finding smart contracts deployed on a distributed ledger.*

Since the deployment of code costs resources (e.g., gas in Ethereum), it is reasonable to use existing contracts or libraries. However, it is not easy to discover them in blockchains, where smart contract addresses are hard to read and not intuitive (e.g., because they are represented as hexadecimals). EOSIO overcomes this challenge by offering entities the ability to define a human-readable name for their account. In Hyperledger Fabric, all organizations that execute a smart contract on their nodes must manually deploy these contracts and thus are aware of the contract names and where to discover the smart contract code.

(S.6.1) Name Service: While EOSIO offers the definition of human-readable account names associated with a smart contract [11], the addresses of Ethereum accounts are represented by hexadecimals, which are not intuitive for humans to read and recall. To easily look up smart contracts, the use of concise names instead of smart contract addresses is promising. This solution is not necessary in Hyperledger Fabric because all entities know all the IDs of the required smart contracts.

To use names instead of smart contract addresses, a Registry Contract can be put in between smart contracts to handle their interactions (see *Name-Service Pattern*) [26]. Smart contracts can be registered at the Registry Contract, which assigns a unique, user-defined name to a smart contract address or function. Thereby, the address of the latest smart contract version can be looked up.

(C.7) Code Updatability: *The limitations in changing code of deployed smart contracts.*

After smart contract bytecode is deployed to Ethereum-based blockchains, tamper resistance of the blockchain decreases maintainability of the deployed contracts for corrective, adaptive, perfective, and preventive maintenance [105]. If the smart contract code is to be updated, the deprecated version should be deactivated (e.g., using `selfdestruct(…)` in Ethereum smart contracts or the *Deactivation Pattern*), and the current contract version should be deployed. To favor maintainability of tamper-resistant code, developers should strictly apply a separation of concerns and the implementation of mechanisms that ease maintenance (see *Façade Pattern* and *Proxy Pattern*). Challenges and solutions that relate to code updateability apply to Ethereum-based blockchains. Smart contracts deployed to blockchains based on EOSIO or Hyperledger Fabric are not stored in a tamper-resistant manner and can be updated after deployment.

(S.7.1) Separation of Concerns: To improve code updatability, smart contracts can be modularized to decouple the application logic from data. In this notion, the *Token Pattern* separates data (i.e., tokens, balances, and their associated account mapping) stored in a Token Contract from the application logic in a Logic Contract. While the Token Contract provides data about an account's balances without depending on the application using the tokens [106], [107], the Logic Contract serves as an entry point for interactions with the Token Contract. The Logic Contract can be easily replaced with another version.

(S.7.2) Observation of Addresses: If multiple smart contracts interact with one smart contracts (e.g., a Token Contract), developers can implement an Observer Contract (see *Observer Pattern*) [108]. Caller Contracts call Target Contracts. The Caller Contracts register with the Observer Contract and subscribe to address updates of Target Contracts. A developer informs the Observer Contract about an update of a Target Contract by sending the Target Contract's new address to the Observer Contract. The Observer Contract notifies all Caller Contracts about the new Target Contract address, and the Caller Contracts update the new address accordingly. This SDP promises increased efficiency in updating multiple smart contracts. However, it might become costly (e.g., in terms of gas) when many Caller Contracts are called to update the Target Contract address. Since this cost must be taken by the developer initiating the update, this approach is suitable for updating smart contracts that are part of a project.

(S.7.3) Static Entry Point: An alternative and less costly approach is to implement a Proxy Contract with a static address that points to the latest version of a target smart contract and has a similar interface to the target smart contract (see *Proxy Pattern*) [9], [107]. All function calls are made to the Proxy Contract, which forwards the calls to the corresponding function of the target smart contract. If the Target Contract's address changes after an update, only the Proxy Contract must be updated.

To update the address of imported libraries in deployed smart contracts, the use of proxy libraries has been proposed as a workaround [109], which follows a similar concept as the *Proxy Pattern*. When implementing proxy libraries, a regular smart contract is used as a dispatcher to communicate with target libraries. The individual addresses of the libraries can be updated in a storage contract called by the dispatcher. Smart contracts that use a library make a `delegatecall` to the dispatcher contracts, which calls the respective libraries in another `delegatecall`.

(S.7.4) Static Entry Point with Additional Logic: To allow for a rigorous separation of concerns by using different smart contracts while keeping the interaction with the separate contracts simple, the *Façade Pattern* can be used [9]. In the *Façade Pattern*, a Façade Contract serves as a unified interface that manages the interaction with multiple smart contracts. The Façade Contract has functions implemented that facilitate calls to a sequence of external smart contract functions of different smart contracts and handles errors. Thus, the Façade Contract can manage the execution of different modules of an application logic implemented in separate smart contracts. All smart contract addresses registered with the Façade Contract are updatable independently.

Although these mechanisms offer different ways to make smart contracts maintainable, it is important to consider

that the tamper resistance of smart contracts is a unique DLT characteristic and an anchor of trust into reliable enforcement of agreements, which should not be mitigated by exhaustive maintainability of smart contracts.

4.1.4 Regulated Executability Challenges

Challenges concerning regulated executability impact the mechanisms put in place to regulate the execution of smart contract code.

(C.8) Execution Restriction: *The undesired executability of smart contract functions by entities that can interact with the distributed ledger.*

In several distributed ledgers (e.g., Ethereum and EOS), smart contracts are exposed to all nodes in a DLT network. Thus, contracts can become subject to undesired function calls. For example, undesired function calls to the `selfdestruct(address a)` function in Ethereum smart contracts are of particular criticality, as seen in the *Parity hack* [13], [110]. After `selfdestruct(address a)` is executed, all balances kept by the smart contract account are transferred to `a`. Then, the smart contract is locked and cannot be executed anymore.

(S.8.1) Visibility Declaration: Developers should carefully declare whether functions should be callable by the identities of the distributed ledger (e.g., using `external` in Solidity [111]) or only by the smart contract itself or in its execution context (e.g., declaring a function's visibility `private` in Solidity) [112], [113].

(S.8.2) Identity-based Authorization: To prevent unauthorized execution of smart contract functions, functions can be guarded by authorization checks that ensure that only specified accounts can execute functions in the intended context [9], [102], [106]. Therefore, function execution can be restricted to specific accounts [106], [108]. Despite different implementations for account-based authorization, these approaches follow a similar structure: when a function is called, the identity is authenticated, and its permission for the function execution is checked for authorization (see *Guarding Pattern*). In EOSIO, entities manage permissions via authorization tables. In authorization tables, the `eosio.code` permission is of particular importance because all entities whose accounts have the `eosio.code` permission can transfer assets from that account [11]. In Ethereum smart contracts, developers should use `msg.sender` to identify the account that issued the original transaction for a function call, especially when smart contracts make external calls [89]. In EOSIO, identity-based authorization per function is at the core of the DLT protocol, and developers must use the authority table that corresponds to their smart contracts to specify permissions of accounts [46]. In Hyperledger Fabric, identity-based authorization is largely managed via the definition of endorsing peer nodes in the chaincode.

(S.8.3) State-based Authorization: Functions can be protected by ensuring that accounts can only execute functions in a particular state using the *Event-Ordering Pattern* [9], [102], [106]. In the *Event-Ordering Pattern*, a state variable *s* is defined and initialized with a nonce. The value of *s* indicates a particular state of the smart contract and is changed after each successful function call associated with the state transition of the smart contract. To successfully execute a function, the transaction invoking the function must pass the current value of *s* as an argument. Otherwise, the function invocation is denied.

(S.8.4) Provisional Authorization: Function execution can be restricted to entities knowing a certain secret (e.g., secret preimage of a hash [108]). For provisional authorization, hash values can be stored in a smart contract. For function invocations, entities must pass the preimage of a stored hash value to the smart contract. If the hash value of the preimage included in the transaction matches the stored hash value required for authorization, the function call proceeds. Otherwise, the call is denied. Each hash value must only be used for a single authorization because the preimages in the transactions are publicly visible. Nonetheless, this solution is prone to front running and needs additional protection [82]. For example, accounts can be associated with individual hash values, and hash values can only be used for authorization when the transaction is sent from the associated accounts.

(S.8.5) Time-based Authorization: Function execution can be restricted to time intervals (i.e., speed bump [8] or automatic depreciation [9]) to prevent a rush of transactions. Whenever a target smart contract receives a transaction, it first checks whether the timestamp of the transaction issuance is within the period that allows the execution of a smart contract function. Otherwise, the smart contract denies the call. When using this solution, developers should be aware of the degree to which nodes' local timestamps can be manipulated.

(S.8.6) Smart Contract Deactivation: In Ethereum, smart contracts can be disabled using `selfdestruct(…)` or deactivated (see *Deactivation Pattern*). If a smart contract is disabled, all asset transfers to the contract's account will get lost [114]. Instead of disabling a contract, developers can deactivate the contract by changing the value of an internal state variable. After the value is changed to deactivated, all incoming requests will be reverted. Thereby, no assets will get lost in regular asset transfer to a deactivated smart contract account, but the contract is still not usable anymore [114]. Regular means that assets are not transferred in the context of executing `selfdestruct (…)` in the caller Ethereum smart contracts.

(C.9) Resource Management: *The limitations regarding the execution of smart contract functions caused by the corresponding allocation and revocation of computational resources by control mechanisms put in place.*

The mechanisms to guarantee the termination of smart contracts mostly limit smart contract execution by a specific resource, such as gas in Ethereum or execution time in EOSIO. After the resource is consumed, the execution is aborted. The abortion of function execution can cause denial of service, for example, in unbounded mass operations. Unbounded mass operations can occur, for example, when entities can add new addresses to the `balanceList` array in the Ethereum smart contract illustrated in Figure 2. Addresses kept in the `balanceList` array are used for payouts initiated by calling the `payout()`

```
1  pragma solidity >=0.5.0 <0.7.0;
2
3  contract Overflow {
4      address[] balanceList;
5      mapping(address => uint256) balances;
6
7      // Your code including a function to add addresses
8
9      function payout() {
10         for (uint8 i = 0; i < balanceList.length; i++) {
11             if(balances[balanceList[i]] > 0) {
12                 uint8 balance = balance[balanceList[i]];
13                 balances[balanceList[i]] = 0;
14                 balanceList[i].transfer(balance);
15             }
16         }
17     }
18 }
```

Figure 2: Example of an unbounded data structure in Solidity that may run into an infinite loop because of an integer overflow.

function. While looping through the `balanceList` array, the execution may run out of gas or exceed the block gas limit in Ethereum. Accordingly, the transaction is reverted, and the address kept in the `balanceList` array will receive payments leading to denial of service [14], [20], [115]. Challenges related to resource management apply to Ethereum-based and EOSIO-based blockchains [14], [44], [116], [117]. In Ethereum, unlike Solidity, Vyper defines an upper bound on gas consumption per function call to prevent DoS from operations on unbounded data structures [115].

(S.9.1) Pull-over-Push: To counter challenges caused by unbounded mass operations on EOSIO-based and Ethereum-based blockchains, developers can use pull mechanisms (see *Pull Pattern*). Pull mechanisms require every entity to call the smart contract themselves, for example to receive payments via `payout()`. This way, the account the transaction has been issued from only pays gas for their own payouts. Although the *Pull Pattern* is especially proposed for payments, it also applies to other unbounded data operations [14]. However, pull payments can decrease the utility of DLT applications because each account must individually invoke the smart contract.

(S.9.2) Continuable Loop: If loops over unbounded arrays cannot be avoided, developers should keep track of the progress inside the loop. This allows the loop to continue in the next call at the last entry before the iterations are aborted [14]. To make the execution of a loop continuable, an index pointing to the index of the last successful iteration can be used (see *Indexed-Loop Pattern*). When resuming the loop, it continues at the entry after the last successful iteration.

4.2 Challenges Caused by the Programming Language & Execution Environment

Challenges related to an offered programming language and execution environment for smart contracts refer to the limitations and shortcomings of the technical conditions offered to develop and execute smart contracts.

