Computer Science Review 45 (2022) 100492

Contents lists available at ScienceDirect

Computer Science Review

journal homepage: www.elsevier.com/locate/cosrev

Review article

Blockchain verification and validation: Techniques, challenges, and research directions

Dusica Marijan [a,*], Chhagan Lal [b]

[a] Simula Research Laboratory, Oslo, Norway
[b] Delft University of Technology, Netherlands

ARTICLE INFO

Article history:
Received 17 August 2021
Received in revised form 2 June 2022
Accepted 12 July 2022
Available online 28 July 2022

Keywords:
Blockchain
Smart contracts
P2P
Consensus
Ledger
Testing
Verification
Validation
Simulation
Benchmarking
Software testing
Security testing
Performance testing
System under test
Formal verification
Platform testing

ABSTRACT

As blockchain technology is gaining popularity in industry and society, solutions for Verification and Validation (V&V) of blockchain-based software applications (BC-Apps) have started gaining equal attention. To ensure that BC-Apps are properly developed before deployment, it is paramount to apply systematic V&V to verify their functional and non-functional requirements. While existing research aims at addressing the challenges of engineering BC-Apps by providing testing techniques and tools, blockchain-based software development is still an emerging research discipline, and therefore, best practices and tools for the V&V of BC-Apps are not yet sufficiently developed. In this paper, we provide a comprehensive survey on V&V solutions for BC-Apps. Specifically, using a layered approach, we synthesize V&V tools and techniques addressing different components at various layers of the BC-App stack, as well as across the whole stack. Next, we provide a discussion on the challenges associated with BC-App V&V, and summarize a set of future research directions based on the challenges and gaps identified in existing research work. Our study aims to highlight the importance of BC-App V&V and pave the way for a disciplined, testable, and verifiable BC development.

Contents

* Corresponding author.
E-mail addresses: dusica@simula.no (D. Marijan), c.lal@tudelft.nl (C. Lal).

https://doi.org/10.1016/j.cosrev.2022.100492
1574-0137/

1. Introduction

In recent years, blockchain (BC) technology has gained increasing attention across many domains, ranging from manufacturing and healthcare to insurance and aeronautics [1]. It is mainly because of BC's inherent features, such as decentralization, immutability, improved security, transparency, and its ability to securely implement complex business logic and processes through Smart Contracts (SCs). On the one hand, these features provide various advantages to businesses [2–4], while on the other hand, they increase the complexity of verifying the correctness of applications using these features [5]. Therefore, with an increasing demand to leverage BC technology for various purposes in different applications, it is equally important to perform adequate V&V to detect bugs and vulnerabilities in BC components and interfaces that could lead to security threats or asset losses [6].

BC-Apps significantly differ from traditional software applications (i.e., applications not using BC technology). Specifically, BC-Apps have specific requirements and acceptance criteria to be reached in testing [5,7]. For instance, deploying correct SCs is highly important in BC, because their execution cannot be reversed once implemented. Due to BC's immutable nature, once a buggy SC goes into a production system, fixing the bug might require a complete revision of the code. Besides, the SC code also defines how efficiently the software performs with increasing workloads. Therefore, it becomes necessary to perform comprehensive performance testing of SCs before deployment, to ensure their expected performance in production. Moreover, SCs are mostly written in new programming languages (e.g., Solidity [8], and Go [9]), for which either the testing tools are not available or are not at mature stages. While SC programming is challenging [10], SC correctness is paramount, because bugs in SCs may lead to immense asset losses and disruptions [11]. To support the development of secure SCs, numerous tools and techniques have emerged in recent years [6,12]. Some of these tools may help the analysis of already deployed SCs [13].

Apart from SCs, the decentralized and anonymous nature of the participating nodes that work together with distributed systems in a peer-to-peer network further adds to the BC V&V complexity. For instance, the distributed nature enforces the need for the validation of synchronization between the nodes, and testing the performance and security of consensus algorithms [14]. Due to software testing challenges inherent to BC-Apps, V&V techniques used for traditional software applications may not be valid for testing BC-Apps, thus calling for specialized BC V&V tools and practices.

Since SCs are important entities in a BC implementation, a large percentage of the state-of-the-art is addressing the area of ensuring the correctness of SCs. Researchers have proposed different approaches, including formal verification methods (e.g., model checking and theorem proving) [12,22], fuzzing methods (e.g., mutation, and hybrid) [23,24], automated program repair [25], symbolic execution and analysis [26], and Control Flow Graph (CFG) construction [27]. Other than SC testing, researchers have worked on approaches for BC performance testing, aimed at evaluating the performance of BC-Apps under different workloads and faultloads [28,29]. Apart from SC testing and BC performance testing, other areas, such as peer/node testing, Application Programming Interface (API) testing, and consensus algorithm testing have not been sufficiently explored by the research community yet.

Due to the fragmented nature of the available solutions for BC-App V&V, it is of utmost importance to the research community to (i) identify specific challenges in BC V&V, key BC components that need to be tested, and the ongoing research efforts in this direction, along with their limitations, and (ii) propose novel V&V techniques to address the identified open challenges.

1.1. State-of-the-art and contribution

There are several survey articles available in the state-of-the-art on V&V-related efforts for BC technology. However, these articles cover only the partial aspects of BC V&V, for example, SC testing [6,13,15,16] or BC performance testing [17,20,21] or security testing [18,19]. To the best of our knowledge, there is no previous study providing a comprehensive survey of different techniques for ensuring the functional and non-functional correctness of BC components at different layers and across the whole BC stack. Moreover, existing surveys mainly cover verification approaches for individual BC components and do not study the verification of BC-Apps, where BC components are integrated with various real-world applications. Finally, existing surveys do not investigate new testing challenges that BC software developers and testers face while developing and testing BC-Apps. Understanding these challenges is an essential step towards progressing the current state-of-the-art in BC V&V. Table 1 provides a comparison of our survey with the state-of-the-art by considering several parameters. The comparison clearly shows the need for our study in the domain of BC V&V.

The key contributions of our paper are as follows.

- We present a detailed survey of existing techniques and tools proposed for the V&V of BC-Apps. We identify the limitations of existing approaches, and provide a comparison

Table 1
Comparison with related work.

	SC testing	Performance testing	Security testing	API/interface testing	BC V&V challenge identification
[15], 2018	Yes	No	No	No	Partially
[13], 2019	Yes	No	No	No	No
[16], 2019	Yes	No	No	No	No
[6], 2020	Yes	No	No	No	No
[17], 2020	No	Yes	No	No	No
[18], 2020	Partially	No	Yes	No	Partially
[19], 2020	Partially	No	Yes	No	Partially
[20], 2021	No	Yes	No	No	No
[21], 2021	No	Yes	No	No	No
This survey	Yes	Yes	Yes	Yes	Yes

between different approaches by discussing their efficiency, performance, and practicability.
- Using a layered design for BC-Apps, we discuss possible approaches for identifying the key components at each layer, and the interfaces across different layers, that need to be verified, along with the best techniques for that purpose.
- We provide a comprehensive discussion of the challenges faced by software developers while developing and verifying BC-Apps, in contrast to traditional (non-BC-based software). Finally, we provide a set of research directions for advancing the state-of-the-art on BC-App V&V.

The remainder of this paper is organized as follows. We provide the required background that includes a brief discussion about BC and SC technology, as well as the overview of V&V techniques used in software engineering in Section 2. We provide the taxonomy of BC V&V in Section 3. In Sections 4–7, we describe the state-of-the-art V&V tools and techniques for performing different types of testing and verification on various BC components. In Section 8, we summarize the findings from our comprehensive state-of-the-art review. We discuss open issues and the directions of future research work in the area of BC V&V in Section 9. Finally, we conclude our work in Section 10.

2. Background

In this section, we present a discussion on BC and SC, BC components that need to be tested, BC-Apps, and software V&V techniques relevant for the understanding of the paper.

2.1. Blockchain and smart contracts

BC is a distributed ledger consisting of a series of chronologically ordered blocks appended in a link-list type of data-structure. To provide integrity and immutability of data in the ledger, it is required to prevent any updates in the committed blocks. To ensure this, each block contains the hash of the previous block, and the ledger is replicated across peers in the BC network. A block usually contains a set of timestamped transactions that are bundled together and stored in the form of a Merkle tree [30]. BC adopts various cryptographic primitives like hashing algorithms, digital signatures, and Public Key Infrastructure (PKI) protocols to ensure adequate security. There are two key participants in the BC network, one that generates transactions and the other that validates and stores them in the ledger.

A BC network runs on a peer-to-peer topology where each node is expected to store the same copy of the ledger. Nodes or organizations may not have a preexisting trust relationship among them. Therefore, to ensure that each peer node has the same copy of the ledger at any given time, a new valid block that will be appended to the ledger is selected by executing a consensus mechanism. In particular, a consensus mechanism, e.g., Practical Byzantine Fault Tolerance (PBFT), Proof-of-Work (PoW), and Proof-of-Stake (PoS), is a protocol that ensures synchronization among all network peers about the validity and ordering of transactions [31]. Therefore, these mechanisms are pivotal for BC's correct functioning and need to be tested properly before their use in real-world applications.

Depending on the type of access, BC can be *public*, *private*, *consortium*, or *hybrid*. A public BC is permissionless, allowing anyone to access the ledger, interact with other participants, and create and validate new data blocks. In a public BC, all transactions are visible to all participants. Examples of public BCs are Ethereum, Bitcoin, and Litecoin. On the other hand, a private BC is permissioned, allowing access to only those who are granted the right of access by the central authority [32]. Examples of private BCs are Hyperledger (HL) and Ripple. While a public BC has longer validation times for new data compared to a private BC, a private BC is more vulnerable to malicious actors. Consortium and hybrid BCs stand in the middle. A consortium BC is permissioned and governed by a group of organizations, thus having higher levels of decentralization and security compared to a private BC [33]. A hybrid BC is managed by a single organization, and uses both private and public BCs. For example, a public BC can be used to verify the data stored in between the ledgers of a private BC. An example of a hybrid BC is XinFin, built on Ethereum as a public BC and Quorum as a private BC. Depending upon the requirements of the target application, one should select a suitable type of BC platform for the deployment of BC-Apps [34].

BC can use SCs, which are securely stored on the BC and are executed manually (via a transaction invoking a function in it) or automatically (when a precondition evaluates to true). Specifically, SCs allow for decentralized automation by facilitating the verification and enforcement of conditions written in the underlying contract [35]. In this way, SCs serve as policies that supervise a transaction. For instance, an SC can define a set of rules for an individual's travel insurance, which trigger's the contract execution when a traveling carrier (such as a flight or train) is experiencing a delay by more than a fixed amount of time. In particular, an SC consists of a set of instructions or operations written in special programming languages (e.g., Solidity and Go Lang), and it gets executed upon the fulfillment of predefined conditions. The key property that makes SCs of great use in many real-world applications is their ability to eliminate the requirement of a trusted third party in multiparty interactions [36]. Parties can participate by performing secure peer-to-peer transactions over BC without placing their trust in outside parties that are generally used to ensure that all parties fulfill the contractual obligations. Currently, the largest BC platform for SC deployment is Ethereum [35]. It uses Solidity, a high-level scripting language that is specifically designed for writing SCs [8].

Besides using SCs to eliminate the requirement of a trusted third party in multiparty interactions, there are other benefits that SCs provide. These include data fusion [37], consent management, fine-grained access control, and reduced bureaucracy

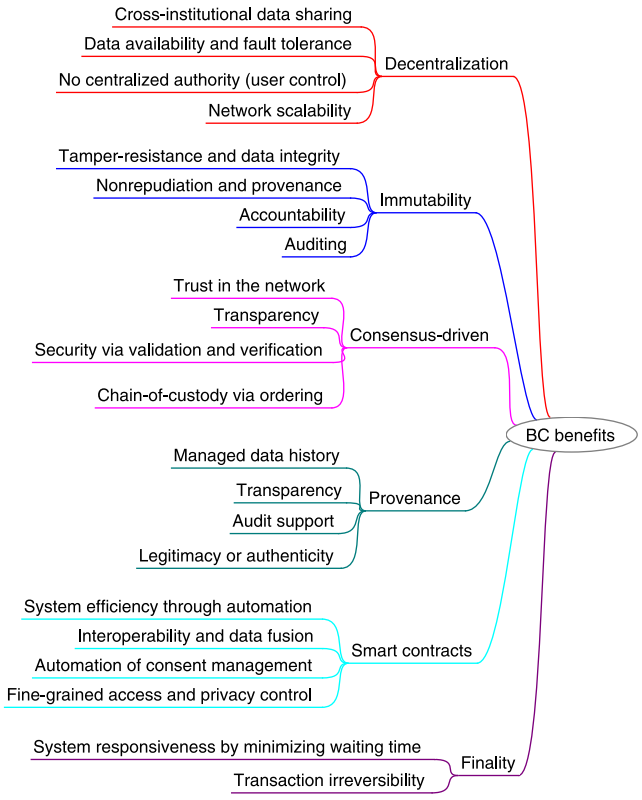

Fig. 1. Key BC features.

and expenses. Moreover, BC with its features such as tamper-resistance, decentralization, and transparency, provides a much-needed platform for secure deployment and execution of SCs. Therefore, in recent years, there has been a rapid increase in the popularity of SCs [38]. However, similar to other software programs, SCs may contain bugs leading to vulnerabilities. Since BC technology along with SCs is mainly used in either financial applications (e.g., banking, insurance, and trade of goods or services) or data-sensitive applications (e.g., Healthcare and smart-grids [39]), these bugs can be exploited by malicious entities for financial gains or leaking sensitive data. Thus, it is vital to perform rigorous testing to ensure the development of bug-free SCs. Further, unlike traditional software, it is difficult to update an SC once it is deployed. Therefore, it is critical to verify SCs before their deployment to avoid serious adverse consequences.

Fig. 1 shows the key features provided by BC platforms used in various applications. These features include decentralization, immutability, consensus (e.g., chain-of-custody via ordering [40]), provenance, SCs, and finality [41]. These inherent features are the reason behind the rapid increase in the usage of BC and SC in various domains. However, as these technologies are new and immature, there is a need for rigorous testing solutions to ensure their correctness for use in any real-world application.

2.2. BC components

To better understand the interaction interfaces and components that require testing in BC-Apps, we provide a generic framework of BC-Apps in Fig. 2. The figure shows different BC components within a BC-App that need to be tested to ensure the BC-App security and overall performance. At the Application Layer of BC-Apps, there are typically APIs enabling the integration with different data sources, and APIs for accessing and interacting with BC components. These APIs need thorough testing for

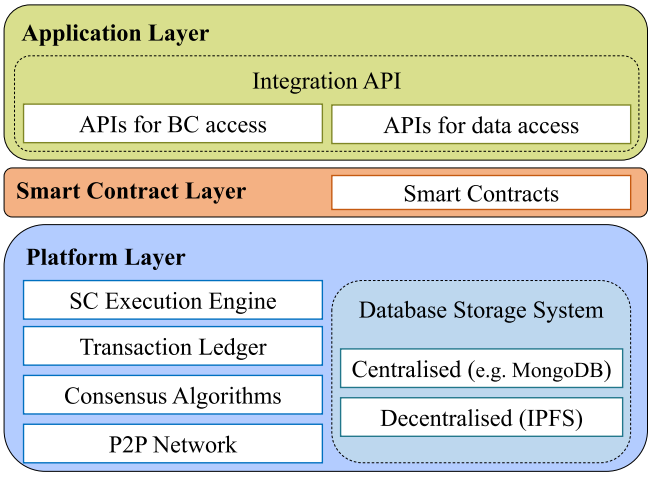

Fig. 2. Generic reference architecture for BC-Apps showing BC components that require testing.

ensuring their correct functionality. Specifically, the APIs used to access data storage systems should be tested for security (i.e., data protection from malicious or unauthorized entities), and performance (i.e., low-latency to data access operations).

At the Smart Contract Layer, there are SCs that require thorough testing to detect different types of vulnerabilities and functional bugs. SC testing may use different testing types and methods, and it may be conducted for different purposes. For instance, functional testing of SCs needs to be done to identify bugs and to check the implemented business logic correctness. Security testing of SCs needs to detect vulnerabilities that may be exploited by a malicious entity after SC deployment. Performance testing of SCs could be done to check their execution and time complexity, leading to code optimizations.

Furthermore, at the Platform Layer, BC-Apps include a set of core BC components such as Peer-to-Peer (P2P) network, consensus protocols, transaction ledger, and storage systems (including both on-chain and off-chain), which should be thoroughly tested using different types of testing techniques. Since a large number of BC-Apps are data-driven applications, i.e., data acts as an asset, in domains where data volume is huge (e.g., medical data sharing and managing a digital forensics evidence), the off-chain data storage solutions are preferred due to performance and compliance reasons. Therefore, in such BC-Apps, to ensure security (i.e., data protection) and good performance (i.e., low access latency), testing of data storage solutions and their interactions with other BC components and with application users should be performed.

We further detail different types of testing required for BC components at and across different layers of BC-Apps in Fig. 4 and Section 3.

2.3. Blockchain-based applications

In this paper, by *BC-App* we mean a software application that uses BC technology. In contrast, by a *traditional software application* we mean an application not using the BC technology.

While BC-App benefits from several features provided by BC, deploying BC-Apps can have specific challenges, such as low scalability, high energy consumption, integration problems, privacy, and security issues. In particular, the BC technology used for an application does not work in isolation. It must interact with various components of the application infrastructure in which it is being deployed. Therefore, before deploying a BC-App, we must

perform systematic testing and verification, to ensure a secure and efficient interaction between the BC components and the application entities.

In some business applications using BC, BC components are considered as an add-on technology in existing business processes. This requires verifying all integration points between the BC and the application. However, there has been a shift towards the inclusion of BC from the start of BC-Apps development, rather than developing them in isolation followed by the integration with BC. Therefore, we can expect the emergence of multiple APIs to facilitate the development of BC-Apps. Consequently, it is important to understand and test such APIs, to ensure proper integration.

