IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON NEW ADVANCES IN BLOCKCHAIN-BASED WIRELESS NETWORKS

Received June 15, 2020, accepted July 13, 2020, date of publication August 10, 2020, date of current version August 26, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3013402

Secure Modular Smart Contract Platform for Multi-Tenant 5G Applications

MATEV PUSTI EK, (Member, IEEE), JAN TURK, AND ANDREJ KOS, (Senior Member, IEEE)
Faculty of Electrical Engineering, University of Ljubljana, 1000 Ljubljana, Slovenia

Corresponding author: Matevž Pustišek (matevz.pustisek@fe.uni-lj.si)

The financial support was partially from the Slovenian Research Agency (research core funding No. P2-024, ICT4QoL – Information and Communications Technologies for Quality of Life).

ABSTRACT Trusted decentralized applications based on distributed ledger technologies provide many potential opportunities to 5G applications and verticals, as well as in fifth generation mobile network (5G) enabling technologies, systems, and services. Apart from the tamper-proof exchange of transactions, distributed ledgers can provide a software environment for the trusted execution of smart contracts. In this article, we explore the security aspects of decentralized applications and, in particular, the security of smart contracts. Distributed ledger characteristics impose distinct requirements on smart contract design, implementation, deployment, and management. We briefly present the approach to the development of secure smart contracts, and highlight key smart contract vulnerabilities and the developer tools supporting smart contract security. We developed a secure, upgradeable modular multi-contract platform. It combines per-contract Smart Contract Tunnels and per-user-based access control to minimize vulnerabilities. The smart contract platform is comprised of service-agnostic, auxiliary, and service-specific smart contracts. It can be therefore easily adapted to different 5G application verticals. For illustration and evaluation, we elaborated on the proposed solution on a case of smart electric charging. During the design and development, state-of-the-art code analysis was applied. Finally, we propose an architecture for the integration of the secure multi-contract platform into a 5G architecture. The integration proposal utilizes hybrid private-public blockchain networks for possible security, scalability, performance, and transaction cost optimization.

INDEX TERMS Access control, blockchain, distributed ledger, module, security, smart contract, upgradeable.

I. INTRODUCTION

Emerging 5G networks are characterized by three major features, which promise to support a variety of new services, applications, and business opportunities. These diverse features include greater data bandwidth, full end-to-end latency reduction, reliable communications, and support of machine-type communications for massive Internet of Things (IoT) deployments, including numerous constrained devices connected to the network [1]. The underlying technologies, like Cloud Radio Access Network (Cloud-RAN, C-RAN), Software-defined Networking (SDN), Network Function Virtualization (NFV), and slicing, are enablers for new radio access, the core of the network and new service provisioning principles. At the same time, 5G imposes new security challenges, such as network availability, data immutability, and privacy. The existing security techniques and architectures that are available in 2G-4G networks are not sufficient for 5G. They cannot adequately address issues such as data tampering, dynamics of new technologies and services in 5G networks, or new requirements on security and privacy beyond protecting data integrity [2]. This is becoming additionally challenging with 5G being distinctly multi-tenant and multi-domain, with diverse network participants, including infrastructure providers, network operators, service providers, enterprises, government regulators, and mobile users.

The security associated with 5G technologies has been considered as one of the key requirements related to both 5G and beyond systems. The existing 5G technology infrastructure has remaining unsolved challenges in terms of security, networking, and computing performance degradation due to its centralized architecture [2]. Centralized cloud/edge service provisioning does not guarantee a seamless provision

The associate editor coordinating the review of this manuscript and approving it for publication was Yunchuan Sun.

of Internet of Things (IoT) services. In the scenarios where tenants share the same cloud infrastructure, the possibility of attacks inside the cloud can increase, which damages the transparency and accountability of service providers. End-to-end service function chains may deploy NFVs in an environment involving multiple cloud providers, which could lead to data leakage concerns [3]. The increase in mobile data traffic and the increasing user demands on 5G infrastructure also introduce new challenges in terms of security and performance degradation. Current resource management strategies for spectrum sharing provide a central point of attacks for malicious users [4]. Despite a considerable heterogeneity of the foreseen 5G requirements, technologies, tenants, and use cases, we believe they share a common demand for secure, trusted, autonomous, and automated interactions, which leverage 5G system management, service provisioning, and related applications. This demand can be leveraged/addressed by distributed ledger technologies.

Distributed ledger technologies (DLT) [6] could efficiently contribute to solving some of the discerned challenges in 5G security and architecture. The opportunities brought by blockchains (BC) to 5G networks and services may include security enhancements, system performance improvements, and even network simplification [2]. 5G services require automated, secure, and transparent interactions among multiple stakeholders, trusted intermediaries for newly emerging options, such as infrastructure crowdsourcing, and sharing. At the same time, BC technology adapted in 5G could contribute to services and applications in 5G verticals, too. Management and service capabilities of massive machine-type communications (mMTC) impose new challenges to the scalability and IoT authentication in 5G networks [5].

Distributed ledger technologies and BC [7] as the most prominent example of DLT provide a unique aspect of security – the decentralized trust. In the initial BC protocols (e.g., the Bitcoin), this refers to a trusted exchange of transactions, which are immutably and non-repudiatively stored in a chain of blocks. There is no centralized entity that would provide trust in the system. The trust is achieved with sophisticated cryptographic mechanisms and a public ledger (chain) of past transactions, which is mutually accessible and verifiable by any BC network participant. But the real proliferation of BC technology arose for smart contracts (SC), a feature enabled in various forms in the second generation of BC systems (e.g., Ethereum [8], Hyperledger Project [9], Hedera Hashgraph [10], EOS [11], Corda [12]). These BCs provide a virtual machine that runs in the decentralized nodes of the BC system and enables a trusted execution of programming code – the smart contract. Smart contracts (the backend code, on-chain code) along with the enabling BC network and the corresponding BC-enabled end user and IoT applications (the frontend code) build the fundament of decentralized applications (DApps).