4.2.1 Language Definition Completeness Challenges

Challenges pertaining to language definition completeness relate to the incomplete coverage of a formal model in the definition of a programming language and the resulting undefined behaviors of smart contracts.

(C.10) Undefined Behavior: *The shortcomings in the specification of the behavior of a programming language.*

Undefined behavior of a smart contract occurs when a language's definition of particular operations is ambiguous or non-existent, and the smart contract relies on these underspecified operations, altering its actual semantic intent at compile time [118]. In Solidity, for example, the order in which expressions are evaluated in the same statement is not specified [111].

(S.10.1) Read the Documentations: Undefined behavior can have individual effects on smart contract execution depending on the specific implementation. Developers should be aware of ambiguous or missing language definitions to avoid unexpected program flow.

4.2.2 Theoretical Expressiveness Challenges

Challenges related to theoretical expressiveness are concerned with the lack of functional capabilities offered by a programming language or its execution environment.

(C.11) Arithmetic Operations: *The limitations and vulnerabilities related to using arithmetic operations.*

Arithmetic operations can lead to truncation errors or undefined behavior that can result in the loss of assets. Truncation errors can occur in Solidity, for example, when dividing numeric values because Solidity only supports integer values. Challenges related to arithmetic operations apply primarily to Solidity and EVM, which do not natively support floating-point data types [43]. EOSIO uses *softfloat* from the IEEE-754 float-point arithmetic [43] supporting deterministic rounding behavior. In Hyperledger Fabric, arithmetic operations offered by supported programming languages can be used because nondeterministic behavior is filtered by applying endorsement policies.

(S.11.1) Fixed-Point Arithmetic: Developers can use fixed-point arithmetic to avoid truncation errors to a certain extant [119], [120]. To express a fixed-point number, developers must specify a fixed number of digits after the decimal point. When using this solution, developers must consider interactions with the smart contract with other contracts or oracles and convert numeric values according to the individual specifications. As an alternative to Solidity, Vyper supports decimal fixed point numbers [121]. Still, fixed-point arithmetic can be prone to truncation errors when not handled appropriately.

(C.12) Concurrency: *The protection from nondeterministic behavior caused by code that is executed with time overlaps.*

In addition to the concurrency between transactions of different accounts regarding their processing order (see *C5 Transaction-Ordering Dependence*), concurrency can occur during the execution of smart contracts causing nondeterminism. Concurrency is a challenge in Hyperledger Fabric smart contracts that are programmed in Go [7]. Go is designed for parallel execution and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

supports concurrent execution using goroutines, which are functions that run concurrently with other functions. Concurrency is not a challenge in EOSIO-based and Ethereum-based blockchains using the EOSVM or the EVM.

(S.12.1) Synchronization: To avoid nondeterministic behavior in smart contracts programmed in Go, developers can synchronize the execution of goroutines within their contracts [122]. For synchronization, Go offers the package `sync`, including `WaitGroup`, for low-level library use [123]. Using `WaitGroup`, Go code waits for a collection of Go subroutines to finish before continuing with subsequent computations [122].

(C.13) Non-Deterministic Behavior: *The use of operations offered by a programming language that returns arbitrary results.*

Several general-purpose programming languages behave nondeterministically for the execution of particular functions [7], [124]. This behavior contradicts the requirements for determinism of most DLT protocols (see Table 1). This challenge applies to smart contracts developed in Go, which is currently only offered for Hyperledger Fabric smart contracts. For example, in Go's type collections, range iterations over maps return values in random order, which challenges deterministic smart contract execution [124].

(S.13.1) Cautious Use of Range Iterations: Developers should avoid using nondeterministic constructs if their use can affect deterministic function execution.

4.2.3 Usability Challenges

Challenges related to usability are concerned with the hurdles faced by developers when using a programming language.

(C.14) Conformity to Expectations: *The mismatch between developers' expectations of how their program should be executed and its actual execution.*

Solidity offers the declaration of different integer types (e.g., `uint8`, `uint32`, or `uint256`) that resemble those in programming language *C*, which can lead novice developers to assume that an `uint8` would allocate 8 bits in memory, while an `uint128` would allocate 128 bits. However, the EVM uses simple `(key, value)` storage, where each value consumes 256 bits. Variables declared as `uint8` even consume more gas than `uint256` variables because of additional operations performed to downscale `uint8` variables from `uint256`. Thus, integer types of Solidity are not entirely consistent with the EVM [118], [125], which may lead to bugs or underestimated costs. Another example of weak typing in Solidity is the instantiation of smart contracts within a contract. If a smart contract SC_{caller} refers to an instantiation of another smart contract SC_{callee} using SC_{callee}'s address, it is not checked whether the smart contract instance stored on the particular address complies with the type declaration of SC_{callee}. Moreover, data type conversions of variables storing a very large `uint` value to `int` or variables storing a negative `int` value to `uint` can cause unexpected results because Solidity uses two's complement to represent `int` [118], [126]. The following solutions should also be considered for blockchains based on EOSIO:

(S.14.1) Data Type Selection: To resolve discrepancies related to the conformity of programming languages to their execution environment, we advise smart contract developers to carefully read the documentation of the programming language and the targeted execution environment to decide on the data types to be used. In Solidity, developers should gauge whether the benefits of using unsigned integers other than `uint256` exceed the costs caused by increased gas consumption.

(S.14.2) Data Type Conversions: When using type conversion from a larger data type (e.g., `uint256`) to a smaller one (e.g., `uint8`), developers should first thoroughly test their code to ensure that conversions do not decrease accuracy.

(C.15) Cross-Account Interactions: *Code flaws that are caused by a call from a smart contract that involves external sources, such as other smart contracts.*

In Ethereum, there are three types of issues that can be caused by cross-account interactions (also called external calls) [21], [22]: first, *unavailable smart contract;* second, *function not found; and* third, *unintended function call.* In *unavailable smart contract*, the target smart contract does not exist or has been destroyed. The EVM does not throw an error if a transaction's recipient does not exist. In EOSIO, *cleos* generates an error message if an action does not comply with the definitions of the functions or the smart contract name. In Hyperledger Fabric, clients are notified through an error message if a target smart contract could not be found. In *function not found*, the interface of the smart contract or library does not match the signature of the function to be called through the transaction. In this case, the target smart contract function cannot be found. If the function cannot be found, the smart contract's fallback function [3] is invoked, which can implement arbitrary procedures. In Solidity, no exception is thrown if a function is not found and the caller is likely to be unaware of the error [21]. If an entity issues a transaction to a non-existing EOSIO account or calls a function that is not implemented in the target contract, an HTTP 404 error is returned, and the transaction is rolled back. The Hyperledger Fabric protocol first checks whether the target function exists in the smart contract before trying to execute the function. Calls to functions that do not exist in the smart contract trigger an unknown transaction handler [127]. In *unintended function calls*, a recipient of funds unintentionally invokes a function (e.g., from its constructor or fallback function). For example, when a smart contract transfers an asset to a recipient smart contract, the recipient smart contract may have a procedure implemented (e.g., in its fallback function), which is executed upon receiving the assets. This procedure

[3] In Solidity, each smart contract can implement a fallback function, which is called when the function signature does not match any function in the smart contract. If no fallback function is given in these situations, the EVM throws an exception.

calls for a function in the original contract. Such a sequence of function calls is called reentrancy. Challenges caused by cross-account interactions apply to smart contracts developed for Ethereum-based blockchains. Smart contracts for blockchains based on EOSIO and Hyperledger Fabric are not prone to vulnerabilities caused by unavailable smart contracts or calls to non-existing functions. Moreover, the processing of instructions in smart contracts on blockchains based on EOSIO or Hyperledger Fabric prevents reentrancy.

(S.15.1) Contract Availability Check: To check whether a smart contract is available (i.e., existing and not destroyed), smart contracts of Ethereum-based blockchains can load code associated with a target address into a `bytes` variable using Solidity Assembly (see *External-Call Pattern*) [128]. If the length of the value stored in the `bytes` variable is larger than zero, the address is associated with a callable smart contract. However, it cannot be uniformly checked whether the smart contract complies with an expected data type. In EOSIO since the Dawn 4.0 update , the availability of an account is automatically checked in the `eosio.token` contract [129].

(S.15.2) Gas Limit Specification: In Solidity, there are three ways to transfer native assets (e.g., Ether) from a smart contract:

(1) `<recipientAddr>.send(value)`
(2) `<recipientAddr>.transfer(value)`
(3) `<recipientAddr>.call.value(value)("")`

When using (1) or (2), a fixed amount of exactly 2,300 gas is forwarded to the recipient, which should protect smart contracts from reentrancy (as of March 2021) [20]. If an out-of-gas exception is thrown in asset transfers, (1) only returns `false` and errors must be handled manually. In contrast, (2) further propagates the exception and automatically reverts the callchains of the failed transactions, which is similar to `require(<address>.send(…))`. Since the Istanbul hard fork, it is known that gas costs for instructions are not constant and (1) and (2) may fail in the future. To counter failed asset transfers in the future due to increased gas costs, developers should use (3), which forwards all available gas to the recipient contract [130]. However, using (3) can make a smart contract vulnerable to reentrancy, which is why developers must also implement mechanisms to protect the contracts from corresponding attacks (see *S.16.4 Protection from Reentrancy*). If the execution of the target contract runs out of gas when using (3), the function returns `false` and error handling must be manually performed similar to (1). Vyper offers `send(recipientAddr, value)` to transfer assets, which works similar to (1) [115] and, thus, is prone to failed transactions caused by volatile gas costs.

(S.15.3) Check Return Values: In favor of proper error handling, the *Error-Handling Pattern* recommends that developers implement return values in all functions so that their successful execution can be determined [10], [131]. This recommendation is especially important for Ethereum smart contracts. Checks of return values are automatically added for calls in Vyper so that failed calls are automatically reverted [115].

(S.15.4) Instruction Order: We identified four types of reentrancy attacks caused by external calls by Ethereum smart contracts [12], [22], [132]: *fallback reentrancy*, *cross-function reentrancy*, *delegated reentrancy*, and *create-based reentrancy*. In *fallback reentrancy*, a smart contract transfers assets to another contract. After receiving the assets, the recipient contract calls the function in the original contract that transferred the assets again from its payable or fallback function. In *cross-function reentrancy*, a smart contract function is invoked and reentered through another function, while the smart contract is still in an inconsistent state. Attackers can perform cross-function reentrancy if a smart contract includes functions that read from or write to the same variables [132], [133]. *Delegated reentrancy* occurs when a smart contract imports a library and state updates are not synchronized appropriately [132], [133]. *Create-based reentrancy* can occur if a smart contract *A* invokes the constructor of another contract *B* before updating its state. During the execution of *B*'s constructor, *B* can call a function in *A*, causing reentrancy [132].

The *Checks-Effects-Interactions Pattern* defines an execution order for instructions in a smart contract function. First, it is necessary to check if the context is valid to execute the function. Second, all changes are to be applied to the values of relevant variables. Third, the function execution can proceed. Following this execution order, malicious smart contracts cannot reenter the same function again in the previous state [9], [132].

(S.15.5) Execution Locking: As an alternative to the *Checks-Effects-Interactions Pattern*, the *Mutex Pattern* can be used to protect smart contracts from reentrancy attacks. In the *Mutex Pattern*, the state of a smart contract is locked using a mutex variable when the execution of logic to be protected starts. After the particular logic is executed, the code is unlocked again using the mutex variable [8], [102]. If an attacker performs a reentrancy attack within the scope of the execution of locked protected logic, the execution of the reentrancy call is aborted when passing the check of the locked mutex variable. Developers can apply checks for the mutex to every function of the smart contract [102]. However, the *Mutex Pattern* can become very complex when trying to prevent reentrancy across multiple function calls and can become prone to programming flaws that, for example, allow attackers to lock a smart contract for an arbitrary time or even forever [82]. In Vyper, the `@nonreentrant(<unique_key>)` decorator corresponds to the *Mutex Pattern* and can be used to protect functions from reentrancy [134].