2.4. Software verification and validation

Software V&V is a process performed during software engineering, aimed to check whether a software system meets its requirements, thus fulfilling its intended purpose. While verification aims at checking whether the software is functioning correctly, validation aims at checking whether the software satisfies customer's needs. One of the most commonly used techniques for verifying that a software works correctly is *software testing*. In general, the primary objective of software testing is detecting software errors (aka bugs).

The practice of software testing can be classified along different dimensions. According to where it happens in the software development life-cycle, there are four major levels recognized: *unit*, *integration*, *system*, and *acceptance* testing. Unit testing aims to ensure that individual software code units are working correctly. Integration testing is performed to evaluate whether multiple units are working correctly when integrated. System testing analyzes the whole system's behavior, evaluating whether it is compliant with its specified requirements. Finally, acceptance testing tests whether the software works for its users. Both the complexity and cost of testing increase while moving from unit testing to acceptance testing. This is because unit tests are more easily automated compared to other types of tests.

According to whether the software is executed or not during analysis, testing can be *static* or *dynamic*. Static testing (aka static analysis) does not involve software execution, but performs the checking of source code structure, syntax, and data-flow. Typical approaches to static analysis include inspections, reviews, and walkthroughs. Static analysis can be used with *formal verification* methods for proving the correctness of software. Formal verification normally requires a formal specification of software, which is used to provide formal proof of the software's correctness. The most used techniques to carry out formal verification are *theorem proving* [42] and *model checking* (also called property checking) [43]. Theorem proving requires well-known axioms and basic inference rules which are used to derive every new theorem or lemma that is needed for the proof. Since it applies to all systems that can be expressed mathematically, theorem proving is considered a flexible verification method. Moreover, it can be interactive, automated, or a hybrid of the two. On the other hand, model checking uses specific software to verify if a system's finite-state model works as per its formal specification and correctness properties. Model-checking software first takes input from a user, including the finite state model of the System Under Test (SUT) and the set of formally specified properties that it should have. Second, it checks if all the states satisfy the specifications. Moreover, authors in studies like [44], and [45] show the usage of formal verification methods for the correctness verification of BC consensus algorithms. Dynamic testing, on the other hand, involves software execution, while feeding inputs and producing outputs. Test cases are typically developed by specifying test inputs and outputs, and the goal of testing is to check whether the actual outputs conform to the expected outputs.

Finally, according to the testing objective, we distinguish *functional* and *non-functional* testing. Functional software testing aims to evaluate whether the software is compliant with specified software functional requirements. Functional testing techniques can be further classified as *white-box* testing and *black-box* testing. White-box testing aims to verify the internal structure of the software and is usually performed at the level of unit testing. Specific white-box testing techniques include API testing and mutation testing with fault injection. Black-box testing aims to examine the functionality of the software without having any knowledge about the software's internal working. It answers the question of "what" the software does, not "how" it does it. Black-box testing is usually performed at the level of integration testing, system testing and user acceptance testing. Typical black-box testing techniques include model-based testing, fuzz testing, pair-wise testing, boundary value analysis, equivalence partitioning, and exploratory testing. Non-functional testing aims to examine non-functional requirements of the software, for example, how does the software perform under unforeseen (at design time) circumstances, or how does the software recover from failures. Some types of non-functional testing techniques include *security* testing, *performance* testing, or *usability* testing. Security testing aims at detecting threats, vulnerabilities and risks within the software, to prevent attacks from intruders. Performance testing assesses how the software responds in conditions of a given workload. Typical types of performance testing include *load* testing, performed to assess the response of software under a given load, *stress* testing, performed to assess the response of software under upper limits of capacity, and *endurance* testing, performed to assess software response under continuous load. Usability testing checks how easily the software can be used by its users, with the overall goal to improve user experience. For a detailed description of the software testing field (both traditional software and software integrating machine learning), we point interested readers to other research work, such as [46–48].

3. Taxonomy of blockchain verification and validation

We propose a taxonomy of BC-App V&V, broadly classified into *SC Testing*, *Platform Testing*, and *Application Testing*, illustrated in Fig. 3. We further discuss different testing types and methods belonging to these three main classes in Sections 4–7.

3.1. Smart contract testing

SC Testing aims to discover bugs and vulnerabilities in SC code. If SC source code is analyzed in a non-runtime environment, such analysis type is called *static analysis*, while if SCs are checked at run-time, it is called *dynamic analysis* (verification). Orthogonal with the testing type (static vs. dynamic), SC Testing can utilize varying testing methods, such as *formal methods*, *machine learning*, *search-based*, or *model-based* methods. Furthermore, SC testing may aim to achieve *functional compliance* or *non-functional compliance*. Testing for functional compliance addresses the functional correctness of SCs with respect to requirements specification. Testing for non-functional compliance focuses on detecting issues such as buffer overflows, run-time safety, command injection, cross-site scripting, and security vulnerabilities present in SCs.

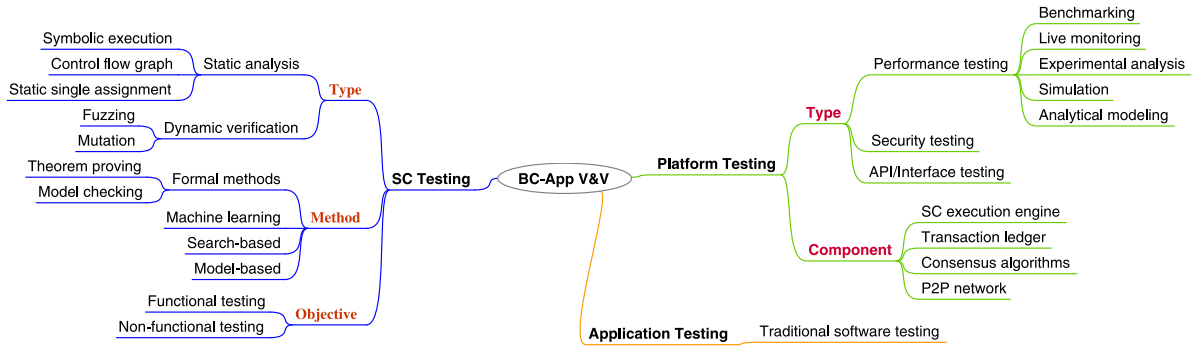

Fig. 3. Blockchain verification and validation taxonomy.

3.2. Platform testing

The end-to-end Quality of Service (QoS) of BC-Apps is highly affected by the capabilities of the underlying BC platform. Testing these platform capabilities can be broadly classified into *Performance Testing*, *Security Testing*, and *API & Interface Testing*. Performance Testing focuses on benchmarking BC platforms on various parameters such as network latency, transaction processing speed (including performance bottlenecks in a production environment), transaction sequence at each node, and responses needed from SCs. Security Testing focuses on the security vulnerabilities in BC platforms, including network security, cross-site scripting, access control to the BC platform. API & Interface Testing focuses on API security, discovery and performance testing. These testing types are orthogonal with the platform *Components*, which include SC execution engine, transaction ledger, consensus algorithms, P2P network and communication protocols.

Furthermore, Platform Testing includes the testing of data storage system illustrated in the BC-Apps architecture in Fig. 2, through the testing of some of the Platform Layer components. In particular, BC-Apps could have on-chain or off-chain storage systems. The security of on-chain data depends on the proper functioning and security of other BC components, such as transaction ledger, P2P network, and consensus algorithms, as these platform components ensure properties such as data integrity, data availability, and data provenance. While the testing of off-chain storage systems is done using traditional database storage systems testing tools/methods.

3.3. Application testing

Applications developed on top of a BC platform can be considered as traditional software applications. Herein, we do not aim to review testing techniques for traditional software applications (applications not using BC technology), since there already exist many such studies [49–52].

3.4. Layer, inter-layer, and cross-layer testing of BC-Apps

Fig. 4 illustrates the abstraction layers in a BC-App architecture showing relevant BC components at each layer and the types of testing at and across the layers. For the sake of clarity, we further break down the *Platform Layer* into *Data Layer*, *Consensus Layer*, and *Network Layer*.

Performance and security testing are typically performed across all the layers of a BC-App. Furthermore, unit testing is performed at each layer, testing individual BC components such as consensus algorithms or communication protocols. In the case of SCs, we call this type of testing SC Testing. Finally, API and Interface testing are performed during the integration of components belonging to different layers.

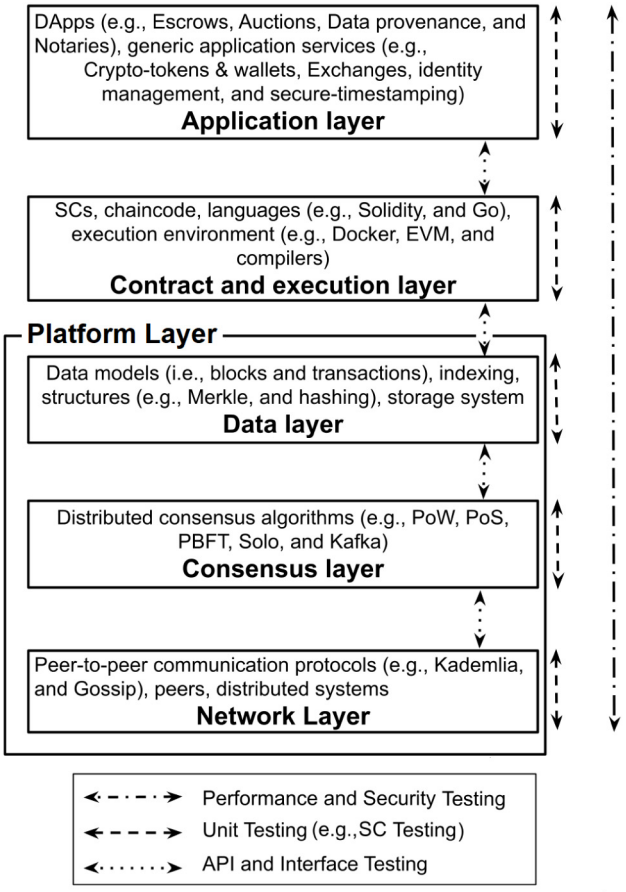

Fig. 4. Layered architecture for BC-Apps, illustrating the types of testing at and across the layers.

4. State of the art: Smart contract testing

The presence of bugs and vulnerabilities (e.g., buffer overflows, command injection, and cross-site scripting) in SCs after their deployment on a BC network causes the following two major problems: (i) wastage of resources (e.g., gas and computing hardware) and delay in transaction processing, due to the presence of bugs in SCs, and (ii) security threats caused by malicious entities by exploiting the vulnerabilities present in SCs. Both these problems may lead to financial and non-financial (e.g., sensitive

data) asset losses in BC-Apps. Moreover, such issues decrease the users' trust in BC, which could further harm the reputation of the organization using the BC-Apps. Therefore, it is important to provide tools and techniques to discover and fix SC bugs and vulnerabilities before deployment.

4.1. Static analysis

SC testing techniques based on static analysis analyze SC source code structure, syntax, and data-flow without source code execution. Next, we discuss popular SC static analysis tools (see Tables 2 and 3).

Oyente [27] is the first symbolic execution tool for Ethereum SCs that automatically detects popular vulnerabilities like transaction order dependency and reentrancy. Oyente directly works with Ethereum Virtual Machine (EVM) bytecode without needing to access the high-level representations (e.g., Solidity, Serpent). The tool was evaluated on 19 366 SCs extracted from the first 1,460,000 blocks in the Ethereum network, and it detected 8833 SCs potentially having documented bugs. However, Oyente only aims to detect potentially vulnerable contracts, leaving the full-scale false positive detection as future work. **Osiris** [53] is another tool based on Oyente, which focuses on detecting integer-related bugs, such as integer overflows. Moreover, the tool combines taint analysis with symbolic execution to reduce the number of false positives. Similarly, authors in [54] propose **Mythril**, a tool for security analysis of Ethereum SCs by using concolic and taint analysis and control-flow checking.

ZEUS [55] is an automated SC verification framework that uses abstract interpretation and model checking, and accepts user-provided policies. First, it inserts policy predicates as assert statements in the SC code, then it translates the code into an intermediate Low-Level Virtual Machine (LLVM) representation, and finally, it verifies ascertain assertion violations. Another tool called **Securify** [56] uses static analysis to extract semantic information about SC bytecode, using a dependency graph of the SC, and then checks safety pattern violations. The tool allows adding new patterns created using a designated Domain-Specific Language (DSL), thus aiding flexibility. **SmartCheck** [57] is a static analysis tool that first changes Solidity code into an XML-based intermediate representation, which is then examined against XPath patterns to find potential security, operative, operational, and development bugs. Other tools, **Gasper** [58] and **GASTAP** [59], use static analysis to identify potential code optimizations in SCs both at high-level and a bytecode instruction level, while mainly focusing on dead code and loop optimizations. Other popular static analysis tools are **Vandal** [60] and **EtherTrust** [61].

Slither [62] is a static analysis tool for automated vulnerability detection and code optimization detection for SCs. It deals with real-world contracts and aims to provide rich information, speed, robustness, and an adequate tradeoff between detection rate and false positives. It also supports users to improve their understanding of SC code by assisting them with code review, and by flagging bad coding practices. Moreover, it uses program analysis techniques such as dataflow and taint tracking to extract and refine the information, and it performs lexical and syntactical analysis on Solidity code. Authors in [63] propose **NPChecker**, which uses static analysis along with code instrumentation for detecting non-deterministic payment bugs in SCs. The novelty of the tool lies in the support to search bugs beyond the predefined vulnerability patterns.

Ethainter [64] is a security analyzer for SCs. It checks information flow with data sanitization in SCs to identify compound attacks that include an intensification of tainted information through many transactions, leading to extreme violations. The Ethainter implementation is highly scalable, shown by its application to the entire set of unique SCs around 38M Line of Code (LoC) on the Ethereum BC in 6 h. The evaluation results show that using automatic exploit generation (e.g., killing about 800 SCs on the Ropsten network) and manual review, Ethainter obtains high accuracy of 82.5% real warnings for end-to-end vulnerabilities. Moreover, Ethainter provides an adequate tradeoff between code coverage and false positives. Finally, the authors in [65] propose **SolidiFI**, an automated tool that uses a systematic approach to evaluate existing static analysis tools to detect SC vulnerabilities. First, SolidiFI allows bug injection (i.e., code defects) at different SC locations to introduce predefined security vulnerabilities. Second, it analyzes the resulting buggy SCs and collects the bugs that remain undetected (along with false positives) by the analyzed tools. SolidiFI evaluates six static analysis tools: SmartCheck, Manticore [26], Oyente, Securify, Mythril, and Slither, using 50 SCs injected with 9369 distinct bugs. The evaluation results show that several bug instances remain undetected by these tools despite their claims of detecting such bugs. Moreover, all the tools list several false positives.

VERX [74] is the first automatic verifier which can prove the temporal safety properties of Ethereum SCs. VERX's design consists of combining the following three ideas: (i) reachability checking via reduction of temporal safety verification, (ii) calculation of particular symbolic states within a transaction by using an efficient symbolic execution engine, and (iii) delayed abstraction, which approximates symbolic states into abstract states at the end of transactions. Based on the evaluation results obtained by considering 12 real-world Ethereum projects, it is determined that VERX automatically proves 83 temporal safety properties of SCs, demonstrating its practicality. Moreover, the experimental results conclude that VERX is a practical framework for testing custom functional properties of SCs. However, it is challenging to scale the verification to a large number of SCs. In particular, while particular specifications of SCs are mandatory to prove its customized functional properties [74], general specifications applicable to large classes of SCs would facilitate verifying SCs in a group. Ideally, the specifications of given SC classes can be written once and reused to test SCs of each class. Indeed, general specifications need to be adequately weak to be mapped with a class's functional properties. Furthermore, genuinely general specifications need to be free of any particular SC's state variables because other SCs belonging to the same class typically differ in name, type, and number. However, such general specifications are not well suited for existing verification tools such as VerX [74] and **Solc-Verify** [71], which assume that input SCs are annotated with expressions that refer to state variables (e.g., pre-conditions and post-conditions). Therefore, it poses a scalability issue since deriving such annotations for each SC from class-wide general specifications can be a labor-intensive process. Recently, authors in [22] introduce an automated method to check unannotated SCs against specs ascribed to a few manually-annotated SCs. Specifically, a concept of behavioral clarification, which signifies the inheritance of functional properties and an automated approach to inductive proof by synthesizing simulation relations on the states of related SCs is contributed.

EClone [68] is a clone detector tool for Ethereum SC. The authors introduce the concept of SC birthmark, which is a semantic-preserving and computable representation for SC bytecode. The tool detects clones by computing statistical similarity between two SC birthmarks. EClone has been evaluated on Ethereum SC and has shown the ability to accurately identify clones. **ESHield** [69] is an automated security improvement tool that protects SC against reverse engineering. The idea behind the tool is that by interfering with identifying the connections between basic blocks it is possible to increase the difficulty of recovering

Table 2
SC static analysis tools.