Blockchain properties impose additional considerations and modified approaches in the development of decentralized applications because traditional software engineering principles cannot address some of the specific security challenges of the smart contracts in the decentralized application.

In this article we therefore:

- Analyzed the background security aspects of decentralized applications and smart contracts based on distributed ledger and blockchain technology.
- Proposed a secure, modular multi-smart contract platform, which is upgradeable, enables secure module and user authorization, and is composed of service-agnostic, service-specific, and auxiliary smart contracts. The platform is multi-tenant and can be adapted to various application scenarios or 5G verticals.
- We implemented a prototype of a decentralized smart electric charging service to illustrate and evaluate the proposed approach in a deployment use case for smart electric charging.
- Finally, we proposed several possible approaches to the integration of such a secure decentralized application into the 5G infrastructure, including considerations on hybrid public and private distributed ledger network architectures for secure and scalable operation.

The remainder of this article is organized as follows. In Section II we provide background on the role of BC technologies for 5G networks, services, and applications, and present the concept of decentralized applications. Section III points out key BC security challenges relevant to 5G and elaborates on the security of smart contracts. In Section IV we present our approach to a secure modular smart contract platform for the administration of multi-tenant 5G applications, including the results of a prototype implementation and evaluation. Hybrid public and private distributed ledger network architectures for secure and scalable operation in 5G applications and verticals are discussed in Section V.

II. BACKGROUND

A. BLOCKCHAIN POTENTIALS IN 5G

Potential opportunities of the BC in 5G can be found in 5G applications and verticals as well as in 5G enabling technologies, systems, and services [2]. The decentralized nature of DL applications can eliminate the need for trusted centralized intermediaries in 5G, e.g., for spectrum, band or database management, or centralized cloud and edge service managers. It potentially eliminates single-point failures and provides data and service availability. Immutability of DL records can assist in 5G services, such as spectrum, data, and virtual resource sharing, and can be useful in accounting tasks and usage information for statistics, billing or utilization monitoring. In addition, transparency of these records can mitigate frauds and facilitate better access, verifications, and tracking of transactions for all the involved participants and according to the pre-agreed access rights. BCs can contribute to other unique security features, such as trusted computation, rule execution, and access control with smart contracts.

Cloud computing has a vital role in 5G for optimizing communications, processing, storage of computing and

storage-intensive applications, and data content delivery [4]. Likewise, cloud infrastructure is being integrated into many of the 5G enabling technologies, e.g., in the Cloud-RAN, where a cloud controller manages a large number of small cells. Cloud management in 5G itself imposes challenging requirements on the availability, integrity, immutability, and privacy of data. DLT has already been investigated for cloud management. However, with the extension of cloud services toward the edge of the network and introduction of (decentralized) mobile edge computing (MEC), the decentralized characteristics of DLT become an obvious solution for the security and management of cloud-edge systems [13]. Even wider decentralization in 5G is present when device-to-device (D2D) communication is brought into play. Peer-to-peer based applications for location-based services, secure content sharing, and computational offload between devices can be well addressed by trusted, autonomous machine-to-machine (M2M) communications in the blockchain [2]. Two other 5G underlying technologies, Software-defined networking (SDN) and Network function virtualization (NFV), are characterized by distributed computing paradigms. The BC can ensure decentralization in a trustful manner to enable distributed controllers and distributed network resources in SDN [14]. NFV can spawn over multiple non-trusting administrative domains for inter-domain transactions and billing. Multi-domain orchestration (MdO) imposes complex distributed service level agreements (SLAs) and automated consensus based on chained SLAs [15], which is one of the key objectives for modern decentralized applications.

1) ILLUSTRATIVE USE CASES

We illustrate these possibilities in three use cases, BC-based slice brokering, identification, and authorization for IoT and BC supported smart grid applications in 5G.

Network slicing is an efficient sharing paradigm for advanced mobile network sharing and multi-tenancy. It enables the network operators to recover up to 30% of operational and infrastructure costs in the radio access network, backhaul, and core network. On the other hand, specific application verticals, for example, public protection and disaster relief (PPRD), are keen to replace their costly and frequently outdated existing infrastructure for a modern, efficient and reliable communication service. Common business scenarios for network sharing include sharing of a common radio access or a common core network, enhancing and sharing mobile coverage, and spectrum sharing. A network slice refers to dedicated network capacities, virtual network functions as well as computing and other resources, which are provided by the infrastructure providers to mobile virtual network operators (MVNO), over-the-top (OTT) providers, or different application verticals. Slices can be set statically or dynamically, and resources in a slice are assigned according to communication loads, key performance indicators, and service level agreements. A slice broker [16] provides dynamic and automated admission control for the incoming request (from, e.g., PPRD), and resource assignment via standardized network sharing application programming interfaces (API). The slice broker complements the infrastructure provider's network management. In [17] they propose a 5G network slice broker based on BC technology to reduce service creation time and enable manufacturing equipment autonomously and dynamically acquire the slice needed for more efficient operations.

IoT is an established technology with many successful large-scale production deployments. It is also one of the major 5G domains, and thus a key driver for 5G development, deployment, and use. Blockchain technology is already being adopted in the IoT in various vertical industries, such as the supply chain and logistics for complex management of sensory data, documentation or regulatory compliance, in food traceability, smart health, smart grid, and alike [18]. The integration of the IoT and BC can result in decentralization and scalability, efficient device management and authorization, trustworthiness in data and sharing services, data monetization, autonomous device operation and automation, machine-to-machine transactions, and micropayments [19].