(C.16) Encapsulation: *The limitations of smart contracts in interacting with data and information systems external to the execution environment.*

To request external data (e.g., sensor data) or move the execution of computation to oracles [70], [71], [107], smart contracts must interact with oracles. Because of the requirement for determinism in blockchains building on Ethereum and EOSIO (see Table 1), these DLT systems encapsulate smart contract execution in virtual machines (i.e., the EOSVM and EVM) that prevent direct calls to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

oracles [43]. In Hyperledger Fabric, direct interaction from smart contracts with oracles is supported [135]. Still, the following solutions also apply to Hyperledger Fabric when gauging how to integrate reliable oracles.

(S.16.1) Push Oracle: To retrieve external data from smart contracts on EOSIO-based and Ethereum-based blockchains despite their encapsulated execution, a Relay Contract can be instantiated that is periodically updated by an oracle[136]. The Relay Contract stores current data, and other smart contracts can retrieve said data. However, this approach can be inefficient due to unnecessary updates to the Relay Contract. Moreover, this approach can introduce vulnerabilities (e.g., toward malicious behaviors of oracle controllers) due to the reliance on individual third parties.

(S.16.2) Pull Oracle: To make push oracles more efficient in terms of required interactions, we propose event-driven updates, for example, using events in the Relay Contract, such as native Solidity events, a listener plugin for EOSIO (e.g., EOS Watcher Plugin [137]), or periodic polls of *nodeos* [138]. The oracle listens to requests triggered by the Relay Contract for specified events (see *Oracle Pattern*). Such requests may refer to arbitrary computational tasks, for example, data storage, data retrieval, or outsourcing of computations [80], [139]. The oracle manages the execution of the requested tasks. Then, the oracle pushes the results to the Relay Contract. Unfortunately, this approach comes with the downside that the oracle is operated by a third party, forming a single point of failure [70], [71].

(S.16.3) Decentralized Pull Oracle: To tackle malicious behavior of oracles while increasing their availability and reliability, developers can use decentralized oracles [71]. Multiple oracles listen to the Relay Contract (see *Oracle Pattern*), process received requests, and push their results to the contract. The Relay Contract decides on one result to use among those provided by the oracles, for example, by choosing the result that has been returned by most oracles. As an extension, an incentive mechanism should be put in place to avoid malicious behavior of oracles (see *Oracle Pattern*) [139], [140]. For example, oracles can pay collateral when registering with the Relay Contract. After oracles push valid results to the smart contract, they are rewarded with coins, while oracles that push wrong results are punished by reducing their collateral.

(C.17) Error Handling: *The difficulties of implementing thorough handling of errors and exceptions in smart contract execution.*

In Ethereum, inappropriate error handling can cause undesired smart contract states and can even lead to asset loss and denial of service [14]. Appropriate error handling is, however, challenging because error handling strongly depends on the individual call chain. A call chain describes the sequence of function calls performed during smart contract execution. In Solidity, a call chain can include different types of calls (i.e., `call`, `delegatecall`, and `staticcall`; see Section 2.2). The EVM propagates exceptions up the call chain and reverts all side effects until the last `call` command, which returns `false`. The smart contract execution is resumed from this point, and only the gas allocated by the `call` command is consumed [21].

(S.17.1) Isolate Calls: To minimize the potential damage caused by flawed error handling for complex call chains in the EVM, developers should isolate separate external calls in Ethereum instead of chaining calls. This way, developers can implement more granular error handling. To orchestrate multiple isolated calls, the *Façade Pattern* applies.

(C.18) Programming Language Concept Compliance: *The degree to which a programming language conforms to established concepts and the use of terms in related programming languages.*

In smart contract development, protected keywords (e.g., *private* or *public*) in established programming languages, such as Java or C++, can mislead developers. For example, visibility declarations in Solidity (e.g., *external*, *private,* or *public*) often suggest to developers that private variables may not be visible to other entities [20].

(S.18.1) Cautious Use: If developers have no other opportunity to develop code than using a language with misleading keywords, developers must be cautious.

(C.19) Iteration through Data Structures: *The functionality provided by a programming language to support the step-by-step traversal of individual elements of a higher-level data structure.*

As in conventional software development, there are data structures that are not iterable but that can store and return data from a collection in *O(1)*.

(S.19.1) Auxiliary Data Structures: To loop through non-iterable data structures (e.g., `mapping` in Solidity), auxiliary data structures (e.g., an `array`) can be used. Auxiliary data structures should be iterable and store all keys of the non-iterable data structure. When iterating over the auxiliary data structure, its current value can be used as a key to retrieve values of the non-iterable data structure.

4.3 Challenges Caused by Coding Practices

Challenges related to the principal challenge origin *coding practices* refer to issues caused by developers in their coding activities.

4.3.1 Code Efficiency Challenges

Code efficiency challenges refer to the constrained quantity of allocated resources to deploy a smart contract code (e.g., gas) and execute the deployed code on a distributed ledger (e.g., in terms of space and time complexity).

(C.20) Data Storage: *The storage of data to keep a smart contract operational in a trustworthy manner, but also efficient with respect to resource consumption for smart contract execution.*

Storing data on EOSIO-based and Ethereum-based blockchains is expensive. Therefore, developers must consider alternatives to storing data in smart contracts. This challenge does not apply to blockchains based on Hyperledger Fabric.

(S.20.1) Off-Ledger Storage: Like heavy computations

and sensitive data, large amounts of data should be stored off-ledger and should be managed on oracles. The *Oracle Pattern* (see *S.1.1 Off-Ledger Computations*) describes how smart contracts can interact with oracles. To make the integrity of stored data provable, developers can implement mechanisms to store a hash value of the externally stored data in the smart contract.

(S.20.2) Store Data in Logs: Solidity offers the implementation of events that are usually used to communicate with oracles or frontends [141]. Events can be used to generate logs that represent a cheap alternative for storing data on a blockchain because a log costs only 8 gas per byte (at the time of writing the study). However, logs are not accessible by smart contracts and only oracles or frontends can render logged data. Using events as logs can be reasonable, for example, when operating an exchange. The history of entities' deposits to a smart contract does not need to be stored in the contract but can be stored as logs, and only the current balances of the entities are stored in the smart contract [141].

(C.21) Data Type Complexity: *The differences between data types with similar functionalities regarding their time and space complexity.*

The selection of appropriate data types affects the cost of storage and execution in smart contracts based on Ethereum and EOSIO. To provide efficient code, developers must gauge between different data types. Still, the variety of data types and their individual complexities regarding storing and retrieving data differ strongly.

(S.21.1) `bytes` *over* `byte[]`*:* In Solidity, the data type `byte[]` is an array of bytes but requires 31 bytes of memory between its elements because of padding rules. Developers can use data type `bytes` to reduce memory consumption [142].

(S.21.2) Array Replacement: In Ethereum, using arrays can be more costly than using individual variables [113]. To save gas, developers should check whether they can replace arrays of fixed length with a corresponding number of individual variables.

(S.21.3) `string` *Avoidance:* In EOSIO, developers should avoid storing variables in strings to save resources. For example, saving SHA3-256 hash values as `checksum` instead of `string` can reduce memory consumption from 64 bytes to 32 bytes. The same applies to 128-bit numbers, such as common universal unique identifiers (UUIDs), which are typically represented as hex-string allocating 16 bytes. In contrast, storing 128-bit numbers as string consumes 36 bytes.

(C.22) Under-Optimized Code: *The optimization of smart contract code toward better performance.*

A recurring problem in software engineering is inefficient code. Code can be inefficient due to useless code (e.g., opaque predicates [117], [143] or dead code [143]) or code smell (e.g., repeated operations in loops with constant outcome [113], [143]). Useless code consumes additional resources (e.g., gas or RAM) without adding reasonable logic to the smart contract. For example, dead code will never be executed, but costs gas for deployment in Ethereum smart contracts or RAM in smart contracts on EOSIO-based blockchains. Useless code in combination with loops can significantly increase resource consumption, for example, when functions with constant outcomes are repeatedly executed within a loop. Dispensable code is particularly important to avoid in smart contracts running via DLT protocols that charge costs for smart contract execution, such as in Ethereum-based and EOSIO-based blockchains.

(S.22.1) Constants: Developers should check if they perform computations with constant outcome. When identifying an opaque predicate, developers can declare the result of the computation as `constant`.

(S.22.2) Code Optimization: To reduce resource consumption when executing smart contracts, developers should check whether variables are required to produce a particular result and dispense with variables that are not required. Additionally, the necessity for functions in a smart contract should be checked to avoid opaque predicates and code smell. To identify dead code and opaque predicates, software tools for formal verification can help (e.g., GASPER [143]). For the optimization of Ethereum smart contracts, existing works present approaches to identifying and performing bytecode improvements [113], [117], [144]. Using these approaches, dispensable operations can be identified, and the smart contract can be optimized.

(S.22.3) Shadowing: Developers should avoid processing data in the persistent storage of the EVM to reduce resource consumption [145]. Instead, developers can apply shadowing. In shadowing, the data to be sorted is copied from the storage into the EVM memory, which is less resource-consuming than sorting in storage [146]. All sorting is performed in the EVM memory.

(C.23) Required Interactions: *The minimization of the required interactions with a smart contract to achieve a targeted result.*

Distributed ledgers enable the management of digital assets without the necessity of a trusted third party to a certain extent. It is possible to represent ownership of real-world assets (e.g., cars or houses) using tokens, which can be implemented as a smart contract [147]. However, it is challenging to create and deploy such tokens manually. For example, an authorized entity responsible for the token must create and deploy a smart contract for each requesting user individually, which poses a single point of failure and a potential source of fraud.

(S.23.1) Automated Deployment: To increase security regarding the creation and deployment of smart contracts (e.g., in terms of token creation and issuance, fraud resistance, and theft), developers can use the *Factory Pattern*. In the *Factory Pattern*, a smart contract (i.e., *Factory Contract*) manages the creation and issuance of such smart contracts (i.e., *Child Smart Contracts*) [148]. This pattern is consistent with the concept of factories in existing programming languages, such as Java. In addition to the automated creation and deployment of smart contracts, Factory Contracts can also implement mechanisms to better observe issued smart contracts, for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

example, by storing the addresses of all created smart contracts. This solution applies to Ethereum-based blockchains.

4.3.2 Comprehensibility Challenges

Comprehensibility challenges relate to the ease with which entities with little experience understand how a specific smart contract code works and how it should be used.

(C.24) Readability: *The hurdles faced by developers when reading program code.*

Readability of program code across smart contracts developed by different entities is important for increasing its comprehensibility and maintainability [20], [101], [149]. However, developers have different programming styles, which can decrease readability and comprehensibility.

(S.24.1) Style Guide Conformity: To favor code readability, developers should align their individual coding style with the style guides published for the programming language (e.g., the Solidity style guide for Ethereum [150] or the Go style guide for Hyperledger Fabric [151]). For example, Ethereum developers should align with established best practices for naming events [20]. In EOSIO, the naming of accounts is already regulated. Style guide conformity can be checked automatically by software tools such as *Ethlint* or *Solhint*.

(C.25) Ease of Code Reuse: *The ease with which developers can inform themselves about the characteristics of smart contract code to understand contract specifications for better code reuse.*

Since code reuse in publicly distributed ledgers (e.g., those based on Ethereum or EOSIO) is often performed [101], it is particularly important that developers can easily understand the purpose and functioning of code, as well as its shortcomings.

(S.25.1) Documentation: To support others in reusing code, developers should add appropriate documentation in the form of comments or additional files (e.g., `README.md` files). The documentation should include the functioning of the smart contract and report known shortcomings (e.g., bugs or vulnerabilities).

4.3.3 Implementation Soundness Challenges

Challenges related to implementation soundness originate from factors that hinder an implementation from being free from errors and flaws.

(C.26) Appropriate Data Type Use: *The degree to which developers appropriately declare, initialize, and use variables.*

To support developers in the selection of data types, data type inferencing is offered by several programming languages (e.g., C++ and Solidity). Data type inference refers to the automatic recognition of a data type likely suitable for storing a given value and can expose a vulnerability in programming languages, for example, due to overflow or underflow [20], [152]. An overflow describes the behavior of programming languages when a value exceeds the boundary of a data type (e.g., assigning numeric values larger than their defined maximum of 2^8 - 1 to `uint8` variables). An underflow occurs when a value assigned to a variable is less than the smallest defined value that can be represented by the variable's data type. To avoid unforeseen code flaws, developers should be aware of the different processing of data types in storage.