Tool	Problem addressed	Benefits	Analysis type, method, objective	Evaluation workload[a]
Slither [62], (2019)	Automated vulnerability detection, code optimization, code review	Speed, robustness, good tradeoff between detection rate and false positives	Static analysis, formal methods, non-functional compliance	1002 SCs, including Decentralized Autonomous Organization (DAO) [66] and SpankChain [67]
SmartCheck [57], (2018)	Vulnerability detection	Extensible, scalable and fast	Static analysis, formal methods, non-functional compliance	4600 verified contracts
Mythril [54], (2018)	SC security analysis	High accuracy due to exploration of all execution paths	Symbolic execution, control flow graph, formal methods, non-functional compliance	N/A
Securify [56], (2018)	Security analysis	Scalable, Captures critical violations, low false positive, high code coverage	Symbolic execution, formal methods, non-functional compliance	Uses more than 18 000 real-world SCs Etherscan
Simulation-based verification (SBV) [22] (2020)	Vulnerability detection	Large scale SC verification	Static analysis, formal methods, non-functional compliance	13 SCs
NPChecker [63], (2019)	Detecting non-deterministic payment bugs	Bug search beyond predefined vulnerability patterns	Static analysis, model checking, functional compliance	30K online contracts (3075 distinct) from mainnet
EClone [68], (2019)	Vulnerability detection, deployment optimization	Accurate identification of clones	Symbolic execution, formal methods, non-functional compliance	Mainnet Etherscan
EShield [69], (2020)	Security enhancement for protecting against reverse engineering	Protects against three different reverse engineering tools with little extra gas cost	Control flow graph, formal methods, non-functional compliance	20 266 SCs
Ethainter [64], (2020)	Checks information flow with data sanitization	Detects composite information flow violations, scalable, provide good tradeoff between precision and completeness	Control flow graph, model checking, non-functional compliance	882 000 from Ropsten testnet, and small random sample of contracts from Ethereum mainnet (240K)
SolidiFI [65], (2020)	Evaluation of six static analysis tools for finding SC security bugs	Automated, systematic	Static analysis, formal methods, non-functional compliance	Bugs are injected into source code of 50 SCs
Vulnerability detection (VD) [70], (2019)	ML-based detection of security vulnerability patterns	Time-efficient compared to traditional static code analyzers	Static analysis, machine learning, non-functional compliance	1000 SCs
Solc-Verify [71], 2020	Source-level formal verification	Discovers non-trivial bugs, proves correctness, allows modular verification, scalable	Static analysis, formal verification, non-functional compliance	37 531 SCs from Etherscan
SmartShield [72], (2020)	Bytecode rectification, makes SCs more gas-friendly and secure against common attacks	Scalability, correctness, cost reduction, gas-friendly	Static analysis, formal methods, non-functional compliance	28,621 real-world buggy contracts
Oyente [27], (2016)	Security testing	Bulk analysis, discovers new classes of security bugs in Ethereum SCs	Symbolic execution, formal methods, non-functional compliance	19 366 SCs from Ethereum BC
MPro [73], (2019)	Detection of *depth-n* vulnerabilities	Scalability, based on Mythril-Classic and Slither tools	Symbolic execution, formal methods, non-functional compliance	100 real-world SCs

[a]All tools are evaluated on Ethereum BC platform.

CFG. Eshield has been evaluated on more than 20K SCs showing that all the protected bytecode cannot be decompiled using existing reverse engineering tools.

Authors in [70] introduce a machine learning (ML)-based tool for detecting the patterns of security vulnerabilities in SCs. The tool uses static code analysis to label SCs that were verified on the Ethereum platform. The experimental results show good prediction performance with the average accuracy of 95%. **SmartShield** [72] is another tool for detecting vulnerable SCs. In addition to vulnerability detection, it rectifies such SCs by

Table 3
SC static analysis tools continued.

Tool	Problem addressed	Benefits	Analysis type, method, objective	Evaluation workload[a]
SCRepair [25], (2020)	Pre-deployment analysis, automated bug repair	Gas-aware, generates patches for vulnerable SCs	Symbolic execution, search-based methods, non-functional compliance	38 225 SCs from Etherscan
VerX [74], 2020	Verification of SC functional properties	Fast, less-expensive	Symbolic execution, formal methods, functional compliance	12 real-world projects (138 contracts), 83 safety properties
MAIAN [75], (2018)	Finds vulnerabilities from SC bytecode	Effective, practical	Symbolic execution, formal methods, non-functional compliance	9825 SC from Etherscan
Osiris [53], (2018)	Finds integer bugs in SC	Low false positives	Symbolic execution, formal methods, non-functional compliance	More than 1.2 million SC
ZEUS [55], (2018)	Verifies correctness and validates fairness of SC	Zero false negatives and low false positives	Symbolic execution, model checking, non-functional compliance	22 400 SC
Gasper [58], (2017)	Locates gas-costly patterns	Discovers 3 representative gas-costly patterns in SC bytecode	Symbolic execution, search-based, non-functional compliance	4240 SC
GASTAP [59], (2019)	Automatic gas analyzer for SC	Effective, automatically infers gas upper bounds	Symbolic execution, formal methods, non-functional compliance	More than 29 000 SC
Vandal [60], (2018)	Security analysis for Ethereum SC	Fast, robust, effective	Symbolic execution, formal methods, non-functional compliance	141 000 SC
EtherTrust [61], (2018)	Sutomated static reachability analysis for EVM bytecode	Practical, scalable	Static analysis, theorem proving, non-functional compliance	148 SC

[a]All tools are evaluated on Ethereum BC platform.

Table 4
SC hybrid analysis tools.

Tool	Problem addressed	Benefits	Analysis type, method, objective	Evaluation workload[a]
ETHPLOIT [76], (2020)	Automated exploit generation	Lightweight in solving unsolvable constraints, good coverage of exploits	Static analysis, fuzzing, formal methods, non-functional compliance	45,308 SCs from Etherscan
CONFUZZIUS [24], 2020	Bug detection with hybrid fuzzing	Addresses environmental dependencies	Symbolic execution, fuzzing, search-based methods, functional compliance	27 real-world SCs collected from 17 GitHub repositories
ContractFuzzer [77] (2019), [78] (2018), [23] (2018)	Fuzz testing of SCs	Setects real-world vulnerabilities, addresses path explosion problem, low false positives	Static analysis, dynamic verification, formal methods, non-functional compliance	6991 real SCs from Etherscan
SolAnalyser [79], (2020)	Automated security analysis	Detects more vulnerabilities than Oyente, Securify, Maian, SmartCheck and Mythril tools, scalable, low false positives	Static analysis, dynamic verification, formal methods, non-functional compliance	1838 SCs, and 12 866 mutated SCs

[a]All tools are evaluated on Ethereum BC platform.

securing their EVM bytecode. The tool can detect and rectify several security-related bugs, including state changes after external calls, missing checks for out-of-bound arithmetic operations, and missing checks for failing external calls. SmartShield was evaluated on more than 28K buggy SCs on Ethereum showing that 91.5% SCs were automatically fixed by SmartShield. Authors in [73] introduce **MPro**, a tool that combines static and symbolic analysis to analyze depth-n vulnerabilities in SCs for the purpose of SC testing. The benefit of combining symbolic and static analysis is better scalability compared to symbolic analysis alone. Furthermore, such an approach avoids the problem of false positives often present for static analysis tools. MPro is evaluated on 100 SCc showing that it is n-times faster than Mythril-Classic for detecting depth-n vulnerabilities, without compromising detection capability. **SCRepair** [25] is a general-purpose automated and gas-aware SC repair tool that uses genetic programming to search for a set of edits to the SC that fixes a given vulnerability. In addition, the authors introduce a gas dominance level for SCs which is useful for comparing the quality of patches based on their runtime gas. **MAIAN** [75] is a tool for identifying and verifying vulnerabilities on trace properties (e.g., identifying SCs that endlessly lock funds or leak them to random users) of Ethereum SCs during runtime, using inter-procedural symbolic analysis and testing. MAIAN labels the malicious SCs into three categories, namely greedy (i.e., lock funds indefinitely), prodigal (i.e., releases funds to arbitrary accounts instead of legitimate owners), and

Table 5
SC dynamic verification tools.

Tool	Problem addressed	Benefits	Analysis type, method, objective	Evaluation workload[a]
SoCRATES [80], (2020)	Test-case generation	Highly configurable	Dynamic verification, formal methods, functional compliance	1905 real SCs from Etherscan
Test coverage criteria (TCC) [81] (2019)	Test coverage	Covers complete transaction basic path set and bounded transaction interactions	Dynamic verification, formal methods, functional compliance	Pool-Shark application consists of 12 SCs
Deviant [82], (2019)	Mutant generation for Solidity SC	Easy to use	Dynamic verification, formal methods, functional compliance	3 projects with a total of 67 contracts
MuSC [83], (2019)	Mutation-based testing for SCs	Robust, novel mutation operators for Solidity, test report generation	Mutation, formal methods, functional compliance	4 real-world Ethereum DApps (SkinCoin, SmartIdentity, AirSwap and CryptoFin)
Solythesis [84], (2020)	Automated runtime detection of SC invariant violations	Provides source-to-source Solidity compiler, higher code coverage, low overhead	Dynamic verification, formal methods, functional compliance	23 SCs from ERC20, ERC721, and ERC1202 standards
Echidna [85], (2020)	Test generation for discovering infractions in assertions and custom properties	Fuzzing based on custom user-defined properties, easy-to-use, fast	Fuzzing, formal methods, functional compliance	VeriSmart and Tether BCs and their token contracts
SoliAudit [86] (2019)	Vulnerability detection	Detects 13 types of vulnerabilities, uses solidity machine code as learning features	Fuzzing, machine learning, non-functional compliance	18k SCs, 14,383 training samples and 3596 test samples
ReGuard [87], (2018)	Bug detection (i.e., Reentrancy Bugs), bulk analysis of SCs	Allow inputs as bytecode and solidity code	Dynamic verification, formal methods, functional compliance	5 SCs
SolUnit [88], (2019)	SC unit testing	Fast, testing through the reuse of SC deployment	Dynamic verification, formal methods, functional compliance	5 applications
EthRacer [89], (2019)	Detection of event-ordering (EO) bugs	Allows input as SCs before and after deployment on BC	Fuzzing, formal methods, functional compliance	10 000 SCs from Solidity source code repository and Ethereum BC
ModCon [90], (2020)	Model-based testing for SC	Customizable testing, test prioritization	Dynamic verification, model-based testing, functional compliance	Credit Management Application (CMA) SC application from WeBank and BlindAuction SC

[a]All tools are evaluated on Ethereum BC platform.

suicidal (a random account kills the SC or forcibly executes the *suicide* code).

4.2. Dynamic verification

Since dynamic verification checks a program code at runtime, it can replicate an attacker looking for vulnerabilities in the code under test. This can be achieved by feeding malicious or anonymous inputs to the specific SC functions. Dynamic verification can also validate the findings of a static code analyzer. Next, we discuss some popular dynamic verification methods along with some hybrid approaches that use both dynamic and static analysis (see Tables 4 and 5).

Authors in [82] present **Deviant**, a mutation-based security testing tool for Solidity SCs. Deviant automatically produces mutants[1] for a target SC and runs them with the predefined test-cases to assess the mutants' effectiveness. To reproduce several faults in Solidity SCs, Deviant uses mutation operators for all the distinct features of Solidity according to its fault model. The simulation results acquired by running Deviant to test three Solidity-based projects show that these tests have not yet achieved large mutation scores. The results further show that a test suite competent for the coverage statement and branch criteria of Solidity SCs does not surely give a high-level assurance of code quality. Such measurements advise Solidity developers to write more effective tests to deliver trustworthy code and decrease security risks.

EthRacer [89] is an automatic security analysis tool that runs on top of Ethereum bytecode and checks if changing the order of input events, i.e., event-ordering (EO) of SC results in differing outputs. These EO bugs are linked to the dynamic ordering of SC events, i.e. function calls, and could facilitate possible exploits of millions of dollars worth of cryptocurrency. The authors propose a concurrent program analysis technique to formulate a generic class of EO bugs that arise due to long permutations of such events. The authors show that the technical challenge in detecting EO bugs in SCs (even in simple ones) is the intrinsic combinatorial dispute in the path and state-space analysis. The experimental outcome shows that most SCs do not manifest varieties in outputs under reordered input events, which means they are EO-safe. EthRacer analyzes 10 000 SCs, out of which it flags 8% for vulnerabilities. Moreover, when SCs do show distinct outcomes upon reordering, it is observed that they are likely to have an unintended behavior most of the time. The comparison with Oyente reveals that EthRacer discovers all 78 true EO bugs that Oyente finds, along with 596 bugs that Oyente

[1] Changes in software code expected to induce errors.

fails to find. Authors in [24] propose a hybrid fuzzer that also uses symbolic execution called **CONFUZZIUS**. The tool aims to provide higher code coverage and detect more bugs using the evolutionary fuzzing and constraint solving method. In particular, evolutionary fuzzing is applied on the shallow parts of SC, while constraint solving generates inputs that meet difficult conditions that restrict the evolutionary fuzzing from investigating deeper paths. Moreover, data dependency analysis efficiently generates sequences of transactions to create specific SC states that may hide bugs. CONFUZZIUS uses a more efficient fuzzing approach than EthRacer, because instead of using an entirely random transaction order, it uses read-after-write data dependencies between transactions. Thus, it generates quicker and more useful combinations of transaction order dependencies.

In the scope of dynamic verification, researchers have used fuzzing-based methods for runtime testing of SCs. However, SC fuzzing presents some unique challenges that are unusual in the traditional fuzzer development approach [85]. More specifically, a considerable amount of engineering effort is needed to simulate the semantics of BC execution and the transaction sequence generation [91]. Furthermore, there is a challenge of finding SC inputs that produce disordered execution times [92]. In BCs such as Ethereum, where a certain amount of gas is needed to execute a transaction, SC design inefficiency can be costly, and malicious inputs can lock SCs by making all transactions require more gas than needed. Therefore, defining a quantitative upper bound on the gas usage is a critical fuzzer feature, besides more traditional correctness restraints. To address the issues mentioned above, the authors in [85] propose a security analysis tool called **Echidna**, which is configurable and supports high code coverage. Echidna works in two steps. First, it leverages a static analysis framework called **Slither** [62] to compile SCs and check them for constants and functions that directly handle Ether (ETH). Second, the fuzzing starts with an iterative method producing arbitrary transactions using the following (i) Application Binary Interface (ABI) given by the SC, (ii) critical constants defined in SC, and (iii) any previously collected sets of transactions from the corpus. A counterexample is automatically minimized upon a property violation to report the smallest and simplest transactions that trigger the failure. Optionally, Echidna can also provide a transaction set to maximize the coverage over all SCs under test.

ContractFuzzer [23] is an accurate and comprehensive hybrid tool that uses static analysis and a fuzzing-based approach for detecting seven types of Ethereum SC vulnerabilities. It contains an offline EVM instrumentation tool that performs instrumentation of EVM, so that an online fuzzing tool can oversee the execution of SCs to get the required information for vulnerability analysis. ContractFuzzer also provides a set of new test oracles that can accurately detect real-world vulnerabilities within SCs. The systematic fuzzing performed on 6991 real-world Ethereum SCs shows that ContractFuzzer had identified at least 459 SCs vulnerabilities, including the DAO and Parity Wallet. The performance analysis also shows that the tool detects more types of bugs, and has lower false positives than Oyente. Furthermore, authors in [80] propose **SoCRATES**, an extremely configurable and extensible framework to generate test cases for SCs. SoCRATES uses a federated organization of bots to mimic the complex interactions among many users. These bots interact with the BC based on a defined set of composable behaviors. In particular, the use of a society of bots in SoCRATES allows triggering faults simulating the multi-user interactions that are difficult to produce by using one bot. The aim is to spot programming defects hidden in complex and articulated interactions backed by SCs. The tool also aims to expose known and unknown faults in SCs currently published in Ethereum. **ETHPLOIT** [76] is another hybrid tool that combines fuzzing and static taint analysis to create exploit-targeted transaction sequences, which dynamically set hard constraints to simulate BC behavior. The tool was evaluated on 45,308 SCs and it discovered 554 exploitable SCs.

SolAnalyser [79] is a fully automated approach that uses static and dynamic analysis for vulnerability detection of Solidity SCs. SolAnalyser supports the detection of eight vulnerability types that are not supported by the state-of-the-art tools. Moreover, the tool can be easily extended to support other kinds of vulnerabilities. [79] also contributes a fault seeding tool that injects different types of vulnerabilities in SCs. SolAnalyser is evaluated by experimenting with 1838 real SCs, from which 12 866 mutated SCs are produced by artificially seeding eight distinct vulnerability types. The results show that SolAnalyser can identify the seeded vulnerabilities. Furthermore, SolAnalyser outperforms five popular analysis tools (i.e., Oyente, Securify, Maian, SmartCheck, and Mythril) by detecting all eight vulnerability types and achieving a high precision/recall rate.

Although there exist many security analysis approaches based on code coverage analysis, it appears that they are not enough on their own. This is because these approaches are not designed for testing SC's functional correctness. A complementary approach could be to use Model-Based Testing (MBT) [93] in which test automation is based on a model. Recently, authors in [90] propose **ModCon**, a model-based testing tool that uses an explicit abstract model of the target SCs to derive tests automatically. ModCon complements code coverage analysis methods by providing more flexible and reliable quality assurance solutions. Moreover, most of the existing security analysis tools used for testing of SCs are designed and analyzed for public BCs (e.g., Ethereum). These tools might not be suitable for enterprise SC applications. ModCon shows its effectiveness specifically for enterprise SC applications based on private BC platforms. It allows SC developers to input their test model for the SC under test. Naturally, the efficiency of MBT depends on the input test model and fault model. An interesting research challenge could be to investigate SCs and their faults to determine suitable fault models or test assumptions.

There exist tools for mutation testing of SCs, such as **MuSc** [83]. MuSc applies mutation operations at the Abstract Syntax Tree (AST) level and allows creating user-defined testnet. It also supports deploying and executing tests. The tool has been evaluated on 4 real-world Ethereum DApps (SkinCoin, SmartIdentity, AirSwap, and CryptoF), showing its efficiency in exposing defects in SCs.

Authors in [84] introduce **Solythesis**, which is the first source to source runtime validation tool that enforces global invariants with quantifiers on SC with low overhead. The tool overcomes the limitations of static analysis tools, which often have a large false-positive or false-negative rate. Solythesis incurs a CPU overhead of only 0.1% on average for 23 benchmark contracts. textbfSoliAudit [86] is a tool for assessing SC vulnerability using ML and fuzzing. First, it uses ML to verify 13 types of vulnerabilities, and then uses fuzzing for on-line transaction verification. The tool has been tested on close to 18k Ethereum SCs and has shown 90% accuracy, while finding vulnerabilities such as reentrancy and arithmetic overflow problems. **ReGuard** [87] is another fuzzing-based analyzer for detecting reentrancy bugs in Ethereum SCs. The tool runs fuzz testing on SCs by generating diverse random transactions. Using runtime traces, it finds reentrancy vulnerabilities. The tool has been evaluated on 5 Ethereum SCs, finding 7 new reentrancy bugs. Authors in [88] propose **SolUnit**, a tool for executing unit tests for SCs with the goal to reduce test execution time. To reduce time, the tool reuses the deployment and setup execution of an SC for every test run. The tool was evaluated on 5 projects, showing the reduction of test execution time by up to 70%.