The security and performance of underlying BC technologies and architectures are being improved to meet the requirements of the IoT [20]. At the same time, some of the key challenges of the IoT are being addressed by the BC, too [21]. One area of research is decentralized storage, preprocessing, and mining for decentralized data management. In this aspect, the BC has a strong competitor in cloud-centric solutions, which are more mature and currently provide performance beyond the one in decentralized solutions for the IoT. But for the ownership, identity and access management, and security of IoT devices, DLT and the BC show immense potentials.

Device management and maintenance of the numerous inexpensive and resource-constrained devices imply a strong potential for security vulnerabilities in the IoT. Device management is characterized by massive deployments of simple unmanned devices. The device maintenance challenge is how to keep the security credentials, firmware, and software properly configured and up to date in these constrained devices. From the security perspective, the BC can address two major issues in the IoT, identification and authentication. Lightweight, yet reliable device authentication is required, along with a convenient way of flexible governance for the manufacturers and owners to track and maintain device identities [22]. A study [23] explores how devices' identities are authenticated and how the consequent data and service access is managed. A semi-decentralized BC-based framework is proposed to provide features of identity creation and transfer of ownership. Similarly, research on the means of identification and authorization of IoT devices with the BC [24] explores and proposes an authenticated device configuration protocol to achieve this authentication process.

A smart grid enables new and innovative use cases in the utility domain, which are technically infeasible in the traditional electric grid. Numerous sensors, actuators, and meters are dispersed over large territories, and 5G con-

nectivity makes smart grid one key 5G verticals. DLT with autonomous machine-to-machine negotiations, micropayments, and utility tokens can support advance smart grid services for microgrids, management of distributed resources (DER) and storage, smart electric vehicle (EV) charging, demand and supply aggregation, and management. Architectural considerations of BC implementation for the IoT and smart grid are addressed in [25], [26]. Some studies point out the need to distribute a part of the BC functionality toward the edge of the communication network [26], and some investigate the possibilities of applying the BC in the SG [27], too. In [28] they focus on the role of the BC in mobile charging of electrical vehicles. A blockchain-based PEV charging system is presented in [29], where energy can be traded directly from one vehicle to another (P2P mode). We present a BC-based smart grid prototype in the introduction of Section IV. The practical part of our research is based on it.

B. DECENTRALIZED APPLICATIONS

The term decentralized application (DApp) refers to an application that is executed by multiple users over a decentralized network. [30]. A DApp implementation framework consists of three key building elements, which we dubbed a DApp triplet. These elements are a trusted decentralized ledger, trusted decentralized execution of program logic, and user applications [18]. An example of a DApp triplet is a BC network, smart contracts implemented in this network, and BC-enabled front-end applications, which provide user interfaces and run embedded IoT devices.

Possible application scenarios impose desirable characteristics on a blockchain platform. The characteristics, which are not narrowed down to only the performance of BC networks in terms of transaction latency, throughput and sequential performance, are strongly shaped by the nature of the chain-structured distributed ledger and consensus mechanisms built in the network. Although the type and configuration of a BC network radically affect these parameters, the performance is still not comparable with the renowned and alternative non-BC transaction processing systems. Another expectation is the management of offline transactions for predictable behavior and security implications on the services that are provided by DApp when connectivity to the BC network is not guaranteed. In public BC networks with cryptocurrencies, reasonable monetary costs can become a problem. They include the transaction processing fees as well as the ability of DApp developers to leverage other models apart from the transaction cost-based ones. Despite the immutability of the distributed ledger, BC platforms need to provide flexible maintainability, which includes upgrades of the BC protocols and management of the readily deployed SC code for bug recovery and updates. And lastly, when required, the privacy in BC networks needs to be addressed, and efficient identity management is expected, especially for the IoT DApps, where unmanned devices act as BC participants [18].

1) DAPP TRIPLET: DISTRIBUTED LEDGER NETWORK FOR DECENTRALIZED APPLICATIONS

A blockchain network is a set of peer nodes, which interconnect in a decentralized manner through an IP network. The nodes, also named clients, are responsible for running the BC protocols and thus all communication with the rest of the BC network. This includes managing peers and the network, monitoring the status and synchronizing the chain of blocks (to keep the local chain up to date), managing the accounts and transactions (e.g., creating and sending outgoing transactions), listening to events, mining blocks, and enabling a decentralized virtual machine where smart contracts can be deployed and executed. A BC node usually exposes a Web-based API through which the off-chain applications use the BC capabilities.

Mining is not a mandatory function of every BC node. However, it is frequently implemented in the common node software and can be arbitrarily enabled. When a node creates a new BC transaction (Tx), it populates the requested Tx data structure and signs it with the private encryption key. The key belongs to the originating BC account and thus the authenticity of the transaction is assured. Miners – i.e., mining nodes – receive newly created transactions from the other nodes in the network. They validate the incoming transactions in terms of authenticity, and verify the account balance. A subset of valid transactions in the transaction pool is used to compose a new block to be included in the chain. The process of building new possible blocks is running concurrently in many decentralized miners. Therefore, miners execute a consensus algorithm to select which of the miners will append a new block to the chain.