(S.26.1) Integer Overflow and Underflow Handling: Overflow and underflow can occur in Ethereum smart contract programmed in Solidity. To counter overflow and underflow, developers should not rely on data type inferences but should define the targeted data type completely [118]. For example, if a variable is declared as `uint8` through data type inference, this variable will overflow if it is assigned a value larger than 2^8-1. Moreover, developers can either manually implement checks for overflow and underflow or use the OpenZeppelin SafeMath library [153] for any arithmetic operations a smart contract performs [118]. Using the SafeMath library can also prevent most overflows and underflows of integer variables (see *Overflow/Underflow Pattern*). Since Solidity v0.80, Solidity checks for overflow and underflow and reverts arithmetic operations [154]. Alternatively, developers can use Vyper [121] instead of Solidity because Vyper is not prone to overflow and underflow [115].

(C.27) Semantic Soundness: *The difficulties of reaching a state where an implementation is free from logical errors and flaws.*

To reach semantic soundness, the implementation should adhere to the agreed-upon business logic for interaction [12] with respect to the *absence of logic*, *incorrect logic*, and *logically correct but unfair* [12]. First, *absence of logic* describes smart contracts that lack important logic, for example, to protect its `selfdestruct(…)` function from being unintendedly executed by attackers [12]. Second, *incorrect logic* is concerned with a smart contract code that is syntactically correct but logically incorrect. Third, *logically correct but unfair* applies to code that is free from errors but misleads entities so that they will be subject to fraudulent program logic (e.g., expected payouts that will never happen, as in Ponzi Schemes [155]).

(S.27.1) Argument Sanitization: As in conventional software engineering, passing inappropriate arguments to functions can cause errors or unforeseen side effects. This also applies to blockchains based on Ethereum, EOSIO, and Hyperledger Fabric. In Ethereum asset transfers, for example, the EVM pads short addresses with trailing zeroes if the recipient address is too short. The padding can result in the transfer of a larger number of tokens than intended [156]. To prevent wrong arguments from being processed, developers can implement guarding functions that first check passed arguments upon function invocation and deny the function execution if one of the arguments does not comply with the function requirements (e.g., using `assert(…)`, `require(…)`, or `revert(…)` in Solidity [20]). For example, the length of an Ethereum address passed to a smart contract should be checked before transferring assets to it to prevent asset loss [156].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2021.3116808, IEEE Transactions on Software Engineering

(S.27.2) Protection from Replay Attacks: In Ethereum-based and EOSIO-based blockchains, the visibility of the payload of transactions favors the exploitation of smart contract vulnerabilities for replay attacks [7]. In a replay attack, an adversary copies the content of a transaction payload to their own transaction and issues their fraudulent transaction to the same smart contract. The target smart contract receives the original and the fraudulent transaction and respectively executes the target function. Often, the copied transaction payload contains data for authentication (e.g., a digital signature). With these, critical logic for an account can be executed (e.g., asset transfers). To prevent replay attacks, the *Replay-Protection Pattern* can be used. In this SDP, a function call may require a digital signature of all other parameters passed to the function and the current value of a nonce defined in the targeted smart contract. When the function is called, the smart contract verifies the signature based on the passed function parameters and the nonce. After successful verification, the nonce is changed and future transactions with the same signature become invalid [115]. In Hyperledger Fabric, peer nodes implement mechanisms to protect the network from replay attacks [54]. This solution is particularly relevant to consider when working on publicly distributed ledgers that are in the stage of a hard fork [157]. Valid transactions can be easily replayed from one ledger to another. To counter replay attacks in this scenario, a chain ID should be a required inclusion in the digital signature. The chain ID takes the function of the nonce.

(S.27.3) Fake-EOS Transfer Protection: In *Fake-EOS Transfer* attacks, an attacker creates a token called *EOS* like the native currency in EOSIO. Then, the attacker sends their fake EOS tokens to a smart contract. If the recipient contract does not verify the issuer of the tokens, it considers them genuine EOS tokens and proceeds with the function execution.

To protect smart contracts from *Fake-EOS Transfer* vulnerability, smart contracts should verify that the asset transfer has been authorized by the `eosio.token` contracts. For this purpose, developers can check whether the `code` parameter in the `apply(…)` function of the recipient contract refers to the `eosio.token` contract [11], [158].

(S.27.4) Fake-EOS Notice Protection: Smart contracts in EOSIO-based blockchains receive a notification as soon as an asset transfer via the `eosio.token` contract is completed. These notifications can be forwarded to other smart contracts. If the notification is not checked by these smart contracts, they may proceed as if they had received the funds. This way, EOSIO smart contracts become vulnerable to *Fake-EOS Notices* [11], [158]. In *Fake-EOS Notice*, an attacker sends tokens to smart contract *A*. The token transfer is handled by the `eosio.token` contract, which notifies *A* and *B* about the token transfer. Upon receiving the notification, *B* forwards the notification to a smart contract *C*. *C* handles the notification as if it had received the tokens.

To protect smart contracts from being prone to *Fake-EOS Notice*, developers can check if the `to` argument in the notification equals their own account; if not, *C* ignores the notification [11], [158].

(S.27.5) Read-Your-Writes (RYW) Consistency: RYW consistency is achieved when a database guarantees that, after a variable value is updated, all subsequent calls will read the updated value of the variable [159]. Smart contracts of Hyperledger Fabric blockchains can access the blockchain's state database, such as LevelDB per default and CouchDB as an alternative [160]. However, LevelDB and CouchDB do not offer RYW consistency [91], which can cause logic errors in smart contract codes that use data from the world state database.

To achieve RYW-like behavior, developers can make isolated calls to the database to read and write operations.

(C.28) Technical Soundness: *The hurdles developers are confronted with handling the technical capabilities and limitations of a smart contract's execution environment.*

Smart contract development is still a novel field in software development, and especially compilers for domain-specific languages (e.g., Solidity) are frequently updated. These updates fix bugs but can also change smart contract execution compared to older compiler versions [20].

(S.28.1) Fixed Compiler Version: To counter potential vulnerabilities caused by different compiler versions, developers should use fixed compiler versions [20].

4.3.4 Interoperability Challenges

Interoperability challenges related to the ease with which smart contracts can be called by other smart contracts or external systems (e.g., wallets) and can communicate with systems outside the distributed ledger.

(C.29) Smart Contract API Conformity: *The certainty with which developers can rely on the uniformity of smart contract interfaces that conform to published conventions and standards.*

Developers can develop code in their own style. However, if each developer defines smart contract functions differently, this can cause inconsistencies across smart contract definitions, can inform flaws in smart contract development (e.g., regarding function call definitions), and hinder cross-contract interoperability [147], [161].

(S.29.1) Ethereum Request for Comments: With an increasing number of smart contracts deployed by unknown entities with individual coding styles, the definition of standardized smart contract interfaces has become increasingly important, for example, to favor code reusability and ease interoperability with smart contracts. To agree on application-level standards and conventions, members of the Ethereum community can propose Ethereum Requests for Comments (ERCs). Smart contract codes can be published for discussion in an ERC. After members agree on a solution presented in an ERC, the ERC can become an Ethereum Improvement Proposal (EIP) that is discussed by the Ethereum community [162]. When the community agrees on the EIP, it can become an official standard, such as the ERC20 token standard or ERC26 and

ERC137 for name registries.

5 Related Work

Existing research has made valuable contributions to the understanding of the peculiarities of smart contracts and the resulting development challenges. These works can be associated with three research streams: *code analysis*, *software testing*, and *system design.* In works related to *code analysis*, selected flaws in smart contract code (e.g., overflow and underflow [20] or reentrancy [22]) and their detection using formal methods have been researched (e.g., [132], [163], [164]). For automated detection of these flaws, software tools have been proposed that perform formal verification (e.g., [163], [165], [166]), dynamic code analysis (e.g., [132], [133], [152]), static code analysis (e.g., [23], [24], [167]–[169]), or machine learning using classifiers like XGBoost or AdaBoost (e.g., [25]). These tools are designed to support developers in improving their code by identifying recurring flaws in smart contract code (e.g., by using formalized patterns of code flaws). Multiple works on code analysis have focused on performance optimization, especially to reduce the gas consumption of Ethereum smart contracts (e.g., [116], [117], [143]). For example, Chen et al. [117] presented patterns for gas-inefficient code (e.g., opaque predicates, dead code, and redundant `SSTORE`) and a software tool for the automated identification of these patterns in bytecode. Works associated with code analysis have also revealed various smart contract vulnerabilities (e.g., reentrancy or unchecked external calls) and methods for their identification (e.g., [14], [22], [132], [163]). Despite these endeavors, existing works on code analysis are highly technology-centric (e.g., by focusing only on Solidity or EVM [8], [20], [25], [115], [163]). Identified frequent flaws in smart contract code are distributed across various works, hindering developers from obtaining an overview of existing challenges. Moreover, the presented tools (e.g., [117], [131], [170]) are applicable only after a smart contract has been developed and are not intended to support developers in anticipating recurring development challenges before writing the code.

To support developers in incorporating new knowledge related to code analysis into their development routines and directly anticipating code flaws, this work describes challenges and corresponding solutions, including 20 SDPs. Moreover, this work extends the findings of foremost performance- and security-focused code analysis studies through knowledge about challenges related to maintainability and the implementation of certain functionalities (e.g., random number generation).

Besides code analysis, research has proposed approaches and tools for *software testing* (e.g., [24], [171]–[173]). Related works offer valuable and practical insights that support smart contract developers in improving their code through different testing strategies and tools. For example, Li et al. [170] proposed a software tool for mutation testing of Ethereum smart contract code to identify and fix flaws in their code. Gao et al. [171] presented an approach for automated testing of Ethereum smart contracts and suggested browser-side events that interact with smart contracts. These tools and respective insights can support development practices. Still, challenges occurring in smart contract development that could be useful for software testing to avoid frequent code flaws (e.g., guarding functions) remain unclear.

The challenges identified in this work can support better planning of software testing for smart contracts on blockchains based on Ethereum, EOSIO, and Hyperledger Fabric. For example, tests can be developed so that all challenges that apply to a specific DLT protocol are covered. Thereby, our work can support the targeted detection and elimination of frequent code flaws.

Works related to *system design* (e.g., [99], [139], [140], [174]) propose specific concepts or implementations to overcome recurring smart contract development challenges, such as random value generation with high entropy [96] and the integration of oracles [70]. For example, Li et al. [100] proposed the implementation of a lottery scheme focusing on random number generation. They applied a commitment pattern in which entities commit coordinates in a 2D coordinate system and compute a random number based on the polynomial that intersects with the coordinates. Still, these works sensitize developers to only a few challenges and hardly make developers aware of bad practices that should be avoided (e.g., using block numbers for random number generation).

This work presents particularities and challenges in smart contract development across DLT protocols. Developers can consider these peculiarities and challenges in two stages of the development process: first, when deciding to use a DLT protocol for developing DLT-based decentralized applications (DApps) that require specific smart contract capabilities (e.g., random number generation); second, when developing smart contracts on the chosen DLT protocol. In doing so, we complement previous work (e.g., [35], [175]) by assisting in the selection of suitable DLT protocols for individual DApps with a specific focus on smart contract integration and implementation.

Building on the described research streams (i.e., code analysis, software testing, and system design), several reviews and surveys on challenges in smart contract development have been published (e.g., [9], [10], [89], [100], [107], [148], [149], [170], [174], [176], [176], [177]). These publications present surveys regarding formal verification approaches (e.g., [176], [177]) and smart contract development challenges perceived by developers (e.g., [10], [149], [174]). Surveys on formal verification (e.g., [176], [177]) compare different approaches for code flaw detection regarding their capabilities and potentials for improvements. For example, Tolmach et al. [176] proposed formal models for smart contracts (i.e., contract-level models and program-level models) and surveyed smart contract specifications for different application domains (e.g., finance and social games). Miller et al. [170] scrutinized existing formal verification approaches regarding their capabilities to detect flaws in smart contract code and their applicability to different programming languages. Surveys on formal verification revealed valuable insights into recurring and automatically detectable

programming flaws, such as vulnerabilities for reentrancy attacks and mishandled exceptions.