5. State of the art: Performance testing

In the literature, there exist different approaches for performance testing of BC-Apps. The most common of these are benchmarking, monitoring, experimental analysis, and simulation techniques. These approaches aim to help developers evaluate different performance characteristics and identify bottlenecks, to improve the performance of BC-Apps. Next, we discuss these approaches in detail, along with their advantages and limitations. In Table 6, we summarize and compare different types of BC performance analysis benchmarking tools. There are several efforts from academia as well as from industry to develop benchmarking tools for performance analysis of BC platforms. All these tools are open-source. Caliper and DAGBench are from industry (i.e., IBM, and IOTA foundation), while the rest are from academia. Both Caliper and DAGBench are well documented and active (i.e., new versions are being updated continuously), and they are being actively used within the research community for BC-App platform performance evaluation. Apart from Caliper, DAGBench, and BBB tool, other tools mentioned in Table 6 are inactive at present.

5.1. Bechmarking

BlockBench [94] is an open-source benchmark tool developed to perform the performance evaluation of BC-Apps. The metrics it evaluates include Transaction Per Second (TPS), latency, scalability, and fault-tolerance. Authors in [94] use BlockBench to compare the performance of three major BC platforms (i.e., Ethereum, Parity, and Fabric). The authors also claim that BlockBench can be used for the evaluation of other private BC platforms by extending the workloads and BC adaptors. The evaluation of the three BC platforms is performed by first splitting BC functionalities into four concrete layers, and then each layer is evaluated against different workloads. The evaluation results show that (i) the consensus algorithm is the main bottleneck in HL Fabric v0.6 and Ethereum, and (ii) the Ethereum execution engine is less efficient compared to Fabric. BlockBench uses two workloads, namely YCSB [95] and Smallbank [96], to quantify the four performance metrics for the target BC platforms. The performance evaluation shows various performance deficiencies in the comparison of BC-Apps. These deficiencies depend on the design choices made by developers at different layers of the BC software stack. Moreover, the results show that current BC platforms are not well suited for large-scale data processing workloads. Some key limitations of BlockBench are as follows: (i) deployment of BC platforms to be tested is managed through bash scripts that do not offer abstractions over the target testbed, (ii) it collects metrics about performance (latency and throughput), but does not include system metrics like CPU, memory or disk usage (important to consider the overall footprint of BC technologies) nor functional metrics (e.g., number of connected peers), (iii) it does not offer testing for faulty behavior, and (iv) insecurity, as it is installed using root user privilege.

Hyperledger Caliper (HLC[2]) is a performance evaluation framework that, at the moment, supports benchmarking the following BC platforms: Hyperledger Besu, Hyperledger Burrow, Ethereum, Hyperledger Fabric, FISCO BCOS, Hyperledger Iroha and Hyperledger Sawtooth. Caliper framework consists of two main components, namely Caliper core (which defines system flow), and Caliper adaptor (which provides support for integration of various BC platforms). A predefined configuration file consists of benchmark workloads, and information required for interfacing the adaptor to the SUT is needed before running a test. During the performance testing, a resource monitoring module gathers resources (e.g., CPU, RAM, network, and I/O) utilization data. After the test, a test report containing the values for various performance metrics is generated. Caliper can also monitor server-related metrics through Prometheus.[3] To gain a better understanding of the Fabric BC platform, authors in [97] provided a detailed empirical study by using the Caliper tool. The study characterizes the performance of BC and highlights potential performance bottlenecks by using a two-phased approach. The first phase aims to understand the impact on performance of Fabric (considering TPS and latency metrics) when the configuration parameters, including block size, endorsement policies, number of channels, resource allocation, and state database choice, are used in the test environment. The evaluation results provide various guidelines on configuring these parameters. Moreover, the three performance bottlenecks or hotspots that are identified include (i) verification of endorsement policy, (ii) state validation and commit (with CouchDB), and (iii) sequential validation of associated policies of a block transaction. The second phase focuses on optimizing the Fabric by considering the obtained observations from performance analysis. Finally, a few limitations of Caliper tool include the lack of network emulation, which is essential for studying the impact of network failure or latency on a BC platform, and the lack of any functionality for resource reservation on scientific testbeds such as Grid'5000.

DAGbench [98] is a framework dedicated to benchmarking Directed Acyclic Graph (DAG) Distributed Ledger Technology (DLT), such as IOTA, Nano, and Byteball. The supported performance metrics include TPS, latency, success indicator, scalability, resource utilization, and transaction fee. DAGbench shares an approach similar to BlockBench and HLC tools, i.e., adopts a modular adaptor-based architecture. In such designs, users can select or develop the required adaptors to integrate different types of workloads and BC platforms that they want to evaluate.

BCTMark [99] is one of the most recent tools for benchmarking BC-Apps on an emulated network in a reproducible way. To illustrate the portability of experiments using BCTMark, the authors have conducted experiments on two different testbeds: a cluster of Dell PowerEdge R630 servers (Grid'5000) and one of Raspberry Pi 3+. Experiments have also been conducted on three different BC platforms (i.e., Ethereum Clique/Ethash, and HL Fabric) to measure their CPU consumption and energy footprint for different numbers of clients. The framework provides an abstraction of the underlying physical infrastructure and can be used to deploy quickly on any platform that supports the Secure Shell Protocol (SSH) protocol. In particular, BCTMark provides the following key advantages over other benchmarking tools: (i) playbooks written in Ansible and deployed with BCTMark can be used to deploy an arbitrary number of peers on any testbed that supports SSH connections, while providing the same abstraction over a network, enabling scientists to easily express network constraints and topology, (ii) collects both system metrics, including CPU, memory or disk usage and functional metrics (e.g., number of connected peers), and (iii) it can be used with the public as well as private BC platforms. As a limitation, it does not support testing for faulty behavior.

BBB [100] is a benchmarking framework for BC platforms, namely Boston Blockchain Benchmark (BBB). The BBB design choice allows emulating the underlying networking infrastructure. The participants can communicate with BC through the emulated network. This provides several benefits, such as support for fine-grained control over network metrics (e.g., network topology, link latency, and bandwidth) and the integration of various network-related attacks (eclipse [101], and Distributed Denial

[2] https://github.com/hyperledger/caliper.

[3] https://prometheus.io/.

Table 6
Benchmarking tools for BC-App performance evaluation.

Tool	Supported BC platform(s)	Supported performance metrics	Evaluation workload	Limitations
BlockBench [94]	Ethereum, Parity, and Hyperledger Fabric	Throughput, latency, scalability and fault-tolerance	YCSB, Smallbank, EtherId, IOHeavy, CPUHeavy, DoNothing	Insecurity (installed using root user privilege), no testing for faulty behavior, collects performance metrics but not system metrics such as CPU, memory/disk usage, nor functional metrics
Hyperledger Caliper [102]	Besu, Burrow, Ethereum, Fabric, FISCO BCOS, Iroha and Sawtooth	Resource consumption, transaction/read throughput and latency, success rate	All types of custom build loads are supported via config files	Lack of network emulation, lack of support for resource reservation on scientific testbeds
DAGbench [98]	IOTA, Nano, and Byteball	Resource consumption, throughput, latency, success rate, transaction data size and fee, scalability	Value/data transfers, transaction queries	Specific for DAG DLTs
BCTMark [99]	Ethereum Clique/Ethash, and Hyperledger Fabric	CPU consumption and energy footprint for different numbers of clients	Ad hoc load generation based on Python scripts and on an history	Testing for faulty behavior is not performed
BBB [100,103]	Ethereum, could be extended to other BC platforms	Tolerance against faulty behavior, network properties (e.g., latency, and bandwidth) affect BC performance	Not specified	Does not include failure injection loads

of Service (DDoS) attacks). Rather than designing and implementing a network emulator tool, the authors choose to seamlessly integrate the BBB tool with a widely popular (in academia and the industry) emulator called Mininet,[4] which is a battle-tested tool to emulate real-world network scenarios. Although BBB usage is only shown to evaluate Ethereum-based applications, the tool design is generic, and it is possible to extend BBB to evaluate other BC platforms. BBB developers claim that the tool is easy to install and use, as it allows every parameter configuration through a simple change of a YAML configuration file. Furthermore, BBB is extensible and configurable. As a limitation, it does not support failure injection loads.

5.2. Live monitoring

Performance benchmarking solutions usually require a standardized test scenario and well-documented workloads as input. However, in the case of public BC platforms, it is not feasible to have adequate control over the workload and the users participating in the consensus process. This makes the use of benchmarking approaches for public BC platforms challenging. Therefore, there are two potential solutions that could be used for evaluating the performance of public BC platforms. The first approach is to build a test network in a private setting, able to closely represent the test network of a public BC platform. Next, we can leverage the state-of-the-art benchmarking tools to evaluate the BC performance, by providing artificially designed workloads as inputs. This solution might need the development of a new adaptor in a benchmark tool for integrating either the workload or the BC network. In such a setting, one must consider the problem of BC scalability, which may arise when the tested private version is implemented publicly. The second approach includes monitoring and evaluating the performance of a live public BC with realistic workloads as input [104]. For instance, authors in [105] provide a real-time performance monitoring architecture that is based on logging. The solution is detailed, and it results in lower overhead and higher scalability, when compared with solutions that use Remote Procedure Call (RPC).

[4] http://mininet.org/.

5.3. Experimental analysis

Experimental analysis approaches are typically based on self-designed experiments [28,106,107] and simulators, i.e., software tools to mimic the behavior of real-world systems [108,109]. Next, we discuss the state-of-the-art performance analysis work that uses experimental-based performance analysis.

The performance of a private variant of Ethereum BC is studied in [106], by evaluating Ethereum's two popular clients, namely PoW based Geth, and Proof of Assignment (PoA) based Parity. The evaluation results depict that, on average, the Parity is 89.82% faster than Geth concerning transaction processing rate under various workloads. To measure the scalability of Ethereum BC, authors in [110] used a quantitative analysis approach, in which synthetic benchmarks over an extensible testing scenario are used to measure the transaction throughput. Based on the test outputs, it was observed that Ethereum platform can hardly achieve the three properties, which include decentralization, scalability, and security, concurrently.

With the release of long-term support for the HL Fabric v1.4 BC platform, the platform has been evaluated for performance by various researchers. For example, the impact of various workloads on networking infrastructure of HL Fabric v1.4 BC is investigated in [28]. The performance metrics include transactions per second (i.e., throughput), latency, and scalability (measured by evaluating the number of users serviced by the system in a specific time period). During the performance evaluation process, various parameters, like number of transactions, generation rate of new transactions, and transaction types (e.g., read or write) were varied to dynamically change the network load. Similarly, authors in [111] used an empirical approach to perform the performance evaluation of Sawtooth BC, which is a popular private BC platform under the umbrella of Hyperledger BCs. The metrics considered for performance analysis include consistency (to check if the system produces the same results each time under the same workloads and a cloud virtual machine configuration), stability (to check if the system's performance remains stable in the same workloads, but under the varying cloud VM configurations), and scalability (to check if the system performance stays scalable with varying parameters related to workloads and cloud VM configurations). It is also observed that by adjusting the configuration parameters, namely scheduler and maximum batches per

block, the performance of Sawtooth can be optimized. Finally, based on the results obtained from the empirical performance analysis in [28,111], it is concluded that Hyperledger BC platforms require improvements on geographical scalability (which is usually limited due to the tradeoff with network latency [112]) and size scalability (e.g., [94] shows that the system fails to scale with more than 16 peers). Moreover, a major performance bottleneck in scalability is PBFT, the consensus algorithm used in many permissioned BC platforms. It is because PBFT adopts a communication-bound mechanism for consensus instead of a computation-intensive PoW consensus.

In the initial implementations of BCs (e.g., Bitcoin and Ethereum), to add transactions in a distributed ledger, multiple user transactions are grouped to form blocks, which are added in an immutable linked-list type of data structure in the global chain. This process of updating the ledger does not allow the generation of concurrent blocks, thus providing low transaction throughput by limiting the number of transactions that can be added to the ledger per second. Alternatively, in the distributed ledgers that are based on the DAG concept, multiple transactions/blocks can be added simultaneously on different vertices of the directed graph, thus supporting parallel generation and inclusion of transactions/blocks. Inspired by this idea and the need to improve the transactions per second, several distributed ledgers use consensus algorithms that support such concepts of transaction generation and addition. For example, IOTA foundation BC uses a cumulative weight approach to confirm transactions, and Markov Chain Monte Carlo (MCMC) sampling method to select *tip* (i.e., the vertex in DAG where the newly confirmed transaction can be added) randomly. Other examples include Byteball[5] system, in which the consensus process relies on a mechanism that selects 12 reputable Witnesses who need to reach to consensus, and Nano,[6] which adopts a balance-weighted vote technique to achieve consensus on transaction commit. As per the design considerations and theoretical functioning details, the DAG-based DLTs have higher transaction per second. However, it is also essential to identify their performance bottlenecks and the tradeoff between different properties (i.e., scalability and latency). To this end, IOTA scalability in the IoT application scenario, consisting of a private network with 40 nodes, has been demonstrated in [107]. The evaluation results show that against the arrival rate of transactions, the processing rate (i.e., TPS) has adequate linear scalability.

5.4. Simulation

Next, we provide a discussion on popular simulators used for measuring the performance of BC-Apps.

BlockSim. Three simulators, all named BlockSim (or BlockSIM), were proposed in 2019 for simulating BC-based systems. First, the authors in [108] proposed an implementation of a framework called BlockSim, which aimed to create a discrete-event dynamic system model for BC-Apps that use PoW-based consensus algorithms. The BlockSim framework was developed by using a layered approach, and it consists of three layers: incentive, connector, and system. With the use of BlockSim simulator, the authors evaluated the performance regarding block creation time for DLTs with a PoW-based consensus algorithm, and it provided important insights concerning the block generation process in PoW. To further show the correctness and feasibility of the proposed solution, in their extension study, the authors used verified predefined test cases, where the results of the simulation were compared against the outcome of real-life BC platforms, such as Bitcoin and Ethereum. However, whether the simulator can be extensible or not, this fact is not yet established and needs further research. To better understand, evaluate, and plan the system architecture and its performance, the authors in [113] provide an implementation of their proposed comprehensive simulation tool called BlockSIM. It is open-source and can be used to simulate private BC-Apps. BlockSIM aims to measure system stability, and transaction per second for private BC platforms, by running them in various target scenarios. Based on the evaluation results, one can then decide about the optimal system parameters that suit best. The effectiveness of the BlockSIM is proved by comparing it with the private Ethereum network running PoA-based consensus.

Recently, another simulator to evaluate BC projects, with the name BlockSim, is proposed in [114], as a flexible discrete-event simulator. When running BlockSim for Bitcoin and Ethereum BC platforms, interesting observations related to the performance of BCs were drawn. For example, it was observed that the impact of doubling the size of blocks on block propagation delay is small (i.e., 10 ms), but if the communication is encrypted then the delay is greatly affected (i..e, more than 25%). Later, an extension of BlockSim, namely SIMBA (SIMulator for Blockchain Applications), is proposed in [115]. SIMBA includes the Merkle tree as an additional feature on BC nodes and shows that it improves simulation efficiency and allows more realistic experiments that were not feasible before. In particular, the use of Merkle trees provided huge improvements (up to 30 times) in verification time of transaction in a block reduction, without impacting the block propagation delay. Since the verification of block transactions plays a critical role in the overall computational load of network nodes, its improvements substantially affect overall performance of network nodes, and consequently, of the whole network.

DAGsim. To simulate DAG-based DLTs, authors in [109] proposed a continuous-time, multi-agent simulator, namely DAGsim. The tool was developed by IOTA foundation to mainly test the performance of their BC system (i.e., IOTA) concerning its transaction attachment probability. The analysis results revealed that agents (who play a similar role as miners) with lower latency and higher connection degrees exhibit higher chances that their transactions will be accepted by the network. Another multi-agent tangle simulator [116] is developed in collaboration with NetLogo. It simulates random uniform, as well as MCMC tip selection, while providing visuals (i.e., graphical user interface) and an interactive experience during simulation. There also exists research work that leverages simulations together with analytical approaches to perform validation and exploration. For example, authors in [117] present a generic DAG-based cryptocurrency simulator. The simulator is built using Python, and it was used to validate an analytical performance model. The results revealed that by issuing a transaction with a smaller average number of parents in DAG, it is possible to increase transaction per second.

PeerSim. Authors in [118] introduce a performance evaluation simulator specifically targeting P2P nodes. The simulator is modular and easy to configure. It is written in Java and each simulation is defined using a plain text configuration file similar to a Java property file. PeerSim supports dynamic scenarios including different failure models. One of its features is graph abstraction, which allows representing overlay networks as graphs.

P2PTester. Another simulator for P2P network performance is P2PTester [119], which is a Java-based application that can interface with any arbitrary P2P system. The simulator is designed with three goals in mind: genericity, scalability, and modularity. Genericity supports working with a wide range of P2P platforms. Scalability allows simulating the performance of a large number of real peers. Modularity enables fine-grained measurements of various components of a P2P system. In addition, the simulator

5 https://byteball.org/Byteball.pdf.

6 https://nano.org/en/whitepaper.

provides a user interface for specifying the simulation parameters, such as the number of peers, the type of peer overlay, the duration of simulation, and how to measure and report the results.

Finally, authors in [120] proposed a technique to predict the latency of a private Ethereum (Geth) platform by using a modeling tool called **Palladio Software Architecture Simulator** [121]. The evaluation shows a lower relative error in response time (mostly under 10%).

5.5. Comparative discussion of performance evaluation approaches

Next, we summarize and provide a comparative discussion along with the pros and cons of the above-mentioned different types of empirical performance evaluation solutions. Our comparison considers the generic properties of individual solutions, as well as their suitability for evaluating different BC platforms.

The performance analysis of BC platforms using live monitoring approaches requires (i) a testing deployment network that has high fidelity with respect to real-time production systems, and (b) realistic workloads and faultloads as its inputs. Although live monitoring approaches are better suited for evaluating public BC platforms, they can also be used for benchmarking the private BC platforms.