In BC network implementations various approaches exist and the selected approach can affect the performance, scalability, security, and privacy of the network. The approaches differ in how new nodes can be added to the network and which nodes are allowed to act as miners. The latter can be to some extent related to the consensus algorithm applied in the specific BC network [18]. In public BC networks (e.g., Bitcoin network [31]; Ethereum mainnet – a public instance of the Ethereum network; Hedera mainnet [32]), anyone can propose a new BC node, including a mining node. There is no authentication of the node proprietor and there is no explicit guarantee that all the nodes behave honestly in creating new blocks for the chain. Anyone can access the ledger and transactions in the ledger. A large number of mining nodes and the decentralized nature of the consensus algorithms make sure that malicious node behaviors are detected and the blockchain remains uncompromised. In a private BC network (e.g., Hyperledger Fabric [33]), nodes may run the same BC protocols as the public BCs and still operate in a distributed manner, but the mining nodes are under the control of a single entity. This might, on the one hand, have a positive impact on the transaction throughput, latency and privacy of the BC data, but on the other hand to a large extent inhibits the trust of the system. Private BCs are no longer decentralized but merely distributed. The chain

owner can decide who can access and write chain data and nodes to the network. Consortium BC platforms (e.g., Hyperledger Fabric) have many of the same advantages as a private blockchain, but operate under the leadership of a group (e.g., a group of collaborating organizations) instead of a single entity. Thus, they combine elements from public and private networks. Unlike in private BCs, where only a single entity appoints nodes and miners, a consortium chain builds on equally powerful parties that function as validators. Access to the ledger can be limited to validators readable to authorized individuals, or granted to all. A consortium blockchain is commonly applied in a setting where multiple organizations operate in the same industry and require a common ground on which to carry out transactions or relay information [34]. Based on how the data are managed and accessed in the blockchain system, we can classify the blockchain into two types, a permissionless and a permissioned blockchain. In a permissionless blockchain, a participant can join the network without any approval or registration, and all the nodes can initiate or verify transactions. On the other hand, access to the network is permissioned, and the operations needed to perform or verify transactions are under administrator control, too, in a permissioned blockchain [21]. Public BC networks are permissionless, whereas private and consortium networks are permissioned and have only known participants.

Apart from a trusted environment to exchange and keep a record of mutual transactions, running the DApps requires the capability to execute on-chain smart contract logic. This feature, if provided by the BC nodes and the network, results in a distributed and decentralized virtual machine (VM). It enables the execution of the SC code that has been previously deployed (''installed'') in the BC network, where the code is guaranteed to produce the same results for everyone who runs it. The SC VM thus introduces trust in the code execution. The VM makes it possible to run DApps that have no central points of failure or control, integrate with a payment network, and operate on a private, public, or consensus-based BC network. An example of such a VM is an Ethereum Virtual Machine (EVM) [35], [36]. In Hyperledger Fabric, smart contracts are packaged as chaincode. Chaincode is installed on peer nodes and then defined and used on one or more channels, which are private blockchain overlays that allow for data isolation and confidentiality [33], [37]. Details on smart contract development, deployment, and execution are presented in the following subsection.

2) DAPP TRIPLET: SMART CONTRACTS FOR DECENTRALIZED APPLICATIONS

The term ''Smart Contract'' was coined by Nick Szabo in the '90s. In the BC, an SC is programming code, deployed and executed in the BC network. It has no central point of failure, it can perform operations, hold value and unlock it only if specific conditions are met [33], [35]. The benefits of smart contracts are most apparent in business collaborations, in which they are typically used to enforce some type of agreement so that all participants can be certain of the outcome without an intermediary's involvement [38]. Various programming languages are applied in the development of SCs, depending on the VM in the selected network. Some VMs support diverse programming languages for SCs [37], others stick to one. The same VM can be supported in different BC protocols (e.g., EVM in Ethereum, Hyperledger Burrow, etc.). If the scripting language used by the VM is Turing-complete, this essentially means that the types of DApps users can design are limited only by their programming skills and creativity. This is the case in Ethereum, too. The VM limits the scope of the SC execution to the BC network. An SC can call methods of other SCs in the network, but cannot access external programming interfaces, like Web APIs. External applications call SC's methods to read the SC parameters, or send BC transactions to the SC to change its state. An SC can emit events, which are received by the BC network nodes and can be passed to the external applications by a node. In Ethereum, the use of a deployed SC is defined by an Application Binary Interface Specification (ABI) [39]. An ABI is the interface between external program modules and the SC bytecode. In the BC, the interface is the de facto method for encoding/decoding data into/out of the SC, i.e., how you call functions in a contract or pass and get data back. Slightly different approaches are taken in other BC architectures, e.g., Hyperledger [37].

A smart contract lifecycle is comprised of several stages: the development and verification, deployment, use, and maintenance. In Ethereum, the SCs are primarily developed in the Solidity (or Vyper) programming language. Solidity is supported by various dedicated (Truffle, Remix) and common (e.g., VS Code) development environments. During the development and after the deployment, an SC should be verified with formal and/or runtime methods to verify the correctness of the SC code and minimize possible security flaws in the specification, code implementation or misunderstandings, mistakes, and bugs in VM implementation. Typical approaches may include static analysis, symbolic execution, or input fuzzing to detect security bugs and verify the correctness of smart contract code. Once the source code is completed, it is compiled to the Ethereum-specific binary format, called the EVM bytecode. The bytecode, along with the Application Binary Interface Specification (ABI) file, is then sent off to the blockchain with a contract creation transaction. This is a special transaction that is sent to an empty BC address, with the EVM bytecode as data. Once the creation transaction is added to the chain by validating nodes in the BC, the SC obtains a unique SC address. An SC has its balance, some code, and some persistent storage for the execution of its operations.

Each node in the BC network (not just miners) executes the SC. This ensures the consensus and thus trust in the results of the SC execution. The SC methods can be accessed in two ways by the BC network participants. If an external application requires, e.g., a simple read of an SC parameter, this is done with a local invocation (occurring only in the local BC node) of the SC method call via the node API. On the other hand, when a BC network participant needs to modify

some state in the SC, the call has to be initialized through a transaction sent to the SC address. This transaction can include a cryptocurrency for the SC, as well as additional input parameters for the method call. Transactions have to be validated and included in the chain in the same way as non-SC transactions. Upon the inclusion, the SC is executed and the state is updated accordingly. The SC's bytecode, of course, remains unchanged. Any code changes of the deployed bytecode are impossible due to the BC's immutable nature.