Extant reviews of smart contract development challenges from the perspective of developers offer brief explanations of challenges, their corresponding solutions, and future research directions (e.g., [10], [101], [174], [178]). Several review studies have focused on specific challenge types, such as security-related challenges (e.g., [8], [19]) or challenges related to performance [117] or maintainability [101]. Only a few survey studies consider different challenge types to derive comprehensive guidance for software developers to handle these challenges by explaining corresponding solutions (e.g., [9], [10], [149]). Among these studies, Chen et al. [10] provided an extensive overview of challenges in Ethereum smart contract development derived from posts on the Ethereum StackExchange website, validated the existence of the identified challenges in a questionnaire with developers, and briefly described solutions to address the perceived challenges. Hu et al. [173] revealed development paradigms for application domains (e.g., Auction, Loan, and Lottery) applicable to DLT protocols with script-based and Turing-complete blockchains, such as Bitcoin and Ethereum. Moreover, the authors provided an overview of tool chains that can support developers in improving the quality of their code (e.g., through formal verification). Zou et al. [149] focused on the Ethereum blockchain and examined the differences between the development of traditional software and smart contracts and highlighted the particular challenges for the latter. They presented procedures like frequent code audits and code reviews to address the identified challenges and derive future research directions.

Our work advances prior reviews by collating different categories of smart contract challenges (e.g., [8], [9]) as well as corresponding solutions and transforming these solutions into detailed and actionable SDPs that align with recommendations in existing research (e.g., [31]–[33], [179]). By applying the canonical pattern structure proposed in prior research [31]–[33], our SDPs contain not only detailed descriptions of each solution, but also a discussion on benefits and boundary conditions. Thereby, the SDPs can help developers make better decisions for using SDPs and ultimately avoid common smart contract development mistakes. By iteratively discussing and refining our literature-based results with DLT experts, we provide empirical validation of our findings.

Finally, existing research focuses on overcoming the challenges of smart contracts developed in Solidity or executed in the EVM (e.g., [19], [20], [25], [163]). Only a few studies have explored smart contract challenges related to other DLT protocols (e.g., EOSIO [11], [180] or Hyperledger Fabric [91], [99]). Our study broadens this one-sided approach by considering three distinct DLT protocols with different smart contract integration concepts and thus shows which challenges and corresponding solutions in smart contract development apply for which DLT protocols considering their corresponding smart contract integration concepts. Thereby, our work can support developers throughout the software development lifecycle and deepen the understanding of how smart contract integration concepts can limit the flexibility of smart contracts (e.g., favoring deterministic execution by encapsulation).

6 Conclusions & Future Work

In this work, we present 29 smart contract development challenges and 60 corresponding solutions associated with 11 sub-themes, including data visibility and interoperability. The sub-themes relate to three principal challenge origins (i.e., *platform*, *programming language and execution environment*, and *coding practice*) that primarily cause the individual challenges. This classification enables a separate consideration of each principal challenge origin so that developers can better gauge between DLT protocols in combination with individual execution environments. This regard will become especially relevant for future DLT protocols that offer developers the option to choose between execution environments. For example, QTUM plans to integrate Neutron, a middleware that allows developers to use the EVM or an x86 virtual machine as desired [34]. Other DLT protocols (e.g., Ontology 2.0 [181]) also strive to offer multiple virtual machines in the future.

To make the generated knowledge handier for developers and adjust their programming habits, we offer 20 SDPs that can be used to address various challenges and augment identified solutions. We developed the 20 SDPs in cooperation with smart contract developers who are experts in Ethereum, EOSIO, and Hyperledger Fabric and refined the SDPs in multiple iterations considering quality criteria that we identified in a literature review.

Our results indicate that challenges in smart contract development are caused by individual characteristics of DLT protocols—foremost, the visibility of data to entities with access to the distributed ledger, the requirement for determinism, and the public executability of smart contract code (see Table 6). In Ethereum, several challenges relate to the difficult maintainability of smart contract code, which cannot be replaced but only redeployed and thereby assigned to a separate account. Regarding current endeavors in DLT protocol development, the updateability of smart contracts can cause novel challenges because entities may call for smart contract actions that execute unexpected logic. Where Hyperledger Fabric requires all entities that are relevant to endorse transactions after smart contract execution to agree on smart contract updates, updates of smart contract code (e.g., through replacement of the current contract version in EOSIO or by using the Proxy Pattern in Ethereum) in especially public DLT systems must be recognized by entities themselves prior to interacting with the contract. The ability to update code after deployment may decrease the trust of entities in the agreements manifested in contract code.

Using established programming languages such as C++ or Java can reduce the entry barrier for developers because they do not need to learn new programming languages. Nonetheless, our interviewees explained that using traditional general-purpose programming languages can

be misleading in smart contract development because developers may align with their usual programming habits, thus neglecting peculiarities in smart contract development (e.g., regarding the visibility of variables declared as private [20]).

Despite our efforts to answer our research questions regarding challenges and solutions in smart contract development, we cannot guarantee the comprehensiveness of our work. We focused on challenges related to current versions of execution environments, programming language, and compilers. Accordingly, we excluded challenges that apply to old versions, such as the Callstack Depth Attack [19] that was fixed in October 2016 and is now practically impossible, the use of unsafe type inferences in Solidity using `var` [20], and manipulating storage variables in Solidity that automatically point to register `0x0` when not initialized [182].

Performing two complementary literature reviews enabled us to identify various challenges and solutions. However, qualitative analysis techniques generally carry the risk of interpretation biases. Although we conducted multiple rounds of coding and refining themes during our thematic analysis to mitigate potential interpretation biases, researchers may come up with different theme conceptualizations. By reviewing the ever-increasing number of grey literature (e.g., DLT foundations' whitepapers) and examining practitioners' discussions on smart contract development (e.g., developer blog and forum entries), future research may analyze the usefulness of the solutions presented in this work and refine the contexts to which the solutions apply. In doing so, future research can ultimately deepen our knowledge of common solutions for overcoming smart contract development challenges.

Our study has limitations concerning the number and depth of interviews we conducted to gather data on challenges and solutions in smart contract development and to improve our SDPs. While we conducted various interviews with DLT and smart contract experts, the interviewees may have found it difficult to verbalize some challenges of smart contract development, and future research might gather more information on specific findings to increase understanding. The limited number and depth of interviews, as well as the fact that we could not consider all SDP quality criteria presented in Appendix B, likely left opportunities for improving the developed SDPs. To improve the SDPs presented in this work, the SDPs should be evaluated in a longitudinal large-scale study considering all quality criteria presented in Appendix B. This way, methodological limitations of this work can be addressed, and the effectiveness of the presented SDPs in overcoming smart contract development challenges will be improved.

We provide a set of solutions, including SDPs, based on related work and interview findings. Given the large number and diversity of solutions, we selected a subset of available solutions as the base for developing SDPs. Developers and researchers may come up with additional solutions and SDPs that can even improve those presented in this work. Moreover, we have discussed the applicability of the identified challenges and solutions with DLT experts for Ethereum, EOSIO, and Hyperledger Fabric. Thus, it remains unclear which challenges and solutions apply to other DLT protocols or whether solutions applied to smart contracts for other DLT protocols may improve the SDPs presented in this work. To improve the presented SDPs and understand their applicability to other DLT protocols, future studies should investigate challenges and solutions in smart contract development for other DLT protocols. In this way, DLT protocol-agnostic SDPs can be uncovered, revealing key best practices for smart contract development.

To advance our solutions to smart contract development challenges, we maintain a public repository, including complete descriptions of all patterns only briefly described in this work. We have planned to add further patterns for blockchains based on Ethereum, EOSIO, and Hyperledger Fabric to constantly support developers in their work.

Acknowledgments

This work was carried out in the project scope "Toward Better Smart Contract Development" and was funded by the EnBW Energie Baden-Württemberg AG in Germany. Moreover, this work was supported by KASTEL Security Research Labs. We thank all the participants in the empirical studies that contributed to this work. Moreover, we thank N. Hasebrook and M. Pfister for their support in analyzing the results from the focus group workshops and refining the SDPs, as well as J. Bartsch, M. Beyene, and F. Morsbach for their valuable input during the preparation of this paper. Especially, we thank A. Kaiser and C. Michelbach from Blockinfinity, P. Mesnier from Object Computing, F. Gerbig and G. Cyriac from the BMW Group, A. Sorniotti from IBM, and B. Sturm for their continuous support in all stages of the development of the manuscript. The corresponding author is N. Kannengießer.

References

[1] N. Szabo, "Formalizing and Securing Relationships on Public Networks," *First Monday*, vol. 2, no. 9, Sep. 1997, doi: 10.5210/fm.v2i9.548.

[2] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, and R. Zunino, "SoK: Unraveling Bitcoin Smart Contracts," in *Principles of Security and Trust*, Cham, 2018, pp. 217–242.

[3] Bitcoin community, "Script," *Bitcoin Wiki*, Jun. 16, 2019. https://en.bitcoin.it/wiki/Script (accessed Nov. 19, 2019).

[4] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure Multiparty Computations on Bitcoin," in *2014 IEEE Symposium on Security and Privacy*, San Jose, CA, May 2014, pp. 443–458. doi: 10.1109/SP.2014.35.

[5] C. S. Wright, "A Proof of Turing Completeness in Bitcoin Script," in *Intelligent Systems and Applications*, vol. 1037, Y. Bi, R. Bhatia, and S. Kapoor, Eds. Cham: Springer International Publishing, 2020, pp. 299–313. doi: 10.1007/978-3-030-29516-5_23.

[6] X. Wang, J. He, Z. Xie, G. Zhao, and S. C. Cheung, "ContractGuard: Defend Ethereum Smart Contracts with Embedded Intrusion Detection," *IEEE Trans. Serv. Comput.*, pp. 1–14, 2019, doi: 10.1109/TSC.2019.2949561.

[7] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart Contract Security: A

Software Lifecycle Perspective," *IEEE Access*, vol. 7, pp. 150184–150202, 2019, doi: 10.1109/ACCESS.2019.2946988.

[8] M. Wöhrer and U. Zdun, "Smart contracts: security patterns in the ethereum ecosystem and solidity," in *2018 International Workshop on Blockchain Oriented Software Engineering*, Mar. 2018, pp. 2–8.

[9] M. Wöhrer and U. Zdun, "Design Patterns for Smart Contracts in the Ethereum Ecosystem," in *2018 IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*, Halifax, NS, Canada, Jul. 2018, pp. 1513–1520. doi: 10.1109/Cybermatics_2018.2018.00255.

[10] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining Smart Contract Defects on Ethereum," *IEEE Trans. Software Eng.*, pp. 1–17, 2020, doi: 10.1109/TSE.2020.2989002.

[11] Y. Huang *et al.*, "Characterizing EOSIO Blockchain," *arXiv:2002.05369 [cs]*, Feb. 2020, Accessed: Mar. 09, 2021. [Online]. Available: http://arxiv.org/abs/2002.05369

[12] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts," presented at the Network and Distributed Systems Security Symposium 2018, San Diego, CA, USA, 2018. doi: http://dx.doi.org/10.14722/ndss.2018.23082.

[13] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," *arXiv:1802.06038 [cs]*, Feb. 2018, Accessed: Aug. 24, 2019. [Online]. Available: http://arxiv.org/abs/1802.06038

[14] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, Oct. 2018, doi: 10.1145/3276486.

[15] S. Palladiono, "The Parity Wallet Hack Reloaded," *OpenZeppelin*, Nov. 07, 2017. https://blog.openzeppelin.com/parity-wallet-hack-reloaded/ (accessed Aug. 27, 2019).