A common issue associated with evaluating a public BC is that it is difficult to change any parameters to vary experiments. In general, a benchmarking approach needs a controlled evaluation environment that mainly consists of a SUT network and artificial workloads. The eligible workloads and performance test metrics cannot be easily modified or tuned after the selection of a benchmark tool (because they are closely coupled). For example, in Blockbench, tuning of a network layer parameters (e.g., network delay) is not supported yet. Blockbench provides support for the evaluation of only three specific BC platforms (i.e., Ethereum, HL Fabric, and Parity). However, with the design of well-designed APIs, users can develop additional adaptors to extend Blockbench's functionality to support the evaluation of other private BC platforms. Therefore, compared to the benchmarking approaches, the extensibility of the monitoring approach is lower. Also, the availability of several well documented and open-source benchmarking tools makes them easier to deploy and use than the monitoring approaches.

Experimental analysis is one of the most common approaches adopted by researchers to evaluate the performance of BC-Apps, by using self-designed experiments. Experimental analysis has several similarities with the benchmarking approach, but there are two main differences. First, since self-design experiments are designed to test requirements and considerations associated with a given BC-App, they have a better capability of parameterization. For example, as mentioned before, Blockbench cannot measure the impact of a network delay in HL Fabric, which can be evaluated with self-designed experiments. Furthermore, unlike benchmarking tools, which are somewhat standardized and their functionality could be extended for evaluating various BC-Apps, tests in self-defined experiments are usually dedicated to a specific BC-App, thus limiting their extensibility.

For both benchmarking and experimental analysis approaches, the complexity of the testing environment depends partly on the complexity of a SUT and what performance parameters need to be evaluated. In contrast, a simulation-based performance testing environment exhibits high complexity only in the simulator design and development phase. Once a BC-App under test is deployed, the simulator provides extensibility, as the simulation can be used to test multiple configurations with different parameter values quickly and cost-effectively. Another inherent benefit of the simulation approach is that there are no dependencies on a physical testbed nor a BC platform. However, the evaluation results obtained from simulation could be quite different from the evaluation results obtained in other performance testing approaches. Additionally, there are several metrics (e.g., transactions per CPU/per memory second/per disk IO, and transactions per network data) for a BC-App that are difficult to evaluate in simulation.

Other than empirical approaches, there are research efforts on using analytical modeling for BC-App performance analysis. Analytical modeling uses mathematical tools to formalize a BC-App abstractly, and to solve the resulting models with precision. The model output (e.g., latency expressed as a function of network indicators) gives an analytical proof for a BC-App performance evaluation. Three popular analytical modeling approaches for performance analysis of DLTs are queueing models [122–124], Markov chains [125,126], and stochastic Petri nets [127].

Table 7
Performance metrics at different layers of the BC stack.

Application layer	TPS, Average response time, TP CPU, TP disk I/O, TP network data, TP memory/sec
SC layer	RPC response rate, SC execution time, State update time
Data layer	Encryption and hash function efficiency
Consensus layer	Consensus-cost time, Byzantine/Crash fault tolerance, Transaction finality
Network layer	Peer discovery rate, Transaction/block propagating rate, Resource consumption

5.6. Performance metrics

Metrics evaluated during BC-App performance testing play a huge role in measuring the effectiveness and correctness of the BC-App. These metrics can be broadly classified into two categories: *macro* (measured across the whole BC stack), and *micro* (evaluated for specific components at different layers of the BC stack) [105]. In particular, a BC-App performance from a user's viewpoint can be measured using macro metrics. These metrics include transaction per second, transaction processing latency, fault tolerance, scalability, transactions per CPU/memory second/disk IO/network data. From these metrics, the first two are measured frequently over all BC platforms. On the other hand, the micro metrics include peer discovery rate, RPC response rate, transaction propagating rate, SC execution time, BC state update time, consensus cost time, encryption, and hash function's efficiency. A well-designed workload should be used to evaluate both types of metrics. Concerning the benchmarking and monitoring approaches to BC performance testing, there are specific workloads designed for evaluating the performance of different BC layers. Table 7 shows the performance metrics for different layers of the BC stack.

6. State of the art: Security testing

In this section, we discuss the Security and Privacy (S&P) related aspects of BC-Apps that should be considered during an end-to-end BC-App security testing. We discuss various S&P aspects considering components at different layers of the BC stack. Please note that specific approaches to SC security testing are covered in Section 4.

BC technology supports the concept of security by leveraging public-key cryptography and primitives such as hash function and digital signature. However, this may give a false impression of the security provided. It is because all cryptographic protocols

have their limits and because holistic security includes technology, people, and processes, which are often overlooked in a BC security analysis. The main objective of security testing is to ensure that BC-Apps are secured against different types of threats caused by viruses and malicious programs injected by malicious entities. Specifically, in BC-Apps, the security analysis and imposed countermeasures should be extremely thorough and responsive due to the unique BC characteristics. For instance, an ongoing transaction cannot be stopped, and thus, the testing process should be effective enough to uncover all potential threats before the transaction deployment on BC. Table 8 shows a list of major threats along with their possible countermeasures across different layers of the BC stack. Security testing should exercise a range of different scenarios potentially leading to such various threats, while ensuring that measures are in place to properly handle such threats, if they arise after a BC-App deployment. There exists extensive literature that discusses an array of S&P threats in BC technology [18,128], which should be considered during the security testing and analysis of BC-Apps.

Since most of the threats in BC-Apps are associated with the data and processes (i.e., business logic), it is essential to understand the criticality of data and processes. In particular, understanding the sensitivity of the data being stored and processed in BC is needed before one starts the security analysis of such systems. To determine the importance of Confidentiality, Integrity, and Availability (CIA) of the data stored in a BC-App, one should first understand the associated regulatory implications and perform a business impact analysis. During security analysis, a comprehensive threat model that closely reflects the real-world adversary model needs to be adopted. The developers should ensure that the well known threats associated with PKI and application development (e.g., user key leakage, vulnerabilities in source code) are factored into the security analysis. Additionally, the security threats specific to a BC-App should be identified. These threats include attacks such as hijacking consensus procedure, DDoS, exploiting private BCs and SCs, and wallet hacking. Based on the identified threats, different scenarios representing one or more risks can be identified and evaluated for their likelihood and impact. Finally, based on the identified risks and their impact level, adequate security controls via security analysis procedures should be selected. Moreover, several well-defined security practices like source code review, secure key management, data protection via encryption methods, restricted data access via access control methods, and regular security monitoring can be deployed. Finally, security improvement techniques specific to the BC technology such as robust wallet management, authorized ledger management, and secure SC development should be employed. In particular, it is vital to understand that technology, processes, and people are equally important to secure BC-Apps. For example, the DAO hack damage could have been minimized, if an adequate governance structure and incident response process were in place.

In its default implementation, BC design does not provide support for data privacy, because all the transactions in the ledger are visible to all the network participants. However, pseudonymity is supported by allowing users to use public keys to transact instead of any identifying value. Depending upon the purpose of a BC-App, confidentiality and data protection considerations may be essential during the design and implementation of the BC-App. One common way to support data confidentiality is to utilize off-chain storage solutions, in which the organizations can store all or part of their data in local storage. To ensure data integrity for the data stored at local storage, the hash of the data is stored in the distributed ledger. Another approach could involve a fine-grained access control technique to regulate the access to the data stored on the ledger. Most of the solutions will require the use of cryptographic techniques, which are still the topic of active research. For example, a version of Zero-Knowledge Proof called zk-SNARKs [129], allows verifiers to validate a statement about encrypted data, while not revealing the corresponding decrypted data. Another alternative is to stop broadcasting transactions in the whole network, and instead, limit the dissemination and visibility to predefined parties in the network. An example of such a design is adopted in R3 Corda[7] BC, which uses an Unspent Transaction Output (UTXO) set model. Finally, the security of transactions is closely coupled with the underlying consensus algorithm, which results in an update in the distributed ledger. As advancements in DLT progress, developers have more options to choose which BC platform and consensus algorithm to use for a target BC-App. The selected design elements, along with their rules, will set various networking parameters such as transaction speed, latency, and scalability. Therefore, during the requirements analysis, or before developing a proof-of-concept, it is vital that developers and security engineers carefully evaluate the algorithms and protocols, to identify the ones that are best suited for their specific BC-App.

Consensus protocols are critical to BC security, therefore, understanding potential threats to their security is essential to securing the BC. To this end, a large number of consensus protocols have been proposed [130,131], some of them with the aim of improved security against various threats and high performance in large-scale networks. However, the adversaries are continuously targeting the consensus protocols, either to destabilize them or gain financial profits from the BC-App using these protocols. Therefore, the security testing should cover all the possible test cases and adversary models, to check if the consensus protocol used in a BC-App could be exploited for vulnerabilities, further leading to other security threats. In particular, the consensus protocol should be able to tolerate or quickly recover from faults, to ensure that it finds consensus and completes transactions even in a sub-optimal network topology. Also, BC-App developers should take precautionary measures (e.g., economic incentives, strong consistency, decentralization, and fast finality) to reduce the risk to consensus protocols. While performing security testing for consensus protocols against well-known threats, it should be ensured that the protocol can satisfy the following three essential properties: (i) consistency (the network peers should agree on a proposed value to reach a consensus in a certain time limit), (ii) transaction censorship resistance (resilience to malicious nodes blocking genuine transaction), and (iii) distributed denial of service DDoS resistance (resilience to malicious nodes launching DDoS attacks on consensus algorithms).

As discussed by authors in [14], a consensus algorithm must satisfy a number of security properties, and the same should be tested during its security analysis. These properties are as follows: (i) Authentication, it implies whether nodes participating in a consensus protocol need to be properly verified/authenticated, (ii) Non-repudiation, it signifies whether a consensus protocol satisfies non-repudiation, (iii) Censorship resistance, it implies whether the corresponding algorithm can withstand against any censorship resistance, and (iv) Attack vectors, it implies the attack vectors applicable to a consensus mechanism. The attack vector can be further divided into the three threats against which a consensus protocol should be tested. These threats include: (i) Adversary tolerance, which signifies the maximum byzantine nodes supported/tolerated by the respective protocol, (ii) Sybil protection, in which an attacker can duplicate his identity as required, to achieve illicit advantages. Within a BC-App, a sybil attack implicates the scenario when an adversary can create/control as many nodes as required within the underlying P2P network

[7] https://www.r3.com/corda-platform/.

Table 8
Various security threats at different layers of BC-Apps.

S&P threats	Potential adversaries	Possible defense mechanisms
Application layer - False data feeds, CIA and Front-Running attacks, attacks on availability and privacy, malicious Trusted Execution Environment (TEE) or token issuer, censorship, permanent hardware (HW) fault of TEE	Internal or external attackers (e.g., users, third-party service providers, malware), application/service developers, TEE manufacturers, token issuers, regulatory authorities	Multi-factor authentication, decentralized authority, reputation-based methods, application-level privacy-preserving constructs, HW wallets, redundancy
Smart contract layer - Exploiting SC specific bugs	SC developers, users, external attackers with lightweight node	Safe languages, static/dynamic analysis, formal verification, audits, best practices, mixers, Non-interactive zero-knowledge proof (NIZK), trusted HW, ring/blinding signatures, homomorphic encryption
Data layer - Quantum attacks, transaction data tampering attacks	Consensus nodes	Quantum-resistant cryptosystems, economic incentives, strong consistency, decentralization
Consensus layer - Protocol deviations, violation of assumptions	Consensus nodes	Economic incentives, strong consistency, decentralization, fast finality
Network layer - MITM attacks, availability attacks, network partitioning, routing attacks, DDoS, deanonymization	Providers of network services	Redundancy, protection of naming, availability, routing, anonymity, and data

to exert influence on the distributed consensus algorithm and to taint its outcome in her favor, and (iii) DoS resistance, which implies whether the consensus protocol has any built-in mechanism against DoS attacks. Finally, securing the consensus protocol should not come at the cost of low performance, i.e., it should not adversely impact the latency, throughput, and scalability, and a suitable trade-off should be considered. This trade-off strongly depends on the requirements of the target BC-App, and the same should be taken into consideration during the security analysis.

Depending upon its implementation nature (i.e., private or public), a BC networking infrastructure is based on either private or public networks. A private network uses centralized administration and it supports features such as low latency, user and transaction privacy, and compliance with regulatory obligations (e.g., HIPAA for healthcare data). Private networks inherently provide authentication and access control, and have full control over communication routes and network resources used, enabling suitable network topology regulation about the given requirements. The network administrators can apply fine-grained access control techniques to implement the security principle of minimal exposure. This way, the insider threats in a local network can be minimized. Authors in [18] observed that internal and external attacks are the specific security threats to permissioned BC platforms. For example, permissioned BC platforms use centralized access control that can be attacked by an external attacker by exploiting a network or system vulnerability. Such an attack is even easier to launch for internal attackers, as they might already have the required privileges or can get them by exploiting certain vulnerabilities in systems, network, or organizations involved in the BC-App. One result of such exploitation could be that the internal attacker can add malicious miners (nodes running consensus algorithms) or remove legitimate ones from the network. Such a change in the network will result in increasing the adversarial hash rate (aka consensus power) demonstrated at the consensus layer. Moreover, the attacker could launch many attacks, such as double spending, and attacks on violation of protocol assumptions, that can be leveraged with the increased hash rate at the consensus layer. Therefore, it is vital that security testing considers all these possible attacks before deploying permissioned BC-Apps. Testing the security of permissioned BC is easier compared to its counterpart, due to the controlled environment. On the other hand, the public BC platforms are, and should be, more secure by implementation, as they are built to deal with many unknown entities (including the malicious ones) participating in different activities of the BC ecosystem.

BC-App components such as peer nodes and APIs use underlying private or public network for communication. Depending upon the type of a BC-App, the peer nodes and their associated roles can differ in different implementation scenarios. These nodes use participants' own (in case of public BC) or an organization's (in case of private BC) networking infrastructure to communicate with the BC network. These networking infrastructures should be equipped with required fundamental security controls and measures, e.g., penetration testing, periodic vulnerability checks, log monitoring, endpoint vulnerability testing, and security patch or update management. The lack of defense mechanisms could lead to the compromising of client nodes, and a single compromised node can result in the loss of assets of the client associated with that node in case of public BCs. While in private BCs, such a malicious node will remain undetected, thus violating the privacy of the attacked client node by eavesdropping on the transactions performed by the node. To protect the client nodes from such attacks, BC-App developers can align to the BC security recommendations set out by Gartner.[8] The recommendations include steps such as taking a holistic view of security, and ensuring the risks are evident at the business, technical, and cryptographic levels. Moreover, same as with any technology implementation, all BC-Apps need to be analyzed for their readiness to handle a security threat, and should have incident response plans in place to handle critical security events during a BC-App life-cycle.

In summary, there exists a large number of security threats at different layers in the BC stack [18]. Still, apart from the security analysis of SCs, there is none or little support, in terms of tools or techniques, available in the state-of-the-art for performing security testing of various components of the BC-Apps (e.g., consensus protocols, peer nodes, and integrating endpoints). A point to be noted is that based on the research articles in literature, there exists a large number of security threats to standalone BC systems as well as BC-Apps, but the security incident types occurring in practice are significantly lower, mainly at consensus and application layers. At the application layer, a large percentage of security incidents are caused by exploiting a centralized component by external or internal adversaries, while at the consensus layer, the majority of incidents are caused via temporary violations of protocol assumptions by 51% attacks.

7. API and interface testing

The users of BC-Apps typically have available APIs to connect to the underlying BC components. A BC-App ecosystem comprises

[8] Gartner, Evaluating the Security Risks to BC Ecosystems, March 2018.

different components, all of which must be connected. Therefore, it is crucial that different APIs associated with these components are tested for their compatibility with each other. API testing plays a major role to ensure that the backend is functional. The APIs in BC-Apps need to be tested for errors including unauthorized access, encrypted data in transit, and cross-site request forgery. Furthermore, APIs developed to interact with BC need to be checked for errors such as starting an automatic call over a large number of transactions. Such a common mistake in the API development could be very costly, especially in BC networks where transaction processing has a cost (e.g., gas in Ethereum BC). The API tests need to ensure that all the interactions between applications/users and the backend (BC network) in a BC-App meet specifications. Furthermore, API testing needs to ensure that the performance of the interactions is correct and smooth, i.e., the application can process and format API requests optimally, and verify that all API requests/replies from the backend are handled correctly. Finally, API testing helps in BC's block verification process. It is because each block information has a unique hash that changes when there is any change performed in the block, and the APIs that fetch and verify these hashes can be validated through API testing.

Furthermore, all the APIs require thorough functional testing to ensure that there are no functional issues and that the service integration works seamlessly. In a practical BC-App scenario, there exist two types of interfaces. The first is between the BC and the application in which API is being integrated, and the second is between the various components of a BC-App. The former is more challenging to test efficiently, as it requires significant knowledge of both domains and specialized testing tools that can work with both these domains. For instance, usually the users of DApp interact with the BC using a web application browser. In the existing test scenarios, the methods that test the web applications [132] only consider browser-side code, while the SC analysis tools [55] consider only the SC code. This approach of independent testing makes it challenging to use these techniques as they are in the DApp setting. Therefore, in practice, the testing procedure should consider the fact that the testers need to work together to understand the interfacing between the browser program (e.g., JavaScript code) and BC programs (i.e., SCs). Finally, API testing should ensure that APIs are secure, simple, and that they provide a high level of performance.

8. Summary of the state-of-the-art

In this section, we provide a discussion of the advantages and limitations of the current state-of-the-art V&V efforts (i.e., testing tools and techniques) for BC-Apps.

Finally, in Table 9, we provide a complete overview of the tools discussed in the paper, categorized based on the type, method, and objective of testing, BC component addressed and target BC platform.

8.1. Smart contract testing

Despite the increasing interest in SC development, this engineering discipline remains somewhat a puzzle to numerous developers, primarily due to the unique design of SCs. Therefore, it is required that the research community investigates the critical questions related to the development process of SCs. These questions include: (i) what are specific differences between SC-based software and traditional software development, and (ii) how do these differences affect V&V of SCs.