3) DAPP TRIPLET: SOFTWARE ARCHITECTURE

Most of the current blockchain-based applications are still limited to utilizing smart contracts for the core data and backend functionality. Smart contract users run their Web, mobile, or embedded programs off-chain to complete the application [18]. This might not be yet the ultimate goal of decentralization, where a DApp is hosted entirely by a BC system. There are some solutions readily available to support user applications in a decentralized manner, for example, the Ethereum Name Service (ENS) or decentralized file transfer systems. The ENS is a distributed, open, and extensible naming system based on the Ethereum blockchain. It offers a secure and decentralized way to address resources both on and off the blockchain, using simple, human-readable names. Many applications are emerging that support this way of naming resources [40]. BC-based decentralized file systems make it possible to distribute data without having to host any kind of server, as the nodes download (or stream) files from hundreds of nodes simultaneously. Efficient decentralized storage layers for the future of the next generation internet are provided by, e.g., Swarm and IPFS. Both the high-level goals and technology used in Swarm and IPFS are very similar. They aspire to provide a generic decentralized distributed storage solution and a content delivery protocol [41].

However, not every part of a DApp can reside in the BC because on-chain SC applications have to manipulate actual devices (e.g., IoT), or face real-word (off-chain) data sources. The BC-aware embedded applications in an IoT device connect the BC backend to the physical environment. The oracles [42] serve as intermediaries, providing data feeds along with an authenticity proof to the smart contracts and the BC from/to external software (e.g., web sites) or hardware entities.

Currently, the DApp design principle is to put into an SC only the part of the application logic that should be resistant to modifications and rely on the proven Web-related technologies for the remainder of the application. As presented later in Section II.B, this is, in fact, positive from the security perspective. But the given approach also alleviates some of the performance bottlenecks of state-of-the-art blockchain systems.

A BC provides computation and storage capabilities via the mechanism of smart contracts. In the relation between the SC and off-chain applications in a DApp triplet, there are three kinds of architectures seen in practice: direct, indirect, and mixed. In the direct architecture, the client directly interacts with smart contracts deployed on the BC network. DApps of the indirect architecture have back-end services running on a centralized server, and the client interacts with smart contracts through the server. DApps of the mixed architecture combine the preceding two architectures, where the client interacts with smart contracts both directly and indirectly [30]. In the Ethereum based DApps, the indirect architecture is usually applied for the constrained IoT devices [43]. These constraints include limited computation, storage, or communication resources. Interestingly, the same approach is frequently taken in mobile BC-based applications, too. For the latter, this is mostly to increase the security and to ensure better user experience in terms of application's instant availability, despite the intermittent connectivity.

4) OPEN CHALLENGES

There are several challenging aspects in the selection and adoption of the appropriate DL technologies for decentralized applications in 5G and elsewhere. These are by far not only the technical characteristics of the underlying technologies, where often potentially promising new features in DLT are discussed. For a sustainable business solution development, a proven, reliable, and future-oriented DApp ecosystem is at least of equal importance. This, along with a thorough design of the DLT role in the overall solution, is the fundament for success. Nevertheless, the exact definition of the decentralization triplet is a challenge, too.

For the DL network, we need to consider mutual impacts on the performance, security (trust), and costs of applying such a network. We select among the readily available public networks or a private one. With public or consortium-based networks meant for public use, we are limited to the given network performance and transaction costs, but we benefit from considerable trust and public cryptocurrencies. If the latter is not required in our envisaged solution, then a private or consortium-based network offers many possibilities to trim the network operation for the desired performance, but we need to be aware of the consequent risks related to (at least partially) the centralization of the network.

The off-chain application development, in particular for the IoT, has to consider the limited resources of the device and possible physical threats to a compromised device. Especially in unmanned devices, the account access keys and passwords might be exposed if a device is, e.g., stolen. To meet the communication and computation constraints, architectural modifications might need to be taken. The IoT device can then perform only a subset of actions needed for the participation in the DL communications. For example, a device might create and sign a transaction but rely on an external node's API for the entire DL communications. Similarly, it could only create a transaction, which would be signed and submitted to the DL network by an intermediary node. These architectural variations can have significant implications for security and trust.

The SC lifecycle imposes an additional challenge in DApp software design and development. In a truly

decentralized and trusted DL network, any changes in a previously deployed SC code should be practically impossible. This, in fact, presents a relevant security issue, which has to be thoroughly addressed during the SC development. Any fallback methods need to consider that the SC bytecode cannot be changed anymore, even if severe mistakes were found after the deployment to the BC network. The aspects of SC security are presented in more detail in Section III.A.

III. SECURITY IN DECENTRALIZED APPLICATIONS AND DISTRIBUTED LEDGER TECHNOLOGIES

DLT brings several new dimensions to application design with their inherent trust and security. However, it is not immune to security threats. Some of the risks share patterns already known in cybersecurity, whereas others are specific for DLT. Among unspecific attack vectors, there are attacks on application clients and user wallets, and mining malware installed into target computers. The known attacks on user wallet credentials typically occur through phishing, dictionary attacks, through the exploits of a digital signature, hash function, and address vulnerabilities [44], or by exploiting bugs in hardware wallets [45]. Distributed denial of service (DDoS) attacks have targeted off-chain and auxiliary services, such as crypto exchanges and mining pools.