[16] X. Zhao, Z. Chen, X. Chen, Y. Wang, and C. Tang, "The DAO attack paradoxes in propositional logic," in *4th International Conference on Systems and Informatics*, Nov. 2017, pp. 1743–1746.

[17] M. Young, "A Billion EOS Tokens Faked to Rob Decentralized Exchange," *Ethereum World News*, Sep. 18, 2018. https://ethereumworldnews.com/a-billion-eos-tokens-faked-to-rob-decentralized-exchange/ (accessed Mar. 01, 2019).

[18] D. Enyeart, "Resolve phantom reads for range queries," *The Linux Foundation*, Jul. 20, 2018. https://jira.hyperledger.org/browse/FAB-1668 (accessed Jan. 10, 2021).

[19] A. Mense and M. Flatscher, "Security Vulnerabilities in Ethereum Smart Contracts," in *20th International Conference on Information Integration and Web-based Applications & Services*, Yogyakarta, Indonesia, 2018, pp. 375–380. doi: 10.1145/3282373.3282419.

[20] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: static analysis of ethereum smart contracts," in *1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, Gothenburg, Sweden, 2018, pp. 9–16. doi: 10.1145/3194113.3194115.

[21] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Principles of Security and Trust*, vol. 10204, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. doi: 10.1007/978-3-662-54455-6_8.

[22] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard: finding reentrancy bugs in smart contracts," in *40th International Conference on Software Engineering: Companion Proceeedings*, Gothenburg Sweden, May 2018, pp. 65–68. doi: 10.1145/3183440.3183495.

[23] Y. Chinen, N. Yanai, J. P. Cruz, and S. Okamura, "Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis," *arXiv:2007.01029 [cs]*, Jul. 2020, Accessed: Feb. 10, 2021. [Online]. Available: http://arxiv.org/abs/2007.01029

[24] S. Akca, A. Rajan, and C. Peng, "SolAnalyser: A Framework for Analysing and Testing Smart Contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Putrajaya, Malaysia, Dec. 2019, pp. 482–489. doi: 10.1109/APSEC48747.2019.00071.

[25] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts," *IEEE Trans. Netw. Sci. Eng.*, pp. 1–1, 2020, doi: 10.1109/TNSE.2020.2968505.

[26] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber, "A Pattern Collection for Blockchain-based Applications," in *23rd European Conference on Pattern Languages of Programs*, Irsee Germany, Jul. 2018, pp. 1–20. doi: 10.1145/3282308.3282312.

[27] ConsenSys, "Smart Contract Security Best Practices." Jan. 26, 2021. Accessed: Mar. 15, 2021. [Online]. Available: https://github.com/ConsenSys/smart-contract-best-practices/

[28] R. Xie, "Best Practices to Level Up Your Ethereum Smart Contracts," *Hacker Noon*, Oct. 09, 2018. https://hackernoon.com/best-practices-to-level-up-your-ethereum-smart-contracts-944d5cea2cab (accessed Oct. 10, 2019).

[29] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic Literature Reviews in Software Engineering," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.

[30] V. Braun and V. Clarke, "Using thematic analysis in psychology," *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77–101, Jan. 2006, doi: 10.1191/1478088706qp063oa.

[31] F. Buschmann, Ed., *Pattern-oriented software architecture: a system of patterns*. Chichester ; New York: Wiley, 1996.

[32] J. Borchers and F. Buschmann, *A Pattern Approach to Interaction Design*. USA: John Wiley & Sons, Inc., 2001.

[33] B. Appleton, "Patterns and Software: Essential Concepts and Terminology." 2000. Accessed: Aug. 02, 2019. [Online]. Available: http://www.bradapp.net/docs/patterns-intro.pdf

[34] Qtum, "Neutron: Middleware for Blockchain Virtual Machines," *Medium*, Apr. 28, 2020. https://blog.qtum.org/neutron-middleware-for-blockchain-virtual-machines-fe267353dfb2 (accessed Oct. 27, 2020).

[35] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev, "Trade-Offs between Distributed Ledger Technology Characteristics," *ACM CSUR*, vol. 53, no. 2, Apr. 2020, doi: 10.1145/3379463.

[36] D. Chaum, "Blind Signatures for Untraceable Payments," in *Advances in Cryptology*, Boston, MA, 1983, pp. 199–203. doi: 10.1007/978-1-4757-0602-4_18.

[37] I. Sergey and A. Hobor, "A Concurrent Perspective on Smart Contracts," in *Financial Cryptography and Data Security*, vol. 10323, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. A. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Cham: Springer International Publishing, 2017, pp. 478–493. doi: 10.1007/978-3-319-70278-0_30.

[38] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *TOPLAS*, vol. 4, no. 3, pp. 382–401, 1982.

[39] E.-E. Gojka, N. Kannengießer, B. Sturm, J. Bartsch, and A. Sunyaev, "Security in Distributed Ledger Technology: An Analysis of Vulnerabilities and Attack Vectors," in *Intelligent Computing*, vol. 285, K. Arai, Ed. Cham: Springer International Publishing, 2021, pp. 722–742. doi: 10.1007/978-3-030-80129-8_50.

[40] V. Buterin, "Ethereum Whitepaper," *Ethereum*, Feb. 09, 2021. https://ethereum.org/en/whitepaper/ (accessed Mar. 06, 2021).

[41] V. Buterin, "EIP-214: New opcode STATICCALL," *Ethereum Improvement Proposals*, Feb. 13, 2017. https://eips.ethereum.org/EIPS/eip-7 (accessed

Oct. 27, 2020).

[42] EOSIO, "Consensus Protocol," Nov. 24, 2020. https://github.com/EOSIO/welcome/blob/master/docs/60_protocol-guides/10_consensus_protocol.md#3-eosio-consensus-dpos--abft (accessed Feb. 03, 2021).

[43] EOSIO, "EOS VM - A Low-Latency, High Performance and Extensible WebAssembly Engine," Nov. 08, 2019. https://github.com/EOSIO/eos-vm (accessed Dec. 29, 2020).

[44] EOSIO, "RAM as system resource," Nov. 24, 2020. https://github.com/EOSIO/eosio.contracts/blob/master/docs/01_key-concepts/02_ram.md (accessed Feb. 03, 2021).

[45] EOSIO, "Storage and Read Modes," Sep. 01, 2020. https://github.com/EOSIO/eos/blob/master/docs/01_nodeos/07_concepts/05_storage-and-read-modes.md (accessed Feb. 08, 2021).

[46] EOSIO, "Accounts and Permissions," *GitHub*, Nov. 24, 2020. https://github.com/EOSIO/welcome/blob/master/docs/60_protocol-guides/40_accounts_and_permissions.md (accessed Feb. 03, 2021).

[47] EOSIO, "Transactions Protocol," Nov. 24, 2020. https://github.com/EOSIO/welcome/blob/master/docs/60_protocol-guides/20_transactions_protocol.md (accessed Feb. 03, 2021).

[48] Blockgenic, "EOSIO Smart Contracts Tutorial." Apr. 29, 2018. Accessed: Feb. 08, 2021. [Online]. Available: https://medium.com/coinmonks/eosio-smart-contracts-tutorial-f22c3bb364d9

[49] EOSIO, "Inline Actions to External Contracts," Dec. 01, 2020. https://github.com/EOSIO/welcome/blob/master/docs/40_smart-contract-guides/70_inline-action-to-external-contract.md (accessed Feb. 08, 2021).

[50] Hyperledger Foundation, "The Ordering Service," *GitHub*, Nov. 06, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/orderer/ordering_service.md (accessed Dec. 30, 2020).

[51] Hyperledger Foundation, "Channels," Sep. 29, 2018. https://hyperledger-fabric.readthedocs.io/en/release-2.2/channels.html (accessed Feb. 02, 2020).

[52] Hyperledger Architecture Working Group, "Fabric chaincode lifecycle," *GitHub*, Sep. 02, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/chaincode_lifecycle.md (accessed Jan. 27, 2021).

[53] Hyperledger, "Smart Contracts and Chaincode," *GitHub*, Aug. 18, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/smartcontract/smartcontract.md (accessed Dec. 30, 2020).

[54] Hyperledger Foundation, "Transaction Flow," *GitHub*, Jun. 26, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/txflow.rst (accessed Jan. 10, 2021).

[55] Hyperledger Foundation, "A New Approach," *GitHub*, Feb. 11, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/whatis.md#a-new-approach (accessed Jan. 10, 2021).

[56] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*. Reading, Mass: Addison-Wesley, 1995.

[57] D. Riehle and H. Züllighoven, "Understanding and Using Patterns in Software Development," *TAPOS*, vol. 2, no. 1, pp. 3–13, 1996, doi: https://doi.org/10.1002/(SICI)1096-9942(1996)2:1<3::AID-TAPO1>3.0.CO;2-%23.

[58] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *18th International Conference on Evaluation and Assessment in Software Engineering*, London, England, United Kingdom, 2014, pp. 1–10. doi: 10.1145/2601248.2601268.

[59] M. D. Myers, *Qualitative research in business & management*, 2nd ed. SAGE Publications Ltd, 2013.

[60] G. Paré, M. C. Trudel, M. Jaana, and S. Kitsiou, "Synthesizing information systems knowledge: A typology of literature reviews," *Information and Management*, vol. 52, no. 2, pp. 183–199, 2015.

[61] M. Q. Patton, *Qualitative research & evaluation methods: integrating theory and practice*, Fourth edition. Thousand Oaks, CA, USA, 2015.

[62] R. K. Yin, *Case study research: design and methods*, 4th ed. Los Angeles, CA, USA, 2009.

[63] J. M. Corbin and A. L. Strauss, *Basics of qualitative research: techniques and procedures for developing grounded theory*, 4th ed. Los Angeles, CA, USA, 2015.

[64] G. Guest, A. Bunce, and L. Johnson, "How Many Interviews Are Enough?: An Experiment with Data Saturation and Variability," *Field Methods*, vol. 18, no. 1, pp. 59–82, Feb. 2006, doi: 10.1177/1525822X05279903.

[65] J. M. Morse, "'Data Were Saturated . . . ,'" *Qual Health Res*, vol. 25, no. 5, pp. 587–588, May 2015, doi: 10.1177/1049732315576699.

[66] P. Fusch and L. Ness, "Are We There Yet? Data Saturation in Qualitative Research," *Qualitative Report*, vol. 20, pp. 1408–1416, 2015.

[67] A. L. Strauss and J. M. Corbin, Eds., *Grounded theory in practice*. Thousand Oaks: Sage Publications, 1997.

[68] B. Saunders *et al.*, "Saturation in qualitative research: exploring its conceptualization and operationalization," *Qual Quant*, vol. 52, no. 4, pp. 1893–1907, Jul. 2018, doi: 10.1007/s11135-017-0574-8.

[69] S. McConnell, *Code complete*, 2nd ed. Redmond, Wash: Microsoft Press, 2004.

[70] J. Heiss, J. Eberhardt, and S. Tai, "From oracles to Trustworthy Data On-chaining Systems," presented at the 2019 IEEE International Conference on Blockchain, Atlanta, GA, USA, 2019. doi: 10.1109/Blockchain.2019.00075.

[71] J. Eberhardt and J. Heiss, "Off-chaining Models and Approaches to Off-chain Computations," in *2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, Rennes, France, 2018, pp. 7–12. doi: 10.1145/3284764.3284766.

[72] A. Ampatzoglou and A. Chatzigeorgiou, "Evaluation of object-oriented design patterns in game development," *Information and Software Technology*, vol. 49, no. 5, pp. 445–454, May 2007, doi: 10.1016/j.infsof.2006.07.003.

[73] D. Khazanchi, J. D. Murphy, and S. C. Petter, "Guidelines for Evaluating Patterns in the IS Domain," in *2nd Midwest United States Association for Information Systems Conference*, 2008, vol. 24. [Online]. Available: https://digitalcommons.unomaha.edu/isqafacproc/7

[74] D. Lea, "Christopher Alexander: an introduction for object-oriented designers," *SIGSOFT Softw. Eng. Notes*, vol. 19, no. 1, pp. 39–46, Jan. 1994, doi: 10.1145/181610.181617.