We broadly classify SCs testing into two categories, namely static and dynamic approaches, including different testing methods and objectives that can be used with each of these two categories (Section 4). Authors in [16,65] provide a comparison of static and dynamic approaches concerning their performance, coverage of finding vulnerabilities, and accuracy, which could be referenced while selecting a testing approach for SC testing. Automation tools implementing static and dynamic analysis methods are convenient to use to analyze vulnerable SCs. However, tools that detect only specifically defined vulnerable patterns are of limited use, since their testing expanse is limited, as the defined patterns are rarely exhaustive.

As mentioned in Section 4.1, researchers have proposed many tools and techniques to discover bugs and vulnerabilities in SCs. Nevertheless, there are many recent incidents reporting various vulnerabilities in SCs. This questions the efficacy of the state-of-the-art tools and techniques used to detect vulnerabilities in SCs. This could be because most of the tools have been evaluated either on developer-generated data-sets and inputs, or on data-sets of contracts with a limited number of bugs. Mainly, the existing solutions do not provide the level of code coverage that could uncover all the threats in SCs, while false positives and false negatives remain high. Moreover, empirical studies of software defects have shown that it is possible to detect many defects using static analysis tools, but this is true only in theory due to limitations of the tools [133]. Recently, authors in [65] focus specifically on the undetected bugs (false-negatives), but also consider false-positives.

Formal testing is considered an important component of software development life cycle, which tests the software behavior and performance against the predefined specifications and requirements, and thus using a predefined set of possible input conditions. Formal testing practices from a traditional software could be used for testing SC-based software. For example, using the test environment in the SC development tool Truffle [134], one can create formal test cases for Solidity SCs based on certain mathematical logic and rules, and these tests can be executed to check SC properties. Although formal methods may be effective for SC verification, their limitation lies in the fact that if some important properties are left out during the verification, an SC could remain buggy. Therefore, one can never be certain that specified properties used during formal testing will detect all undesirable outcomes of an SC. Furthermore, formal verification frameworks could incur high time complexity in bug detection, be expensive and highly complex.

Recently, authors in [10] concluded that developers are facing several significant challenges during SC development, based on findings obtained from interviews and a survey. These challenges include (i) there is no practical way to assure the security of SC code, (ii) existing tools for development are still immature and exhibit limited functionalities, (iii) the programming languages and the virtual machines still have several limitations, (iv) performance obstacles are difficult to manage under a resource-constrained running scenario, and (v) online resources (including advanced/updated reports and community support) are still insufficient.

8.2. Performance testing

As an important component of BC research, performance evaluation plays a crucial role in improving BC-Apps. Although numerous tools and techniques for BC performance evaluation have been proposed and implemented in the literature, as mentioned in Section 5, only few of them have been well analyzed and evaluated. Due to the unavailability of standardized interfaces while running workloads, comparative analysis between various BC platforms is challenging (specifically for systems using different consensus protocols and data structures). Apart from the

Table 9
Complete overview table. Dyn: Dynamic, Hyb: hybrid, FM: formal methods, ML: machine learning, SB: search-based method, MB: model-based, Funct: functional, Non-funct: non-functional, BM: benchmarking, SIM: simulation, SC: smart contract, E2E: end-to-end, P2P: peer-to-peer. BC Platforms ET: Ethereum, P: Parity, F: Fabric, BE: Besu, BU: Burrow, BC: BCOS, I: Iroha, S: Sawtooth, IO: IOTA, N: Nano, BY: Byteball, C: Clique, E: Etash, BI: Bitcoin.

Tool	SC testing						Platform testing (Type)	Component/E2E	BC platform
	Static	Dyn	Hyb	Method	Funct	Non-funct			
Slither [62]	•			FM		•		SC	ET
SmartCheck [57]	•			FM		•		SC	ET
Mythril [54]	•			FM		•		SC	ET
Securify [56]	•			FM		•		SC	ET
SBV [22]	•			FM		•		SC	ET
NPChecker [63]	•			FM	•			SC	ET
EClone [68]	•			FM		•		SC	ET
EShield [69]	•			FM		•		SC	ET
Ethainter [64]	•			FM		•		SC	ET
SolidiFI [65]	•			FM		•		SC	ET
VD [70]	•			ML		•		SC	ET
Solc-Verify [71]	•			FM		•		SC	ET
SmartShield [72]	•			FM		•		SC	ET
Oyente [27]	•			FM		•		SC	ET
MPro [73]	•			FM		•		SC	ET
SCRepair [25]	•			SB		•		SC	ET
VerX [74]	•			FM	•			SC	ET
MAIAN [75]	•			FM		•		SC	ET
Osiris [53]	•			FM		•		SC	ET
ZEUS [55]	•			FM		•		SC	ET
Gasper [58]	•			SB		•		SC	ET
GASTAP [59]	•			FM		•		SC	ET
Vandal [60]	•			FM		•		SC	ET
EtherTrust [61]	•			FM		•		SC	ET
ETHPLOIT [76]			•	FM		•		SC	ET
CONFUZZIUS [24]			•	SB	•			SC	ET
ContractFuzzer [77]			•	FM		•		SC	ET
SolAnalyser [79]			•	FM		•		SC	ET
SoCRATES [80]		•		FM	•			SC	ET
TCC [81]		•		FM	•			SC	ET
Deviant [82]		•		FM	•			SC	ET
MuSC [83]		•		FM	•			SC	ET
Solythesis [84]		•		FM	•			SC	ET
Echidna [85]		•		FM	•			SC	ET
SoliAudit [86]		•		ML		•		SC	ET
ReGuard [87]		•		FM	•			SC	ET
SolUnit [88]		•		FM	•			SC	ET
EthRacer [89]		•		FM	•			SC	ET
ModCon [90]		•		MB	•			SC	ET
BlockBench [94]							Performance BM	E2E	ET, P, F
HL Caliper [102]							Performance BM	E2E	BE, BU, ET, F, BC, I, S
DAGbench [98]							Performance BM	E2E	IO, N, BY
BCTMark [99]							Performance BM	E2E	ET, C, E, F
BBB [103]							Performance BM	E2E	ET
BlockSim [108]							Performance SIM	E2E	BI, ET
SIMBA [115]							Performance SIM	E2E	BI, ET
DAGsim [109]							Performance SIM	E2E	IO
Tangle sim [116]							Performance SIM	E2E	IO
DAG-based sim [117]							Performance SIM	E2E	N/A
PeerSim [118]							Performance SIM	P2P	N/A
P2PTester [119]							Performance SIM	P2P	N/A

requirement to evaluate the performance of basic functionalities in the most popular BC platforms (e.g., HL Fabric and Ethereum), there is a requirement for performance testing of the newly proposed BC functionalities. For example, sharding approaches [135] have been implemented in many BC platforms as a viable solution to BC scalability, however, such shard-based BC platforms have not been compared for their performance. Therefore, it is not clear how the use of sharding could impact the performance of the target BC-App. Moreover, support is needed for evaluating the impact on the performance of BC platforms implementing different solutions, such as sharding vs. DAG, and off-chain vs. side-chain. Additionally, it would be beneficial to evaluate a BC performance by combining empirical and analytical performance evaluation techniques.

8.3. Security testing

The lack of best practices along with innovative tools and techniques for security testing of BC-Apps is an important reason that makes BC security a significant challenge. In BC-Apps, vulnerabilities that lead to security threats are not limited just to the BC platform and SCs. Other aspects are susceptible to vulnerabilities, such as governance, compliance, human errors or misunderstanding, and assert at risk, which should also be evaluated. Consequently, BC-Apps should embed security mechanisms specifically targeted to various components lying at different layers of the BC stack. This should be done in the application design phase, instead of putting security patches afterwards, when a vulnerability is detected. Such an approach to security can hopefully provide a robust defense, and make a BC-App cyber-resilient.

We suggest that for each layer of our BC-layered model, secure cryptographic primitives with recommended key lengths based on existing standards [136,137] are considered. Examples involve secure communication (i.e., network layer), the use of digital signatures that are based on private keys for signing the transaction (i.e., consensus layer), and login credentials management for BC-oriented services (i.e., application layer).

To provide end-to-end security in BC-Apps, it is required to ensure that all the individual components of BC (e.g., SCs, consensus algorithms, peer nodes, peer-to-peer network, and an offchain storage system) are tested for vulnerabilities [18]. Furthermore, it is required to analyze and test new threats arising from the integration of the BC technology with the target application. Based on the state-of-the-art presented in Section 6, we observe that many security attacks happen at the application layer, due to exploiting a centralized element by external or internal adversaries. In contrast, for the consensus layer, many attacks happened due to a temporary breach of protocol hypotheses by 51% attacks. Therefore, further research is required to propose solutions to address these threats at both the layers. Moreover, to manage safe and correct software at each of the layers, alike to the contract and execution layer case, developers should employ verification tools, code reviews, testing, audits, known design patterns, and best practices. We believe that as BC-Apps mature with time, new security threats will be discovered, but at the same time there will be a more mature security testing strategy and a rich set of BC-specific security testing tools available. However, at present, it is safe to conclude that there is a lack of comprehensive security testing frameworks that could be used to perform systematic testing of BC-Apps.

9. Open issues and future research directions

In this section, we summarize various open issues related to BC-Apps V&V, and derive future research directions based on our comprehensive study of the state-of-the-art.

9.1. Open issues

- **Lack of best practices for developing BC-Apps:** As BC is a new technology, there is a lack of technology understanding, and the lack of skills and experience in designing and developing BC-Apps. Moreover, the lack of standardization in the usage of BC concepts and terminologies (i.e., conceptual and architectural ambiguity) leads to decreased clarity, quality, and productivity of the BC-App development. Furthermore, heterogeneous domain knowledge, such as technical, non-technical, legal (e.g., GDPR compliance rules), and the target application are critical for effective and exhaustive testing of BC-Apps.
- **Lack of best practices for testing BC-Apps:** BC-App development is a rapidly evolving area where efforts are largely focused on the core technology development. This may imply that testing is given less importance over programming, leading to the BC-App development ecosystems with few or no experienced testers to evaluate a developed application. Moreover, due to the complexity of the BC ecosystem, the scope of a minimal set of tests (i.e., sanity check) for BC-Apps can expand dynamically, to a significant margin.
  One of the factors determining the scope and level of testing required for BC-Apps is the choice of the used BC-platform. For public platforms, such as Ethereum or Openchain, testing efforts will be comparatively lower than for a customized platform that is purpose-built for an organization's needs. It is because public BC platforms have more evolved testing recommendations and guidelines, as well as testing methods developed and improved by a vast community over time. In contrast, in-house BC-App implementations need a detailed test strategy framework based on the functionality that is custom-developed.
- **Lack of mature development and testing tools:** The lack of efficient development tools is a significant hurdle for BC-App development. In particular, BC-Apps need an integrated development environment that offers the required linters and plugins, a build tool and compiler, a deployment tool, a testing framework, and debugging and logging tools. Although few versions of such tools exist, they are not yet adequate to satisfy various needs of BC-App developers.
  There exist open-source BC development frameworks, such as Ganache, Hyperledger composer, Ethereum Tester, Exonum Testkit, and Embark, that have some built-in features for testing BC-Apps. However, the available testing support in these frameworks is limited, the available tools are not standardized, or they include a limited set of testing features. The choice of testing tools heavily depends on the underlying BC platform. As a BC-App consists of various BC components, such as SCs, peer-to-peer networks, distributed systems, and consensus protocols, a set of specialized testing tools is needed for testing each of these components and their integrations.
  Several programming languages like Go, Solidity, and Ruby are increasingly used for BC-App development. However, compared to older programming languages (e.g., Java), there is a lack of enhanced testing and debugging suites distinct to these languages.
- **Lack of pre-production environments:** To comprehensively test a BC-App before deployment to production, we need a testing environment that resembles a production environment with high fidelity. The deployment of the BC network for testing a BC-App requires a huge effort. Although a few deployment automation utilities are available in the market offering such functionality via blockchain-as-a-service (BaaS), these BaaS are limited concerning the supported BC platforms and hardware infrastructure. If such support is not available, then a significant part of time and resources needs to be invested in setting up a testing environment or spawning from the real implementation. To address this issue, few tools with open-source implementations exist for generating test cases, which do not exactly reflect the test cases of a real-world scenario, but they can still be effective to test some of the transaction functionalities. Compared to the public BC, setting up a testing environment is easier in private BCs, as it can be set up by configuring the deployment tools with customized functionality.
- **Blockchain immutability:** Implementing BC-Apps without paying special care to immutability carries a significant asset risk to institutions or users. Immutability is an inherent property of BC that ensures data integrity and auditability. Moreover, immutability supports a secure and transparent nature of BC. It guarantees that the data stored on the BC ledger are tamper-proof (i.e., it cannot be removed or modified). However, immutability repudiates many privacy requirements and data protection rights when personal data is involved as a BC asset. Among others, it disputes the Right-to-be-Forgotten (RtbF), described in the new EU data protection act (i.e., GDPR), according to which individuals have the right-to-delete their data if specific provisions apply [138]. Therefore, BC-App testers need to have adequate knowledge about various data protection acts. Furthermore, compliance testing needs to be carried out to ensure that a BC-App does not breach any of these acts. This implies that BC immutability calls for an interdisciplinary approach to BC-App testing.

Furthermore, BC transactions are used to invoke various functions in SCs. Therefore, users must be aware of the possible outputs of a transaction they are performing. SC developers could also place the required checks that invalidate the transactions that are trying to invoke a wrong function in SCs. To this end, testers need to ensure that SCs are checked for such possible transaction validation and verification codes before their deployment in the BC network.

- **Performance evaluation:** The aim of performance testing in BC-Apps typically includes identifying performance bottlenecks, defining metrics (e.g., latency and transactions per second) for tuning the application, and assessing whether it is production-ready. Accurate and comprehensive performance testing with varying workloads and fault-loads is a key to gathering insights into how a BC-App will perform in a production environment, under specific loads and network conditions.
  A BC-App consists of many components, and its overall performance is tightly coupled with the individual and combined performance of these components. This, along with several dynamic events (e.g., rate of input transactions, network conditions, and dependencies), makes an end-to-end performance testing of BC-Apps a significant challenge in the BC ecosystem. For instance, testers need to predict variances in performance test cases, because a transaction commit latency varies with the size of the P2P network and the transaction volume. Data types and server locations can also influence performance. Moreover, for performance testing, a high-fidelity replica of the production scenario is needed, but replicating real-world transactions and the transaction processing latency is difficult. Let us take an example to understand this issue better. To initiate a transaction in the Bitcoin system, miners have to confirm and validate the transaction, which could get delayed due to a surge in usage. Also, the distributed ledger that powers BC needs to reflect the same order of transactions at each network node. Since the latency across different consensus mechanisms may vary, testers have to perform peer/node testing to ensure the consistency and performance of newly committed transactions. Furthermore, to ensure the integrity of the network and the ledger, all these newly added transactions should be provided in the proper sequence. Hence, considering the above issues, it can be challenging to replicate the production system in a dummy environment.
  One way to overcome some of the above-mentioned performance testing challenges is to have all the details, such as block size based communication latency, network size, expected size of a transaction, and time taken to process different queries stored on ledger data. In particular, testing for block size and ledger size is essential, as improper validation of these elements leads to a BC-App failure. Moreover, automated performance testing could be a key to assessing the overall scalability of a BC-App. Finally, in BC-Apps, identifying the adequate trade-off between the consistency of ledger data and the availability and partition tolerance by setting the optimal values of different network parameters during performance testing is critical to the success of the target BC-App.
- **Dependability evaluation:** During a BC-App performance testing, one of the critical assessments that needs to be performed using a comprehensive set of fault-loads is the application dependability. However, due to the distributed nature of BC, performing dependability assessment for BC-Apps is a challenging task. Furthermore, it is important to assess fault-tolerance levels and the performance of a BC-App, to provide evidence of the offered guarantees for the application performance and security levels. However, at present, such assessments are partially conducted because developers rely on the chosen BC platform.
  Fault-injection (emulating custom faults in a BC-App under test) is considered one of the most reliable techniques to assess the dependability of a distributed system. During fault-injection, fault-loads of different granularity are sent at different levels in the target application under test, to exercise its dependability under a various and rich set of possible issues (e.g., vulnerabilities and faults) that may occur in practice. Therefore, the need for developing the tools that can produce realistic fault-loads at different levels (i.e., system, network, and software) is a key challenge for efficient dependability assessment. For instance, at the system level, fault-loads should test the ability to tolerate process hangs and memory leaks. At the network level, fault-loads should test the ability to handle network partitions and message losses.

9.2. Future research directions

Based on the comprehensive survey of the state-of-the-art on BC-App V&V and identified open issues, we derive the following suggestions for future research directions.

- **Practical guidelines for BC-App testing:** There is a lack of standardized best practices for developing BC-Apps, which if followed, could alleviate some of the open issues of BC-App testing. For instance, at present, SCs follow a non-standard software development life cycle, according to which a deployed BC-App can hardly be updated, and bugs can only be resolved by releasing a new version of the software (i.e., hard fork). Therefore, one of the important research directions for BC-App testing is establishing practical guidelines that could support BC-App developers to carry out testing specific to BC-Apps. Such guidelines would help make sure that all necessary steps are made for ensuring that the developed BC-App will work as expected in the real-world environment. The best practices defined in the vast literature on software engineering for testing traditional software applications help, but only partially. It is because BC-Apps exhibit some unique properties (e.g., immutability, and transparency) and new components (e.g., SCs, and consensus algorithms) that need comprehensive understanding before standardized test guidelines can be defined.
  Furthermore, with a rapid development and deployment of BC-Apps, the use of standard best practices for BC source code review is particularly encouraged. When feasible, having a practice of peer-review and software testing performed by an external independent team before BC-App release can be beneficial. Another important practice that such testing guidelines can contain is the usability testing of BC-Apps. Usability testing aims to evaluate the overall quality of the software from the user's perspective. Current research shows that user experience of interacting with BC-Apps can be a significant concern [139]. For example, users face a lot of hurdles in interacting with BC-Apps, if the decentralization tenet of BC is kept intact. However, such hurdles are hard to evaluate, due to the lack of proper usability evaluation tools for BC-Apps. To that end, an interesting research direction would be developing the client-side of BC-based software such to honor the decentralization without compromising user experience.
- **Compliance testing:** The inherent features of BC, such as transparency and immutability, could cause S&P related threats to data owners, when their personal data is being

processed and managed via BC-Apps. For instance, if there is a requirement for BC-Apps to comply with the "right to be forgotten" regulations, then there is a conflict with data immutability provided by the BC. Such issues make BC-App compliance testing important, because failing to comply with regulatory bodies (e.g., GDPR or HIPPA) can have serious consequences, such as fines, negative press, revenue decline, and even jail time. Furthermore, compliance testing also ensures that all data privacy-related risks stated in the associated regulation acts are taken into consideration during the design of BC-Apps. Compliance testing checks BC-Apps against non-compliance with regulations or confidentiality agreements governing data. For instance, the following issues could be evaluated during the compliance testing, related to data privacy regulations: (i) does the application involve personally identifiable information (PII) or confidential freight data, and (ii) do the application requirements allow on-chain data storage, or data must be stored off-chain. The importance of compliance testing, as well as the lack of support for BC-App compliance testing, shows a significant research gap in this area that needs to be filled.