Possible attacks and new attack vectors comprise the governance and operation of the DL network, mining and consensus, tampering of transactions, and attacks on smart contracts. Various vectors are frequently combined into one attack area, with the objective to first gain control over the network, a part of the network or individual nodes, and then to exploit this control to manipulate transactions. Some of the attack vectors might be extremely hard and costly to execute. In truly decentralized DL networks they might be practically almost infeasible. However, in more centralized (private) networks, the risks remain real. Many of the attack vectors are protocol and implementation specific, and are not present in every variant of DLT. So the actual implementation of a DL network may mitigate or prevent some of the risks with, e.g., high decentralization, authorized nodes instead of anonymous ones, consensus mechanisms other than traditional PoW, etc.

DL network attacks are usually the basis for further attacks. They target the governance or operation of a DL network, with the ambition of taking the control of the majority of nodes and thus the entire ledger (e.g., 51% attacks). They might also try to partition the DL network into several separate and inconsistent partitions, or to isolate a selected node from the remaining trusted nodes (Eclipse attack, Sybil). In the Eclipse attack, they introduce a large number of node IPs, possibly with a distributed botnet. All the victim's outgoing connections are redirected to these IPs, and the victim is unable to obtain transactions. In the Sybil attack, they assign several identifiers to the same (malicious) node. This is then followed by presenting the node under attack with a deceptive state of the ledger, often through manipulating time perception of the node under attack (e.g., Timejacking, delay attack). The attacked node can then be subjected to transaction tampering by delaying a pre-mind block with the first transaction to invalidate the second, identical transaction in the next block (Finney attack).

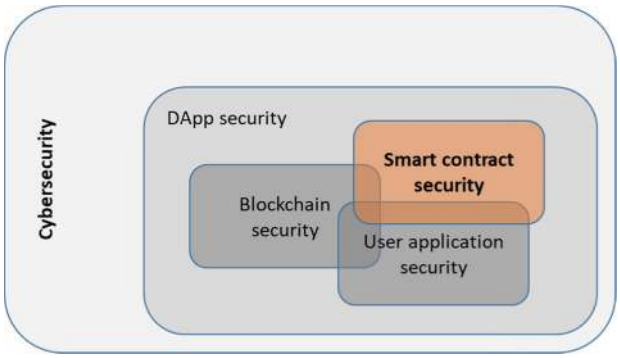

FIGURE 1. Smart contract security in the context of cybersecurity. Security aspects of decentralized applications are a part of a broader cybersecurity landscape. In DApp security, in addition to common cyber risks and vulnerabilities, specific DL network and specific smart contract security-related challenges appear.

A transaction can be tampered by exploiting propagation and block delays due to the consensus mechanism, too. This exploits the intermediate time between the initiation and confirmation of a transaction to invalidate some previous transactions, and to enable double-spending. In the Race attack, one creates two conflicting transactions so that the second makes the first invalid. Transaction malleability attack intends to trick the victim into paying twice, by broadcasting a second transaction with a changed hash to the network.

Mining nodes and consensus mechanisms are vulnerable, too. A mining node can be dishonest and behave selfishly, and withhold some of the already mined blocks (e.g., Selfish mining, Block withholding). In this way, the miners attempt to increase their share of the reward. Effective countermeasures include the application of block timers or increasing the randomness of miner pool assignments.

The immutable nature of a previously deployed SC code exposes vulnerabilities that could be easily mitigated in traditional software systems but can present severe risks in decentralized applications. As any source code, even a thoroughly tested and validated smart contract might have bugs. There can be bugs in the SC execution environment, the EVM, and alike. Unlike the SC code, these can be fixed with DL node updates. However, the time of updating a decentralized communication system run by independent node owners might be longer than in centralized ones. Some of the known Ethereum attacks include Integer overflow and underflow, denial of SC services, Timestamp dependence, where the timestamp of the block can be manipulated by the miner, Reentrancy, which can occur across multiple functions and even multiple contracts, or Displacement, Insertion, Suppression front-running attacks [46]. Smart contract security is thus a specific, yet an integral part of the broader DLT security.

Security aspects of decentralized applications are a part of a broader cybersecurity landscape, as depicted in Fig. 1. For secure decentralized applications, security aspects of all three DApp triplet building elements are relevant. These aspects can overlap to some extent (e.g., vulnerabilities have common roots and similar mitigations can be applied). But DApp security, in addition to common cyber risks and vulnerabilities, introduces specific DL network and specific smart contract security related challenges. In this article, we focus on smart contract security.

A. SMART CONTRACT SECURITY

There are tens [47] of decentralized application platforms using different SC languages and having different DLT characteristics, which makes them difficult to compare. Some vulnerabilities arise from specific SC languages and may appear only in the corresponding platform. Others arise from specific features of the specific platform, such as transaction gas and fees in Ethereum, and order dependence in Solidity. Vulnerabilities from a misunderstanding of common practices may appear on most DL platforms. But in the two most relevant ones we can observe similar vulnerability aspects. In Ethereum, smart contract vulnerabilities arise from the Solidity language, the Ethereum or EVM blockchain platform, and a misunderstanding of common practices. In Hyperledger Fabric chaincodes, vulnerabilities arise from almost identical points: the Go language, the blockchain platform, and a misunderstanding of common practices [48].

The trusted environment provided by a DL network may enable smart contracts. However, the same underlying decentralized mechanisms present a very specific software environment in terms of SC security design, implementation, testing, and monitoring. In most of the DL networks, SC transactions and data may be visible to an adversary. Public functions are public in the broadest sense, and this exposure makes smart contract vulnerability easy to exploit. Similarly, the privacy of data is not granted, and network participants might be anonymous or pseudo-anonymous. These aspects might be reduced in authorized private or consortium-based networks, but cannot be fully eliminated. In the Ethereum network, additional consensus and computation related challenges are known, for example, imprecise block and transaction timestamps, potential miner interference to transaction execution, non-trivial and possibly unreliable random generation, overflows and underflows in mathematical calculations, and alike [27]. An additional challenge is the immaturity of blockchain platforms and smart contract languages. Blockchain technologies are evolving fast, and the DApp developers are confronted with changing platform features. Frequently, there is a substantial mismatch between planned and promised features and their actual delivery. Therefore, design flaws may exist in blockchain platforms or smart contract languages. Common software security weaknesses [26], [49] may be amplified on blockchain platforms and in the related SC security [50]. Such weaknesses include improper behavioral workflow, access control or initialization, incorrect calculation and insufficiently random values, the inclusion of untrusted external functionalities (e.g., external libraries, smart contracts deployed by others), and improper exception handling and cryptographic understanding [48].