[75] S. Niebuhr, K. Kohler, and C. Graf, "Engaging Patterns: Challenges and Means Shown by an Example," in *Engineering Interactive Systems*, vol. 4940, J. Gulliksen, M. B. Harning, P. Palanque, G. C. van der Veer, and J. Wesson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 586–600. doi: 10.1007/978-3-540-92698-6_35.

[76] M. J. Brotherson, "Interactive Focus Group Interviewing: A Qualitative Research Method in Early Intervention," *Topics in Early Childhood Special Education*, vol. 14, no. 1, pp. 101–118, Jan. 1994.

[77] R. L. Gorden, *Interviewing: strategy, techniques, and tactics*, Rev. ed. Homewood, Ill: Dorsey Press, 1975.

[78] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "Keccak implementation overview." May 29, 2012. Accessed: Jul. 15, 2019. [Online]. Available: https://keccak.team/files/Keccak-implementation-3.2.pdf

[79] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart

Contracts," in *2016 IEEE Symposium on Security and Privacy*, May 2016, pp. 839–858.

[80] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *2016 ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2016, pp. 270–282.

[81] S. Eskandari, S. Moosavi, and J. Clark, "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain," in *Financial Cryptography and Data Security*, vol. 11599, A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, and M. Sala, Eds. Cham: Springer International Publishing, 2020, pp. 170–189. doi: 10.1007/978-3-030-43725-1_13.

[82] ConsenSys, "Known Attacks," *GitHub*, Jan. 26, 2021. https://github.com/ConsenSys/smart-contract-best-practices/blob/master/docs/known_attacks.md (accessed May 21, 2020).

[83] Hyperledger Foundation, "Private data," *GitHub*, Jan. 29, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/private-data/private-data.md (accessed Feb. 02, 2021).

[84] Hyperledger Foundation, "Private Data," *GitHub*, May 23, 2020. https://hyperledger-fabric.readthedocs.io/en/release-2.2/private-data-arch.html?highlight=private%20data (accessed Dec. 17, 2021).

[85] Hyperledger Foundation, "Membership Service Providers (MSP)," *GitHub*, Jun. 25, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/msp.rst (accessed Jan. 05, 2021).

[86] M. Lücking, F. Kretzer, N. Kannengießer, M. Beigl, A. Sunyaev, and W. Stork, "When Data Fly: An Open Data Trading System in Vehicular Ad-Hoc Networks," *Electronics*, vol. 1, no. 5, Mar. 2021, doi: https://doi.org/10.3390/electronics1010005.

[87] Web of Trust Info, "did:erc725 method," *GitHub*, Feb. 21, 2018. https://github.com/WebOfTrustInfo/rwot6-santabarbara/blob/master/topics-and-advance-readings/DID-Method-erc725.md (accessed Oct. 21, 2020).

[88] M. Luecking, C. Fries, R. Lamberti, and W. Stork, "Decentralized Identity and Trust Management Framework for Internet of Things," in *2020 IEEE International Conference on Blockchain and Cryptocurrency*, Toronto, ON, Canada, 6.05 2020, pp. 1–9. doi: 10.1109/ICBC48266.2020.9169411.

[89] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 1–29, Oct. 2019, doi: 10.1145/3360615.

[90] Hyperledger, "Peers," *GitHub*, Sep. 19, 2019. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/peers/peers.md (accessed Feb. 17, 2021).

[91] K. Yamashita, Y. Nomura, E. Zhou, B. Pi, and S. Jun, "Potential Risks of Hyperledger Fabric Smart Contracts," in *2019 IEEE International Workshop on Blockchain Oriented Software Engineering*, Hangzhou, China, Feb. 2019, pp. 1–10. doi: 10.1109/IWBOSE.2019.8666486.

[92] J. Bonneau, J. Clark, and S. Goldfeder, "On Bitcoin as a public randomness source." 2015. Accessed: Sep. 17, 2018. [Online]. Available: https://eprint.iacr.org/2015/1015.pdf

[93] C. Pierrot and B. Wesolowski, "Malleability of the blockchain's entropy," *Cryptogr. Commun.*, vol. 10, no. 1, pp. 211–233, Jan. 2018, doi: 10.1007/s12095-017-0264-3.

[94] C. Cachin, K. Kursawe, and V. Shoup, "Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography," *J Cryptology*, vol. 18, no. 3, pp. 219–246, Jul. 2005, doi: 10.1007/s00145-005-0318-0.

[95] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theoretical Computer Science*, vol. 777, pp. 155–183, Jul. 2019, doi: 10.1016/j.tcs.2019.02.001.

[96] P. Schindler, A. Judmayer, N. Stifter, and E. Weippl, "HydRand: Practical Continuous Distributed Randomness." Jul. 30, 2019. Accessed: Nov. 17, 2020. [Online]. Available: https://eprint.iacr.org/2018/319

[97] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani, "Probabilistic Smart Contracts: Secure Randomness on the Blockchain," in *2019 IEEE International Conference on Blockchain and Cryptocurrency*, Seoul, Korea (South), May 2019, pp. 403–412. doi: 10.1109/BLOC.2019.8751326.

[98] randao, "Random number in programming is very important!" Nov. 28, 2019. Accessed: Jan. 03, 2021. [Online]. Available: https://github.com/randao/randao/blob/master/README.md

[99] J. Li, Z. Zhang, and M. Li, "BanFEL: A Blockchain Based Smart Contract for Fair and Efficient Lottery Scheme," in *2019 IEEE Conference on Dependable and Secure Computing*, Hangzhou, China, Nov. 2019, pp. 1–8. doi: 10.1109/DSC47296.2019.8937559.

[100] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016, pp. 254–269. doi: 10.1145/2976749.2978309.

[101] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintaining Smart Contracts on Ethereum: Issues, Techniques, and Future Challenges," *arXiv:2007.00286 [cs]*, Jul. 2020, Accessed: Mar. 12, 2021. [Online]. Available: http://arxiv.org/abs/2007.00286

[102] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach," *arXiv:1711.09327 [cs]*, Nov. 2017, Accessed: Aug. 24, 2019. [Online]. Available: http://arxiv.org/abs/1711.09327

[103] Hyperledger Foundation, "Glossary," *GitHub*, Mar. 03, 2021. https://github.com/hyperledger/fabric/blob/master/docs/source/glossary.rst (accessed Feb. 08, 2021).

[104] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Beijing, China, 2019, pp. 363–373. doi: 10.1145/3293882.3330560.

[105] ISO/IEC JTC 1/SC 7 Software and systems engineering, "ISO/IEC 14764:2006 Software Engineering — Software Life Cycle Processes — Maintenance." Sep. 2006. Accessed: May 11, 2019. [Online]. Available: https://www.iso.org/standard/39064.html

[106] M. Bartoletti and L. Pompianu, "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns," in *Financial Cryptography and Data Security*, 2017, pp. 494–509.

[107] C. R. Worley and A. Skjellum, "Opportunities, Challenges, and Future Extensions for Smart-Contract Design Patterns," in *Business Information Systems Workshops*, 2019, pp. 264–276. doi: https://doi.org/10.1007/978-3-030-04849-5_24.

[108] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, "Applying Design Patterns in Smart Contracts," in *Blockchain – ICBC 2018*, vol. 10974, S. Chen, H. Wang, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 92–106. doi: 10.1007/978-3-319-94478-4_7.

[109] M. Araoz, "Proxy Libraries in Solidity," *Medium*, Mar. 06, 2017. https://medium.com/zeppelin-blog/proxy-libraries-in-solidity-79fbe4b970fd (accessed May 18, 2018).

[110] Parity Technologies, "A Postmortem on the Parity Multi-Sig Library Self-Destruct," *parity*, Nov. 15, 2017. https://www.parity.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/ (accessed Jul. 22, 2019).

[111] Ethereum Foundation, "Expressions and Control Structures," *GitHub*, Jun. 26, 2019. https://github.com/ethereum/solidity/blob/v0.5.12/docs/control-structures.rst (accessed Dec. 03, 2020).

[112] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart Contract: Attacks and Protections," *IEEE Access*, vol. 8, pp. 24416–24427, 2020, doi:

10.1109/ACCESS.2020.2970495.

[113] Y. Jeng, Y. Hsieh, and J.-L. Wu, "Step-by-Step Guidelines for Making Smart Contract Smarter," in *2019 IEEE 12th Conference on Service-Oriented Computing and Applications*, Kaohsiung, Taiwan, Nov. 2019, pp. 25–32. doi: 10.1109/SOCA.2019.00012.

[114] Ethereum, "Introduction to Smart Contracts," *GitHub*, Dec. 20, 2020. https://github.com/ethereum/solidity/blob/v0.8.2/docs/introduction-to-smart-contracts.rst (accessed Jan. 18, 2021).

[115] M. Kaleem, A. Mavridou, and A. Laszka, "Vyper: A Security Comparison with Solidity Based on Common Vulnerabilities," in *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services*, Paris, France, Sep. 2020, pp. 107–111. doi: 10.1109/BRAINS49436.2020.9223278.

[116] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering*, London, ON, Canada, Feb. 2020, pp. 9–15. doi: 10.1109/IWBOSE50093.2020.9050163.

[117] T. Chen *et al.*, "GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts," *IEEE Trans. Emerg. Topics Comput.*, pp. 1–1, 2020, doi: 10.1109/TETC.2020.2979019.

[118] C. Ferreira Torres, J. Schütte, and R. State, "Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts," presented at the Annual Computer Security Applications Conference, San Juan, PR, USA, Dec. 2018. doi: https://doi.org/10.1145/3274694.3274737.

[119] M. Wöhrer and U. Zdun, "Domain Specific Language for Smart Contract Development," in *2020 IEEE International Conference on Blockchain and Cryptocurrency*, Toronto, ON, Canada, May 2020, pp. 1–9. doi: 10.1109/ICBC48266.2020.9169399.

[120] A. C. Cañada, "Fixed point math in Solidity," *Medium*, Apr. 22, 2019. https://medium.com/cementdao/fixed-point-math-in-solidity-616f4508c6e8 (accessed Jan. 31, 2020).

[121] vyperlang, "Vyper," *GitHub*, Jul. 02, 2020. https://github.com/vyperlang/vyper/blob/5db35ef4eb07650eb57f769deba9d3dc22b646af/docs/index.rst (accessed Oct. 27, 2020).

[122] Anton Efremov, "Concurrency patterns for Hyperledger Fabric Go chaincode," *SAP Community*, Feb. 10, 2020. https://blogs.sap.com/2020/02/10/concurrency-patterns-for-hyperledger-fabric-go-chaincode/ (accessed Jun. 19, 2021).

[123] unknown, "Package sync," *GoLang*. https://golang.org/pkg/sync/ (accessed Jul. 12, 2020).

[124] Aniruddha, "Iterating over maps in Go," *Medium*, Jul. 27, 2019. https://medium.com/i0exception/map-iteration-in-go-275abb76f721 (accessed Jan. 09, 2021).

[125] D. Perez and B. Livshits, "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited," *arXiv:1902.06710 [cs]*, Oct. 2020, Accessed: Mar. 16, 2021. [Online]. Available: http://arxiv.org/abs/1902.06710

[126] Ethereum, "Contract ABI Specification," *Solidity v.0.6.10*, Jun. 09, 2020. https://solidity.readthedocs.io/en/v0.6.10/abi-spec.html (accessed Jul. 02, 2020).

[127] Hyperledger, "Transaction handlers," *GitHub*, Jun. 27, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/developapps/transactionhandler.md (accessed Nov. 20, 2020).

[128] Ethereum, "Solidity Assembly," *GitHub*, Oct. 11, 2019. https://github.com/ethereum/solidity/blob/v0.5.13/docs/assembly.rst (accessed Jan. 03, 2021).

[129] EOSIO, "eosio.token.cpp," *GitHub*, May 09, 2018. https://github.com/EOSIO/eos/blob/dawn-v4.0.0/contracts/eosio.token/eosio.token.cpp#L76 (accessed Feb. 20, 2021).