- **Ecosystem and third-party risks analysis:** Compared to the BC-Apps where BC technology is integrated within the target application, the standalone BC platforms (such as Bitcoin, and Ethereum) have proven secure till now. However, the security of a BC-based solution relies on all the applications that are part of the ecosystem in which the BC is integrated. Often, such an ecosystem consists of multiple organizations and third-party service providers (e.g., SC developers, and wallet and payment platforms). This heterogeneity of autonomous organizations makes testing the BC ecosystem as a whole challenging. For instance, different organizations may use different types of devices, communication protocols, and security protocols. Specifically, an organization's BC-App consisting of third-party BC solutions and platforms is as secure as its weakest link across all the technology provided. The security considerations of a public BC differ from the security requirements of each organization or a service provider taking part in the BC ecosystem. Therefore, to avoid vulnerabilities caused by third-party services, it is required to do a thorough vetting of parties involved in the ecosystem. The vetting phase should be accompanied by comprehensive testing that could ensure that the performed security tests cover the risks associated with the usage of third-party solutions integrated in the BC-App.
- **Tool automation:** Since the deployment of BC-Apps is still in early phases, there is a lack of automation tools for developing, deploying and testing BC-Apps. The lack of such tools makes the testing expensive and time-consuming, thus discouraging thorough testing altogether. Moreover, the highly competitive market to provide BC-Apps and services further adds fuel by creating a race condition between different BC-based service providers. Therefore, there is a significant need for automation tools for BC-App development, deployment and testing. For instance, the BC network deployment process is usually complicated and therefore should be automated, so that the time and resource consumption of testing can be reduced significantly. Some deployment automation utilities[9] exist on the market, typically as part of a blockchain-as-a-service offering. However, they are limited in terms of the supported BC platforms and hardware infrastructure. Moreover, they fail to capture high-level design decisions, and thus do not represent the high-fidelity testing setups when compared with the production environment. Manual test generation is likely to form an important component, but inevitably is limited, therefore, there is a need for effective automated test generation and execution tools. To this end, researchers have already started to work towards the creation of automation tools for SC testing [82,85]. Moreover, there are few automation tools for performance testing [98]. However, these tools are at the early stages and do not support the automation of all the required functionalities and operations [17,140]. For example, Caliper tool takes a configuration file as input, to represent the workload, and falutloads to evaluate the performance of the system, but it does not allow SC functions to interact with the fabric during the evaluation process. Thus, it does not truly automate the performance analysis process when complex SCs are involved that need to interact with the BC system during its execution. Therefore, another future research direction could consist in developing specialized automation tools for testing different components of BC-Apps.
- **Endpoint vulnerability testing:** With a growing demand to release BC-Apps rapidly, there is often the risk of deploying a partially tested code, sometimes even on live BCs. For any new technology, its endpoints are most vulnerable, thus an easy target for the adversaries to launch malicious attacks. The BC endpoints, such as digital wallets, specific devices, or any user-side applications, are interfaces for users/clients to connect with the BC-App. If an adversary can compromise (i.e., get possession of a user's password or private key, or get physical access to the devices) any of these endpoints, it can get access to the user accounts, unless enhanced security measures, such as multi-factor authentication or fine-grained access control, are in place. Such malicious access, if successful, can put everything in the user account at risk. For example, the attacker can misuse the account without raising any external alarms or leaving signs of any abnormal behavior. In private or centralized systems such as banking, there are possibilities to detect and even correct (e.g., reverse) a malicious transaction. However, the decentralized and immutable nature of public BCs does not support such corrections, and the same applies to private BCs up to a certain point. Therefore, it is essential to thoroughly test the interfaces used to access the BC infrastructure. To ensure that the client-side applications are secure, progress has been made in protective measures, such as the use of cold wallets, often along with the Hardware Security Models (HSMs), which are being implemented by various companies. Unlike hot wallets that need internet connection and that store all the account information (e.g., password and private keys) in an online storage, the cold wallets work in an offline mode, thus making them hard to compromise. In summary, it is essential to carefully check and address all end-point vulnerabilities that exist at the integration points of different components of BC, and between the BC and endpoints of the application in which the BC is integrated.

10. Conclusion

In this paper, we provide a comprehensive study on the testing of BC-Apps, which includes the challenges it faces, the techniques and tools available to test various BC components, and the different types of testing required for it. Moreover, we identify a set of research gaps that need attention from the research community working on the topic. As concluded from our study, the key component requiring extensive and rigorous testing are

[9] https://www.ansible.com.

SCs, and the same have received a lot of attention from researchers that have proposed different tools to test SCs for finding bugs and vulnerabilities. Such tools aim to automate BC testing, improve code coverage, and achieve low false positives and negatives. Next, there exist a few tools that provide support for performance testing, but these tools are at their primary phases, and require further improvement and new test features. The efforts towards the security testing of BC-Apps are limited. There are research works that address the security of individual BC components, but tools that could access the security of the whole BC stack (i.e., end-to-end threat detection) do not exist yet. Finally, techniques to test the performance and security of consensus algorithms and BC nodes need to be explored. The issues related to the testing of applications that use SCs and BC technologies raise huge concerns to developers. It is because the rapid usage of these technologies in industries is currently worth billions of dollars. Hence, the testing of these technologies needs testers from multiple research domains (e.g., distributed systems, new coding languages, formal V&V methods, and cryptography) to work together to perform inter-domain research activities. Achieving significant progress on these issues would be difficult if the research challenges and gaps mentioned in this paper are not addressed properly. To this end, we hope that the testing challenges and research gaps highlighted in this work will help the research community to coordinate and work together to improve SCs and BC-Apps in general.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by the Research Council of Norway, grant number 288106.

References

[1] D. Di Francesco Maesa, P. Mori, Blockchain 3.0 applications survey, J. Parallel Distrib. Comput. 138 (2020) 99–114.
[2] J. Xie, H. Tang, T. Huang, F.R. Yu, R. Xie, J. Liu, Y. Liu, A survey of blockchain technology applied to smart cities: Research issues and challenges, IEEE Commun. Surv. Tutor. 21 (3) (2019) 2794–2830.
[3] D.C. Nguyen, P.N. Pathirana, M. Ding, A. Seneviratne, Integration of blockchain and cloud of things: Architecture, applications and challenges, IEEE Commun. Surv. Tutor. (2020) 1.
[4] Y. Liu, F.R. Yu, X. Li, H. Ji, V.C.M. Leung, Blockchain and machine learning for communications and networking systems, IEEE Commun. Surv. Tutor. 22 (2) (2020) 1392–1431.
[5] R. Koul, Blockchain oriented software testing - challenges and approaches, in: 2018 3rd Int. Conf. for Convergence in Technology (I2CT), 2018, pp. 1–6.
[6] P. Praitheeshan, L. Pan, J. Yu, J.K. Liu, R.R.M. Doss, Security analysis methods on ethereum smart contract vulnerabilities: A survey, 2019, ArXiv, abs/1908.08605.
[7] S. Porru, A. Pinna, M. Marchesi, R. Tonelli, Blockchain-oriented software eng.: Challenges and new directions, in: 2017 IEEE/ACM 39th Int. Conf. on Software Eng. Companion (ICSE-C), 2017, pp. 169–171.
[8] C. Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, first ed., A Press, USA, 2017.
[9] A.A. Donovan, B.W. Kernighan, The Go Programming Language, first ed., Addison-Wesley, 2015.
[10] W. Zou, D. Lo, P.S. Kochhar, X.D. Le, X. Xia, Y. Feng, Z. Chen, B. Xu, Smart contract development: Challenges and opportunities, IEEE Trans. Softw. Eng. (2019) 1.
[11] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts SoK, in: Proceedings of the 6th Int. Conf. on Principles of Security and Trust - Volume 10204, Springer-Verlag, 2017, pp. 164–186.
[12] A. Singh, R.M. Parizi, Q. Zhang, K.-K.R. Choo, A. Dehghantanha, Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities, Comput. Secur. 88 (2020).
[13] J. Liu, Z. Liu, A survey on security verification of blockchain smart contracts, IEEE Access 7 (2019) 77894–77904.
[14] M.S. Ferdous, M.J.M. Chowdhury, M.A. Hoque, A. Colman, Blockchain consensus algorithms: A survey, 2020, [Online]. Available: https://arxiv.org/abs/2001.07091.
[15] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, R. Hierons, Smart contracts vulnerabilities: a call for blockchain software engineering? in: 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2018, pp. 19–25.
[16] M. di Angelo, G. Salzer, A survey of tools for analyzing ethereum smart contracts, in: 2019 IEEE Int. Conf. on Decentralized Applications and Infrastructures (DAPPCON), 2019, pp. 69–78.
[17] C. Fan, S. Ghaemi, H. Khazaei, P. Musilek, Performance evaluation of blockchain systems: A systematic survey, IEEE Access 8 (2020) 126927–126950.
[18] I. Homoliak, S. Venugopalan, D. Reijsbergen, Q. Hum, R. Schumi, P. Szalachowski, The security reference architecture for blockchains: Towards a standardized model for studying vulnerabilities, threats, and defenses, IEEE Commun. Surv. Tutor. (2020).
[19] J. Leng, M. Zhou, L.J. Zhao, Y. Huang, Y. Bian, Blockchain security: A survey of techniques and research directions, IEEE Trans. Serv. Comput. (2020) 1.
[20] M. Dabbagh, K.-K.R. Choo, A. Beheshti, M. Tahir, N.S. Safa, A survey of empirical performance evaluation of permissioned blockchain platforms: Challenges and opportunities, Comput. Secur. 100 (2021) 102078.
[21] H. Huang, W. Kong, S. Zhou, Z. Zheng, S. Guo, A survey of state-of-the-art on blockchains: Theories, modelings, and tools, ACM Comput. Surv. 54 (2) (2021).
[22] S.M. Beillahi, G. Ciocarlie, M. Emmi, C. Enea, Behavioral simulation for smart contracts, in: Proc. of the 41st Conf. on Programming Language Design and Implementation, in: PLDI 2020, 2020, pp. 470–486.
[23] B. Jiang, Y. Liu, W.K. Chan, ContractFuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE Int. Conf. on Automated Software Eng., in: ASE 2018, 2018, pp. 259–269.
[24] C.F. Torres, A.K. Iannillo, A. Gervais, R. State, Towards smart hybrid fuzzing for smart contracts, 2020, ArXiv, abs/2005.12156.
[25] X.L. Yu, O. Al-Bataineh, D. Lo, A. Roychoudhury, Smart contract repair, ACM Trans. Softw. Eng. Methodol. 29 (4) (2020) [Online]. Available: https://doi.org/10.1145/3402450.
[26] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, in: 2019 34th IEEE/ACM Int. Conf. on Automated Software Eng. (ASE), 2019, pp. 1186–1189.
[27] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security, in: CCS '16, 2016, pp. 254–269.
[28] M. Kuzlu, M. Pipattanasomporn, L. Gurses, S. Rahman, Performance analysis of a hyperledger fabric blockchain framework: Throughput, latency and scalability, in: 2019 IEEE Int. Conf. on Blockchain, 2019, pp. 536–540.
[29] M. Schäffer, M. Di Angelo, G. Salzer, Performance and Scalability of Private Ethereum Blockchains, 2019, pp. 103–118.
[30] R.C. Merkle, A digital signature based on a conventional encryption function, in: A Conf. on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology, in: CRYPTO '87, Springer-Verlag, 1987, pp. 369–378.
[31] Y. Xiao, N. Zhang, W. Lou, Y.T. Hou, A survey of distributed consensus protocols for blockchain networks, IEEE Commun. Surv. Tutor. 22 (2) (2020) 1432–1465.
[32] M.J. Amiri, D. Agrawal, A. El Abbadi, Permissioned blockchains: Properties, techniques and applications, in: Proceedings of the 2021 International Conference on Management of Data, Association for Computing Machinery, New York, NY, USA, 2021, pp. 2813–2820, [Online]. Available: https://doi.org/10.1145/3448016.3457539.
[33] O. Dib, K.-L. Brousmiche, A. Durand, E. Thea, E. Ben Hamida, Consortium blockchains: Overview, applications and challenges, Int. J. Adv. Telecommun. (2018) [Online]. Available: https://hal.archives-ouvertes.fr/hal-02271063.

[34] M. Belotti, N. Božić, G. Pujolle, S. Secci, A vademecum on blockchain technologies: When, which, and how, Commun. Surv. Tutor. 21 (4) (2019) 3796–3838, [Online]. Available: https://doi.org/10.1109/COMST.2019.2928178.
[35] V. Buterin, A next-generation smart contract and decentralized application platform, White Paper 3 (2014) 1–36, [Online]. Available: https://cryptorating.eu/whitepapers/Ethereum/Ethereum_white_paper.pdf.
[36] D. Macrinici, C. Cartofeanu, S. Gao, Smart contract applications within blockchain technology: A systematic mapping study, Telemat. Inform. 35 (8) (2018) 2337–2354.
[37] M. Pincheira, E. Donini, R. Giaffreda, M. Vecchio, A blockchain-based approach to enable remote sensing trusted data, in: 2020 IEEE Latin American GRSS & ISPRS Remote Sensing Conference (LAGIRS), 2020, pp. 652–657.
[38] C.D. Clack, V.A. Bakshi, L. Braine, Smart contract templates: foundations, design landscape and research directions, 2016, ArXiv, abs/1608.00771.
[39] P. Zhuang, T. Zamir, H. Liang, Blockchain for cyber security in smart grid: A comprehensive survey, IEEE Trans. Ind. Inf. (2020) 1.
[40] M. Chopade, S. Khan, U. Shaikh, R. Pawar, Digital forensics: Maintaining chain of custody using blockchain, in: 2019 Third International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2019, pp. 744–747.
[41] R.A. Das, M.M.S. Pahalovi, M.N. Yanhaona, Transaction finality through ledger checkpoints, in: 2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS), 2019, pp. 183–192.
[42] Y. Hirai, Defining the ethereum virtual machine for interactive theorem provers, in: M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P.Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, M. Jakobsson (Eds.), Financial Cryptography and Data Security, Springer Int. Publishing, Cham, 2017, pp. 520–535.
[43] E. Seligman, T. Schubert, M.V.A.K. Kumar, Chapter 2 - Basic formal verification algorithms, in: Formal Verification, Morgan Kaufmann, Boston, 2015, pp. 23–47.
[44] J. Yoo, Y. Jung, D. Shin, M. Bae, E. Jee, Formal modeling and verification of a federated Byzantine agreement algorithm for blockchain platforms, in: 2019 IEEE Int. Works. on Blockchain Oriented Software Eng. (IWBOSE), 2019, pp. 11–21.
[45] W.Y. Maung Maung Thin, N. Dong, G. Bai, J.S. Dong, Formal analysis of a proof-of-stake blockchain, in: 2018 23rd Int. Conf. on Eng. of Complex Computer Systems (ICECCS), 2018, pp. 197–200.
[46] B. Beizer, Software Testing Techniques, Dreamtech Press, 2003.
[47] M. Pezzè, M. Young, Software Testing and Analysis: Process, Principles, and Tech, John Wiley & Sons, 2008.
[48] D. Marijan, A. Gotlieb, Software testing for machine learning, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, 2020, pp. 13576–13582, no. 09. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7084.
[49] Z.M. Jiang, A.E. Hassan, A survey on load testing of large-scale software systems, IEEE Trans. Softw. Eng. 41 (11) (2015) 1091–1118.
[50] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678.
[51] A.A. Omar, F.A. Mohammed, A survey of software functional testing methods, SIGSOFT Softw. Eng. Notes 16 (2) (1991) 75–82.
[52] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, Z. Su, A survey on data-flow testing, ACM Comput. Surv. 50 (1) (2017) [Online]. Available: https://doi.org/10.1145/3020266.
[53] C.F. Torres, J. Schütte, R. State, Osiris: Hunting for integer bugs in ethereum smart contracts, in: 34th Annual Computer Security Applications Conf., in: ACSAC '18, 2018, pp. 664–676.
[54] B. Mueller, Smashing ethereum smart contracts for fun and real profit, in: 9th Annual HITB Security Conf. (HITBSecConf). HITB, Amsterdam, Netherlands, 54, 2018.
[55] S. Kalra, S. Goel, M. Dhawan, S. Sharma, ZEUS: Analyzing safety of smart contracts, in: Network and Distributed Systems Security (NDSS) Symposium, 2018.
[56] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 2018 ACM SIGSAC Conf. on Computer and Communications Security, in: CCS '18, 2018, pp. 67–82.
[57] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, Y. Alexandrov, SmartCheck: Static analysis of ethereum smart contracts, in: 2018 IEEE/ACM 1st Int. Workshop on Emerging Trends in Software Eng. for Blockchain (WETSEB), 2018, pp. 9–16.
[58] T. Chen, X. Li, X. Luo, X. Zhang, Under-optimized smart contracts devour your money, in: 2017 IEEE 24th Int. Conf. on Software Analysis, Evolution and ReEng. (SANER), 2017, pp. 442–446.
[59] E. Albert, P. Gordillo, A. Rubio, I. Sergey, Running on fumes-preventing out-of-gas vulnerabilities in ethereum smart contracts using static resource analysis, in: P. Ganty, M. Kaâniche (Eds.), Verif. and Evaluation of Computer and Communication Systems, Springer Int. Publishing, Cham, 2019, pp. 63–78.
[60] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A scalable security analysis framework for smart contracts, 2018, [Online]. Available: https://arxiv.org/abs/1809.03981.
[61] I. Grishchenko, M. Maffei, C. Schneidewind, Ethertrust: Sound static analysis of ethereum bytecode, 2018, [Online]. Available: https://pdfs.semanticscholar.org/26c2/b7e7479336d44891aadda6b5eaae2ca2ee91.pdf.
[62] J. Feist, G. Grieco, A. Groce, Slither: A static analysis framework for smart contracts, in: 2019 IEEE/ACM 2nd Int. Workshop on Emerging Trends in Software Eng. for Blockchain (WETSEB), 2019, pp. 8–15.
[63] S. Wang, C. Zhang, Z. Su, Detecting nondeterministic payment bugs in ethereum smart contracts, Proc. ACM Program. Lang. (2019).
[64] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, Y. Smaragdakis, Ethainter: A smart contract security analyzer for composite vulnerabilities, in: Proceedings of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation, in: PLDI 2020, 2020, pp. 454–469.
[65] A. Ghaleb, K. Pattabiraman, How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection, in: Proc. of the 29th Int. Symposium on Software Testing and Analysis, 2020, pp. 415–427.
[66] P. Daian, Analysis of the dao exploit, 2022, http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/ (June 18, 2016 (accessed on March 01, 2022)).
[67] We got spanked: What we know so far, 2022, https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe (Oct 8, 2018 (accessed on March 01, 2022)).
[68] H. Liu, Z. Yang, Y. Jiang, W. Zhao, J. Sun, Enabling clone detection for ethereum via smart contract birthmarks, in: 2019 IEEE/ACM 27th Int. Conf. on Program Comprehension (ICPC), 2019, pp. 105–115.
[69] W. Yan, J. Gao, Z. Wu, Y. Li, Z. Guan, Q. Li, Z. Chen, EShield: Protect smart contracts against reverse eng. in: Proc. of the 29th Int. Symposium on Software Testing and Analysis, 2020, pp. 553–556.
[70] P. Momeni, Y. Wang, R. Samavi, Machine learning model for smart contracts security analysis, in: 2019 17th Int. Conf. on Privacy, Security and Trust (PST), 2019, pp. 1–6.
[71] A. Hajdu, D. Jovanović, Solc-verify: A modular verifier for solidity smart contracts, in: S. Chakraborty, J.A. Navas (Eds.), Verified Software. Theories, Tools, and Experim., Springer Int. Publishing, Cham, 2020, pp. 161–179.
[72] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, D. Gu, SMARTSHIELD: Automatic smart contract protection made easy, in: 2020 IEEE 27th Int. Conf. on Software Analysis, Evolution and ReEng. (SANER), 2020, pp. 23–34.
[73] W. Zhang, S. Banescu, L. Pasos, S. Stewart, V. Ganesh, MPro: Combining static and symbolic analysis for scalable testing of smart contract, in: 2019 IEEE 30th Int. Symposium on Software Reliability Eng. (ISSRE), 2019, pp. 456–462.
[74] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, M. Vechev, VerX: Safety verification of smart contracts, in: 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1661–1677.
[75] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, prodigal, and suicidal contracts at scale, in: Proc. of the 34th Annual Computer Security Application Conf., 2018, pp. 653–663.
[76] Q. Zhang, Y. Wang, J. Li, S. Ma, EthPloit: From fuzzing to efficient exploit generation against smart contracts, in: 2020 IEEE 27th Int. Conf. on Software Analysis, Evolution and ReEng. (SANER), 2020, pp. 116–126.
[77] X. Mei, I. Ashraf, B. Jiang, W.K. Chan, A fuzz testing service for assuring smart contracts, in: 2019 IEEE 19th Int. Conf. on Software Quality, Reliability and Security Companion (QRS-C), 2019, pp. 544–545.
[78] W.K. Chan, B. Jiang, Fuse: An architecture for smart contract fuzz testing service, in: 2018 25th Asia-Pacific Software Eng. Conf. (APSEC), 2018, pp. 707–708.
[79] S. Akca, A. Rajan, C. Peng, SolAnalyser: A framework for analysing and testing smart contracts, in: 2019 26th Asia-Pacific Software Eng. Conf. (APSEC), 2019, pp. 482–489.
[80] E. Viglianisi, M. Ceccato, P. Tonella, A federated society of bots for smart contract testing, J. Syst. Softw. 168 (2020) 110647.
[81] X. Wang, Z. Xie, J. He, G. Zhao, R. Nie, Basis path coverage criteria for smart contract application testing, in: 2019 Int. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2019, pp. 34–41.
[82] P. Chapman, D. Xu, L. Deng, Y. Xiong, Deviant: A mutation testing tool for solidity smart contracts, in: 2019 IEEE Int. Conf. on Blockchain, 2019, pp. 319–324.