A recent study [30] revealed interesting smart contract usage patterns in a large set of Ethereum based DApps. About 75% of them are single contract, meaning that the on-chain application logic consists of only one smart contract. The remaining were multi-contract, on average comprised of three SCs. This indicates that modularization of SC design and development is not very common. This limits the efficient reuse of SC source code, which is mostly being simply duplicated among different implementations [46]. The same research investigates the openness of the DApp and SC source code. In most cases, the DApp and SC source code have mixed access, where key parts of the software are not fully open. Only 16% of DApps were found to be fully open, whereas 25% of them are fully closed and so are 52% of SCs. The overall SC security is affected by the importance of good external data, where an SC needs additional data feeds from traditional on-line resources, independent from the user data in the transactions for SC calls. Most previous researches surveyed smart contract security only in Ethereum. However, much of it has a broader impact and is not limited to Ethereum, especially where security risks occur because of the misunderstanding of common practices.

B. SC VULNERABILITIES COVERAGE

Key SC vulnerabilities stem from logical flows, bugs in the SC code, and under-optimized code patterns. Logical flows in the implementation of the contract, where, for example, an SC does not refund the initial deposit, lack of cryptography for user inputs to ensure fairness, or incentive misalignments [51], may be deliberate or caused by a misunderstanding of DL properties. In public DL networks, where cryptocurrencies and miner awards are part of the DApp execution, there is a risk of under-optimized SC code patterns that lead to unnecessary gas consumption [52]. Their research states that the two main reasons for this are useless-code related and loop-related patterns. The result of scanning 4,240 smart contracts shows that at least 80% of contracts suffer from one or more of such patterns. However, most of the SC vulnerability detection work is focused on the SC code and flaws specific for the DL environment. The Smart Contract Weakness Classification and Test Cases registry (SWC Registry) [50] is a comprehensive list of key security flaws in the Solidity SC code. It provides a checklist for SC programmers as well as for many of the SC vulnerability detection tools.

Some of the common Solidity SC vulnerabilities [51], [53] include transaction-ordering dependence, timestamp dependence, mishandled exceptions, and reentrancy. Transaction-ordering dependence (SWC114) can be a problem if users have no control over the order of transaction execution. The order is up to miners. If there is more than one transaction that invoked the same contract, the order of those transactions can affect the new state of the blockchain. Timestamp

Dependence is related to smart contracts, which include conditions that are triggered by the block timestamp. Block timestamps are set by miners based on their local system time, and are therefore unreliable, or can be manipulated by an adversary. Mishandled exceptions (SWC104) target the contract that calls another contract. If any exception occurs in the called contract, it terminates and returns false, but it may not notify the caller contract. Reentrancy vulnerability (SWC107) is present when a contract calls another contract, and the current contract execution waits until the called contract finishes. This provides an opportunity for the adversary to exploit the intermediary state of the caller contract and call its methods several times. Authorization through tx.origin (SWC115) could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.

In Hyperledger Fabric chaincode development, Go is most widely used. Most security vulnerabilities in Hyperledger Fabric chaincodes arise from the nondeterministic behavior of Go, which may lead to consensus failure [48]. Apart from this, other vulnerabilities arise from almost identical points as in Ethereum: the blockchain platform and a misunderstanding of common practices.

C. ANALYSIS AND VERIFICATION TOOLS

Several projects are aiming at formal verification of smart contracts and a comprehensive list is being maintained at [54]. Some of these tools operate at the source code level (Solidity), whereas others at the compiled bytecode level.

Mythril [55] is a free and open-source smart contract security analyzer for the EVM bytecode. It detects security vulnerabilities in smart contracts that are built for EVM-compatible blockchains (e.g., Ethereum, Hedera, Quorum). It uses symbolic execution, SMT solving, and taint analysis to detect a variety of security vulnerabilities. It is also used in combination with other tools and techniques in the MythX security analysis platform.

MythX [56] is a cloud-based smart contract security service. It performs security analysis remotely, accepting jobs, and returning results via an API. A free version is available, but for a complete analysis, you must sign up and purchase a subscription. It currently detects most weaknesses found in the SWC Registry, and thus covers assertions and property checking, byte-code safety, authorization controls, control flow, the correctness of ERC standard implementations, and various coding best practices for Solidity [57]. The analysis types include symbolic analysis, fuzzing (bytecode), Solidity code analysis, taint analysis, and static analysis. The MythX API has been integrated into many development frameworks, including Brownie, Truffle, and Remix.

OYENTE [58] is another tool to analyze the Ethereum smart contracts code based on symbolic execution. OYENTE takes two inputs, the Ethereum smart contract bytecode, and Ethereum global state. It checks a contract against Transaction-Ordering Dependence (SWC114), Timestamp Dependence (SWC116), Mishandled Exceptions (SWC104), and Reentrancy Vulnerability (SWC107).

SmartCheck [59], [60] is a tool that translates the contract source code (Solidity code) to the XML format, and looks for problematic patterns using XPath queries. The knowledge base [61] includes several known security, functional, operational, and development issues.