[130] S. Marx, "Stop Using Solidity's transfer() Now," *ConsenSys*, Sep. 02, 2019. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/ (accessed Apr. 01, 2020).

[131] K. Bhargavan *et al.*, "Formal Verification of Smart Contracts: Short Paper," in *2016 ACM Workshop on Programming Languages and Analysis for Security*, 2016, pp. 91–96. doi: 10.1145/2993600.2993611.

[132] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks," *arXiv:1812.05934 [cs]*, Dec. 2018, Accessed: Aug. 24, 2019. [Online]. Available: http://arxiv.org/abs/1812.05934

[133] S. Grossman *et al.*, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 1–28, Jan. 2018, doi: 10.1145/3158136.

[134] vyperlang, "Structure of a Contract," *GitHub*, Aug. 04, 2019. https://github.com/vyperlang/vyper/blob/v0.1.0-beta.12/docs/structure-of-a-contract.rst (accessed Jan. 04, 2021).

[135] L. Desrosiers and R. Olivieri, "Oracles: Common architectural patterns for Hyperledger Fabric," *IBM Developer*, Mar. 11, 2019. https://developer.ibm.com/technologies/blockchain/articles/oracles-common-architectural-patterns-for-fabric (accessed Jan. 11, 2021).

[136] X. Xu, I. Weber, and M. Staples, "Blockchain Patterns," in *Architecture for Blockchain Applications*, Cham: Springer International Publishing, 2019, pp. 113–148. doi: 10.1007/978-3-030-03035-3_7.

[137] EOS Authority, "EOSIO Watcher Plugin by EOS Authority," *GitHub*, Dec. 10, 2018. https://github.com/eosauthority/eosio-watcher-plugin (accessed Jan. 02, 2021).

[138] EOSIO, "How EOS get data from outside world," *Github*, Feb. 27, 2018. https://github.com/EOSIO/eos/issues/1483 (accessed Feb. 13, 2021).

[139] DOS Network, "DOS Network - A Decentralized Oracle Service boosting blockchain usability with off-chain data & verifiable computing power." May 29, 2020. Accessed: Jun. 02, 2020. [Online]. Available: https://dosnetwork.github.io/docs/#/homepage

[140] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania, "Astraea: A Decentralized Blockchain Oracle," in *2018 IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*, Halifax, NS, Canada, Jul. 2018, pp. 1145–1152. doi: 10.1109/Cybermatics_2018.2018.00207.

[141] Joseph Chow, "A Guide to Events and Logs in Ethereum Smart Contracts," *ConsenSys Blog*, Jul. 06, 2016. https://consensys.net/blog/blockchain-development/guide-to-events-and-logs-in-ethereum-smart-contracts/ (accessed Jun. 17, 2019).

[142] Ethereum Foundation, "Solidity Documentation - Types." Jan. 14, 2019. Accessed: Jul. 09, 2020. [Online]. Available: https://github.com/ethereum/solidity/blob/v0.5.12/docs/types.rst

[143] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, Klagenfurt, Austria, Feb. 2017, pp. 442–446. doi: 10.1109/SANER.2017.7884650.

[144] T. Chen *et al.*, "Towards Saving Money in Using Smart Contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results*, May 2018, pp. 81–84.

[145] R. Hitches, "Getting Loopy with Solidity," *Medium*, Oct. 16, 2018. https://blog.b9lab.com/getting-loopy-with-solidity-1d51794622ad (accessed Sep. 23, 2019).

[146] J. Goddard, "Shadowing Solidity Storage Variables in Memory," *Medium*, Jun. 20, 2020. https://medium.com/coinmonks/shadowing-solidity-storage-variables-in-memory-b56f471edd81 (accessed Mar. 09, 2021).

[147] A. Sunyaev *et al.*, "Token Economy," *Bus Inf Syst Eng*, vol. 63, pp. 457–478,

Feb. 2021, doi: 10.1007/s12599-021-00684-1.

[148] A. Vitanov, "Solidity Smart Contracts Design Patterns," *Medium*, Feb. 24, 2018. https://medium.com/@i6mi6/solidty-smart-contracts-design-patterns-ecfa3b1e9784 (accessed Feb. 12, 2020).

[149] W. Zou *et al.*, "Smart Contract Development: Challenges and Opportunities," *IEEE Trans. Software Eng.*, pp. 1–20, 2019, doi: 10.1109/TSE.2019.2942301.

[150] Ethereum, "Style Guide," *GitHub*, Jun. 23, 2020. https://github.com/ethereum/solidity/blob/v0.7.0/docs/style-guide.rst (accessed Nov. 27, 2020).

[151] Hyperledger, "Coding guidelines," *GitHub*, Apr. 11, 2020. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/smartcontract/smartcontract.md (accessed Nov. 03, 2020).

[152] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "EASYFLOW: Keep Ethereum Away from Overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion - Companion*, Montreal, QC, Canada, 31.05 2019, pp. 23–26. doi: 10.1109/ICSE-Companion.2019.00029.

[153] OpenZeppelin, "SafeMath." Aug. 17, 2019. Accessed: Aug. 23, 2019. [Online]. Available: https://github.com/OpenZeppelin/ openzeppelin-solidity/blob/master/contracts/math/SafeMath.sol

[154] Ethereum, "Solidity v0.8.0 Breaking Changes," *GitHub*, Dec. 16, 2020. https://github.com/ethereum/solidity/blob/v0.8.0/docs/080-breaking-changes.rst (accessed Nov. 27, 2020).

[155] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, "Detecting Ponzi Schemes on Ethereum: Towards Healthier Blockchain Technology," in *2018 World Wide Web Conference on World Wide Web*, Republic and Canton of Geneva, Switzerland, 2018, pp. 1409–1418. doi: 10.1145/3178876.3186046.

[156] G. Konstantopoulos, "How to Secure Your Smart Contracts: 6 Solidity Vulnerabilities and how to avoid them (Part 2)," *Medium*, Jan. 17, 2018. https://medium.com/loom-network/how-to-secure-your-smart-contracts-6-solidity-vulnerabilities-and-how-to-avoid-them-part-2-730db0aa4834 (accessed Oct. 28, 2019).

[157] A. Hertig, "Rise of Replay Attacks Intensifies Ethereum Divide," *coinbase*, Jul. 29, 2016. https://www.coindesk.com/rise-replay-attacks-ethereum-divide (accessed Feb. 23, 2020).

[158] L. Quan, L. Wu, and H. Wang, "EVulHunter: Detecting Fake Transfer Vulnerabilities for EOSIO's Smart Contracts at Webassembly-level," *arXiv:1906.10362 [cs]*, Jun. 2019, Accessed: Mar. 17, 2021. [Online]. Available: http://arxiv.org/abs/1906.10362

[159] A. S. Tanenbaum and M. van Steen, *Distributed systems: principles and paradigms*, 2nd ed. Upper Saddle River, NJ, USA: Maarten van Steen, 2016.

[160] Hyperledger, "Using CouchDB," *GitHub*, Sep. 19, 2019. https://github.com/hyperledger/fabric/blob/release-2.2/docs/source/couchdb_tutorial.rst (accessed Feb. 17, 2021).

[161] N. Kannengießer, M. Pfister, M. Greulich, S. Lins, and A. Sunyaev, "Bridges Between Islands: Cross-Chain Technology for Distributed Ledger Technology," presented at the Hawaii International Conference on System Sciences (HICSS), Maui, Hawaii, USA, 2020.

[162] Ethereum Foundation, "EIP-1," *GitHub*, Mar. 06, 2021. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1.md (accessed Mar. 19, 2021).

[163] P. Antonino and A. W. Roscoe, "Formalising and verifying smart contracts with Solidifier: a bounded model checker for Solidity," *arXiv:2002.02710 [cs]*, Feb. 2020, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/2002.02710

[164] X. Bai, Z. Cheng, Z. Duan, and K. Hu, "Formal Modeling and Verification of Smart Contracts," in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, Kuantan Malaysia, Feb. 2018, pp. 322–326. doi: 10.1145/3185089.3185138.

[165] T. Abdellatif and K. L. Brousmiche, "Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security*, Feb. 2018, pp. 1–5. doi: 10.1109/NTMS.2018.8328737.

[166] S. Rezaei, E. Khamespanah, M. Sirjani, A. Sedaghatbaf, and S. Mohammadi, "Developing Safe Smart Contracts," in *IEEE 44th Annual Computers, Software, and Applications Conference*, Madrid, Spain, Jul. 2020, pp. 1027–1035. doi: 10.1109/COMPSAC48688.2020.0-137.

[167] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, Montreal, QC, Canada, May 2019, pp. 8–15. doi: 10.1109/WETSEB.2019.00008.

[168] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts," *arXiv:2005.06227 [cs]*, May 2020, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/2005.06227

[169] Y. Huang, Q. Kong, N. Jia, X. Chen, and Z. Zheng, "Recommending Differentiated Code to Support Smart Contract Update," in *2019 IEEE/ACM 27th International Conference on Program Comprehension*, Montreal, QC, Canada, May 2019, pp. 260–270. doi: 10.1109/ICPC.2019.00045.

[170] A. Miller, Z. Cai, and S. Jha, "Smart Contracts and Opportunities for Formal Methods," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, vol. 11247, T. Margaria and B. Steffen, Eds. Cham: Springer International Publishing, 2018, pp. 280–299. doi: 10.1007/978-3-030-03427-6_22.

[171] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "MuSC: A Tool for Mutation Testing of Ethereum Smart Contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, San Diego, CA, USA, Nov. 2019, pp. 1198–1201. doi: 10.1109/ASE.2019.00136.

[172] J. Gao *et al.*, "Towards Automated Testing of Blockchain-based Decentralized Applications," in *27th International Conference on Program Comprehension*, Piscataway, NJ, USA, 2019, pp. 294–299. doi: 10.1109/ICPC.2019.00048.

[173] E. Andesta, F. Faghih, and M. Fooladgar, "Testing Smart Contracts Gets Smarter," *arXiv:1912.04780 [cs]*, Dec. 2019, Accessed: Mar. 19, 2021. [Online]. Available: http://arxiv.org/abs/1912.04780

[174] B. Hu *et al.*, "A comprehensive survey on smart contract construction and execution: paradigms, tools, and systems," *Patterns*, vol. 2, no. 2, p. 100179, Feb. 2021, doi: 10.1016/j.patter.2020.100179.

[175] M. E. Peck, "Blockchain world - Do you need a blockchain? This chart will tell you if the technology can solve your problem.," *IEEE Spectrum*, vol. 54, no. 10, pp. 38–60, Oct. 2017.

[176] V. Dwivedi, V. Deval, A. Dixit, and A. Norta, "Formal-Verification of Smart-Contract Languages: A Survey," in *Advances in Computing and Data Sciences*, Singapore, 2019, pp. 738–747.

[177] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A Survey of Smart Contract Formal Specification and Verification," *arXiv:2008.02712 [cs]*, Apr. 2021, Accessed: Jun. 19, 2021. [Online]. Available: http://arxiv.org/abs/2008.02712

[178] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, "Security Smells in Smart Contracts," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion*, Sofia, Bulgaria, Jul. 2019, pp. 442–449. doi: 10.1109/QRS-C.2019.00086.

[179] C. Alexander, "The origins of pattern theory: the future of the theory, and the generation of a living world," *IEEE Softw.*, vol. 16, no. 5, pp. 71–82, Oct.

1999, doi: 10.1109/52.795104.

[180] N. He *et al.*, "Security Analysis of EOSIO Smart Contracts." Jul. 30, 2020. Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/2003.06568

[181] The Ontology Team, "Multi-VM in Ontology 2.0: the First to Support Seamless Contract Interactions Among Three Environments," *Medium*, May 12, 2020. https://medium.com/ontologynetwork/multi-vm-in-ontology-2-0-f76178022ad4

[182] R. Hitchens, "Storage Pointers in Solidity," *Medium*, 16.112018. https://blog.b9lab.com/storage-pointers-in-solidity-7dcfaa536089 (accessed Mar. 17, 2011).