[83] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, Z. Chen, MuSC: A tool for mutation testing of ethereum smart contract, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 1198–1201.

[84] A. Li, J.A. Choi, F. Long, Securing smart contract with runtime validation, in: Proc. of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation, in: PLDI 2020, 2020, pp. 438–453.

[85] G. Grieco, W. Song, A. Cygan, J. Feist, A. Groce, Echidna: Effective, usable, and fast fuzzing for smart contracts, in: Proc. of the 29th Int. Symposium on Software Testing and Analysis, 2020, pp. 557–560.

[86] J. Liao, T. Tsai, C. He, C. Tien, SoliAudit: Smart contract vulnerability assessment based on machine learning and fuzz testing, in: 2019 Sixth Int. Conf. on Internet of Things: Systems, Management and Security, 2019, pp. 458–465.

[87] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, ReGuard: Finding reentrancy bugs in smart contracts, in: 2018 IEEE/ACM 40th Int. Conf. on Software Eng.: Companion (ICSE-Companion), 2018, pp. 65–68.

[88] H. Medeiros, P. Vilain, J. Mylopoulos, H.-A. Jacobsen, SolUnit: A framework for reducing execution time of smart contract unit tests, in: Proceedings of the 29th Annual Int. Conf. on Computer Science and Software Eng., in: CASCON '19, IBM Corp., USA, 2019, pp. 264–273.

[89] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, P. Saxena, Exploiting the laws of order in smart contracts, in: Proceedings of the 28th Int. Symposium on Software Testing and Analysis, in: ISSTA 2019, 2019, pp. 363–373.

[90] Y. Liu, Y. Li, S.-W. Lin, Q. Yan, ModCon: A model-based testing platform for smart contracts, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in: ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1601–1605, [Online]. Available: https://doi.org/10.1145/3368089.3417939.

[91] C. Pacheco, S.K. Lahiri, M.D. Ernst, T. Ball, Feedback-directed random test generation, in: 29th Int. Conf. on Software Eng. (ICSE'07), 2007, pp. 75–84.

[92] C. Lemieux, R. Padhye, K. Sen, D. Song, Perffuzz: Automatically generating pathological inputs, in: Proceedings of the 27th ACM SIGSOFT Int. Symposium on Software Testing and Analysis, in: ISSTA 2018, 2018, pp. 254–265.

[93] M. Utting, A. Pretschner, B. Legeard, A taxonomy of model-based testing approaches, Softw. Test. Verif. Reliab. 22 (5) (2012) 297–312.

[94] T.T.A. Dinh, J. Wang, G. Chen, R. Liu, B.C. Ooi, K.-L. Tan, BLOCKBENCH: A framework for analyzing private blockchains, in: Proc. of the 2017 ACM Int. Conf. on Management of Data, 2017, pp. 1085–1100.

[95] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: Proceedings of the 1st ACM Symposium on Cloud Computing, 2010, pp. 143–154.

[96] M.J. Cahill, U. Röhm, A.D. Fekete, Serializable isolation for snapshot databases, ACM Trans. Database Syst. 34 (4) (2009).

[97] P. Thakkar, S. Nathan, B. Viswanathan, Performance benchmarking and optimizing hyperledger fabric blockchain platform, in: 2018 IEEE 26th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2018, pp. 264–276.

[98] Z. Dong, E. Zheng, Y. Choon, A.Y. Zomaya, DAGBENCH: A performance evaluation framework for DAG distributed ledgers, in: 2019 IEEE 12th Int. Conf. on Cloud Computing (CLOUD), 2019, pp. 264–271.

[99] D. Saingre, T. Ledoux, J.-M. Menaud, BCTMark: a framework for benchmarking blockchain technologies, in: AICCSA 2020 - 17th IEEE/ACS Int. Conf. on Computer Systems and Applications, IEEE, Turkey, 2020, pp. 1–8.

[100] H. Pan, X. Duan, Y. Wu, L. Tseng, A. Boukerche, M. Aloqaily, BBB: A lightweight approach to evaluate private blockchains in clouds, in: 2020 IEEE Global Communications Conf. (GLOBECOM), 2020, pp. 1–6.

[101] E. Heilman, A. Kendler, A. Zohar, S. Goldberg, Eclipse attacks on bitcoin's peer-to-peer network, in: Proceedings of the 24th USENIX Conf. on Security Symposium, in: SEC'15, USENIX Association, USA, 2015, pp. 129–144.

[102] IBM, Blockchain Performance Benchmarking for Hyperledger Besu, Hyperledger Fabric, Ethereum and FISCO BCOS Networks, White Paper, 2018, [Online]. Available: https://hyperledger.github.io/caliper/.

[103] X. Duan, H. Pan, L. Tseng, Y. Wu, BBB: Make benchmarking blockchains configurable and extensible, in: 2019 IEEE 24th Pacific Rim Int. Symposium on Dependable Computing (PRDC), 2019, pp. 61–611.

[104] M.T. Oliveira, G.R. Carrara, N.C. Fernandes, C.V.N. Albuquerque, R.C. Carrano, D.S.V. Medeiros, D.M.F. Mattos, Towards a performance evaluation of private blockchain frameworks using a realistic workload, in: 2019 22nd Conf. on Innovation in Clouds, Internet and Networks and Workshops (ICIN), 2019, pp. 180–187.

[105] P. Zheng, Z. Zheng, X. Luo, X. Chen, X. Liu, A detailed and real-time performance monitoring framework for blockchain systems, in: 2018 IEEE/ACM 40th Int. Conf. on Software Eng.: Software Eng. in Practice Track, 2018, pp. 134–143.

[106] S. Rouhani, R. Deters, Performance analysis of ethereum transactions in private blockchain, in: 2017 8th IEEE Int. Conf. on Software Eng. and Service Science (ICSESS), 2017, pp. 70–74.

[107] C. Fan, H. Khazaei, Y. Chen, P. Musilek, Towards a scalable DAG-based distributed ledger for smart communities, in: 2019 IEEE 5th World Forum on Internet of Things (WF-IoT), 2019, pp. 177–182.

[108] M. Alharby, A. van Moorsel, BlockSim: A simulation framework for blockchain systems, SIGMETRICS Perform. Eval. Rev. 46 (3) (2019) 135–138.

[109] M. Zander, T. Waite, D. Harz, DAGsim: Simulation of DAG-based distributed ledger protocols, SIGMETRICS Perform. Eval. Rev. 46 (3) (2019) 118–121.

[110] M. Bez, G. Fornari, T. Vardanega, The scalability challenge of ethereum: An initial quantitative analysis, in: 2019 IEEE Int. Conf. on Service-Oriented System Eng. (SOSE), 2019, pp. 167–176.

[111] Z. Shi, H. Zhou, Y. Hu, S. Jayachander, C. de Laat, Z. Zhao, Operating permissioned blockchain in clouds: A performance study of hyperledger sawtooth, in: 2019 18th Int. Symposium on Parallel and Distrib. Computing, 2019, pp. 50–57.

[112] T.S. Nguyen, G. Jourjon, M. Potop-Butucaru, K. Thai, Impact of network delays on hyperledger fabric, in: IEEE INFOCOM 2019 - IEEE Conf. on Computer Communications Workshops (INFOCOM WKSHPS), 2019, pp. 222–227.

[113] S. Pandey, G. Ojha, B. Shrestha, R. Kumar, BlockSIM: A practical simulation tool for optimal network design, stability and planning., in: 2019 IEEE Int. Conf. on Blockchain and Cryptocurrency (ICBC), 2019, pp. 133–137.

[114] C. Faria, M. Correia, BlockSim: Blockchain simulator, in: 2019 IEEE Int. Conf. on Blockchain, 2019, pp. 439–446.

[115] S.M. Fattahi, A. Makanju, A. Milani Fard, SIMBA: An efficient simulator for blockchain applications, in: 2020 50th Annual IEEE-IFIP Int. Conf. on Dependable Systems and Networks-Supplemental Volume (DSN-S), 2020, pp. 51–52.

[116] M. Bottone, F. Raimondi, G. Primiero, Multi-agent based simulations of block-free distributed ledgers, in: 2018 32nd Int. Conf. on Advanced Information Networking and Applications Workshops (WAINA), 2018, pp. 585–590.

[117] S. Park, S. Oh, H. Kim, Performance analysis of DAG-based cryptocurrency, in: 2019 IEEE Int. Conf. on Communications Workshops (ICC Workshops), 2019, pp. 1–6.

[118] A. Montresor, M. Jelasity, PeerSim: A scalable P2P simulator, in: 2009 IEEE Ninth International Conference on Peer-To-Peer Computing, 2009, pp. 99–100.

[119] B. Butnaru, F. Dragan, G. Gardarin, J. Manolescu, B. Nguyen, R. pop, N. Preda, L. Yeh, P2PTester: a tool for measuring P2P platform performance, in: 2007 IEEE 23rd International Conference on Data Engineering, 2007, pp. 1501–1502.

[120] R. Yasaweerasinghelage, M. Staples, I. Weber, Predicting latency of blockchain-based systems using architectural modelling and simulation, in: 2017 IEEE Int. Conf. on Software Architecture (ICSA), 2017, pp. 253–256.

[121] C. Rathfelder, B. Klatt, Palladio workbench: A quality-prediction tool for component-based architectures, in: 2011 Ninth Working IEEE/IFIP Conf. on Software Architecture, 2011, pp. 347–350.

[122] S. Ricci, E. Ferreira, D.S. Menasche, A. Ziviani, J.E. Souza, A.B. Vieira, Learning blockchain delays: A queueing theory approach, SIGMETRICS Perform. Eval. Rev. 46 (3) (2019) 122–125, [Online]. Available: https://doi.org/10.1145/3308897.3308952.

[123] W. Zhao, S. Jin, W. Yue, Analysis of the average confirmation time of transactions in a blockchain system, in: T. Phung-Duc, S. Kasahara, S. Wittevrongel (Eds.), Queueing Theory and Network Applications, Springer Int. Publishing, Cham, 2019, pp. 379–388.

[124] P. Ferraro, C. King, R. Shorten, Distributed ledger technology for smart cities, the sharing economy, and social compliance, IEEE Access 6 (2018) 62728–62746.

[125] D. Huang, X. Ma, S. Zhang, Performance analysis of the raft consensus algorithm for private blockchains, IEEE Trans. Syst. Man Cybern.: Syst. 50 (1) (2020) 172–181.

[126] B. Cao, S. Huang, D. Feng, L. Zhang, S. Zhang, M. Peng, Impact of network load on direct acyclic graph based blockchain for internet of things, in: 2019 Int. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2019, pp. 215–218.

[127] P. Yuan, K. Zheng, X. Xiong, K. Zhang, L. Lei, Performance modeling and analysis of a Hyperledger-based system using GSPN, Comput. Commun. 153 (2020) 117–124.
[128] M. Conti, E. Sandeep Kumar, C. Lal, S. Ruj, A survey on security and privacy issues of bitcoin, IEEE Commun. Surv. Tutor. 20 (4) (2018) 3416–3452.
[129] P. A.M., An introduction to the use of zk-SNARKs in blockchains, in: Mathematical Research for Blockchain Economy, Springer Proceedings in Business and Economics, 2020, pp. 233–249.
[130] M. Salimitari, M. Chatterjee, An overview of blockchain and consensus protocols for IoT networks, 2018, CoRR, abs/1809.05613. [Online]. Available: http://arxiv.org/abs/1809.05613.
[131] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, G. Danezis, SoK: Consensus in the age of blockchains, in: Proceedings of the 1st ACM Conf. on Advances in Financial Technologies, in: AFT '19, 2019, pp. 183–198.
[132] S. Artzi, J. Dolby, S.H. Jensen, A. Moller, F. Tip, A framework for automated testing of javascript web applications, in: 2011 33rd Int. Conf. on Software Eng. (ICSE), 2011, pp. 571–580.
[133] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, P.T. Devanbu, To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools, in: 2012 Proceedings of the 27th IEEE/ACM Int. Conf. on Automated Software Eng., 2012, pp. 50–59.
[134] Truffle, 2022, https://trufflesuite.com/docs/index.html (accessed on March 01, 2022)).
[135] H. Dang, T.T.A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, B.C. Ooi, Towards scaling blockchain systems via sharding, in: Proc. of the 2019 Int. Conf. on Management of Data, in: SIGMOD '19, 2019, pp. 123–140.
[136] P. Pritzker, W.E. May, Secure Hash Standard (SHS), National Institute of Standards and Technology, Gaithersburg, MD, USA, 2015.
[137] R. Cavanagh, Federal Information Processing Standard (FIPS) 186-4, Digital Signature Standard; Request for Comments on the NIST Recommended Elliptic Curves, National Institute of Standards and Technology, 2015.
[138] European Union, Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC, Off. J. Eur. Union L 119 (2016) 1–88.
[139] L. Glomann, M. Schmid, N. Kitajewa, Improving the blockchain user experience - an approach to address blockchain mass adoption issues from a human-centred perspective, in: T. Ahram (Ed.), Advances in Artificial Intelligence, Software and Systems Eng., Springer Int. Publishing, Cham, 2020, pp. 608–616.
[140] H. Chen, M. Pendleton, L. Njilla, S. Xu, A survey on ethereum systems security: Vulnerabilities, attacks, and defenses, ACM Comput. Surv. 53 (3) (2020).