Securify [62] is a fully automated online static analyzer for smart contracts, providing a security report based on vulnerability patterns. The project was funded by an Ethereum Foundation grant and created by ICE center, ETH Zurich, and ChainSecurity AG. Securify first decompiles the EVM bytecode of a smart contract, extracts data and control flow dependencies of the contract, and checks the security pattern represented in the Domain-specific language (DSL). The observed patterns are classified as compliant or violating [63].

GASPER [52] is focusing on the identification of gas-costly programming patterns in Solidity. In particular, they identify 7 gas-costly patterns and group them into 2 categories, i.e., useless-code related, and loop-related patterns.

Chaincode scanner [64] is a static security checker for Hyperledger Fabric Smart Contracts. It takes a chaincode written in Go as input, and checks it for nine vulnerability patterns.

With Etherscan [65], one can explore and search for transactions, addresses, tokens, prices, and other activities taking place on Ethereum (ETH) public networks. Etherscan also supports smart contract source code verification, which is done after the deployment. The verification provides proof that the supplied SC source code matches the bytecode deployed in the network. This increases the transparency of the deployed SCs if developers decide to make code public. This is a popular checking tool but limited by the insufficient support for the passing of additional arguments to the constructor of a contract.

D. APPROACHES TO SMART CONTRACT SECURITY ASSURANCE

According to [48], smart contract security should be explored from a global perspective and reaching beyond merely analyzing the smart contract code for possible vulnerabilities and flaws. They refer to four development phases, which all contribute to SC security: i. security design, ii. security implementation, iii. testing before deployment, and iv. monitoring and analysis. These phases are similar to a traditional software lifecycle.

The design includes secure design principles and patterns. ConsenSys elaborated good design practices for Solidity SCs [46]. The practices strive for the simplicity of the SC logic, modularization, and reuse. This facilitates more efficient mechanisms to upgrade an already implemented SC in a controlled manner. Several approaches can be taken or auxiliary services used to do this, e.g., a registry contract, delegate calls, or using the Ethereum Name Service. In addition, they

advise to include mechanisms to pause, freeze or delay smart contract actions if relevant flaws are found after deployment, as well as mechanisms to migrate and transfer data and funds to the updated version of the SC. Modularization implies the multi-contract DApp architecture, which is still not a common practice (see Section III.A). For DApps with multiple SCs, there are three usage patterns of smart contracts [48]. In **leader-member,** one SC is the leader and is responsible for user interactions, and the remaining sub-contracts are invoked through internal transactions. In the **equivalent** architecture, all the SCs are at an equivalent level and there is no invocation between smart contracts. **Factory** enables the simple creation of new child contracts, which are deployed by a (parent) factory contract.

An approach to upgradeable modular contracts has been recently proposed as the Ethereum Improvement Proposal (EIP) 2535 [66]. This draft document proposes a proxy contract that supports using multiple logic contracts. These delegate contracts are called facets, and each facet supplies one or more functions. A dedicated function enables adding, replacing, or removing functions. Events are emitted upon changes in diamond functions, and a user can verify what version of the function is called. With this design approach, one can develop and incrementally improve the SC logic of a DApp over time. However, the design of SCs ownership, authentication, and authorization is not a part of this draft.

Secure implementation follows secure design principles. Security libraries offer instant provisioning of several relevant security features. OpenZepplin [67] provides role-based access control and secure mathematical library for SC programming for Ethereum. It helps minimize risk by using well-tested libraries. It includes the most used implementations of ERC standards. However, the application of external libraries might contradict the desired simplicity of the SC code, as well as it may introduce possible new risks with additional code in the library. Secure development also includes a well-defined upgrade mechanism and systematic applications of suggested fixes for the known SWC vulnerabilities. Another approach is security templates that connect legal agreements to the executable code, and can then generate standardized SC code.

Several kinds of testing can be made before deployment of an SC to a DL network. This reduces the need for a rather complicated updating of smart contracts if bugs are found after the deployment. Testing methods include code analysis (for a brief overview of some of the tools see Section III.C). Some of the code analyzers are integrated into the SC development tools, but most of them are still maturing. Many cover only a limited scope of SC vulnerabilities, and some need to considerably improve the accuracy of detection. Formal verification is a mathematical approach to verify the correctness and security of a program. The approach to formal verification for Ethereum is well defined and relies on the formalization of the EVM and SC code with some general-purpose intermediate language (e.g., Lem, Scilla), and the proof is done on the intermediate language. DL ecosystems that use general-purpose programming languages for SCs are more complicated for formalization. Lastly, security audits are provided as a service by dedicated companies. They combine automated code and manual analysis. Audits can be costly, but with substantial financial consequences in many DApps, they might be necessary to ensure the security and correctness of the code.

Finally, even once the SC has been deployed and is being used, measures like bug bounty programs, security runtime monitoring, and DL data analysis can be taken. Blockd Build [68] is Blockd's innovative approach to active smart contract security. It combines real-time monitoring of incoming transactions at the DL network level (in the transaction pool of a mining node), and can actively protect a smart contract from accepting calls from unregistered or malicious transactions. The DL data analysis applies modern big data and machine learning techniques to the vast amounts of the chain data. It discovers possible attacks and irregular activities in the DL network, such as crypto money laundering attempts.

IV. SECURE MODULAR SMART CONTRACT PLATFORM

We based the practical part of our research of the secure, modular, multi-contract, multi-service, and multi-tenant SC platform on a smart electric grid and charging service. A part of our previous research [69] is a prototype of an end-to-end solution, based on an Ethereum BC-controlled IoT electric switch and meter, dubbed Swether. This DApp is comprised of the hardware and software for the IoT device, along with the smart contract and Ethereum-compliant Web applications for the use and control of the system. This prototype has been later applied in various research projects related to requirements and architecture for IoT BC [70] and BC edge services [71], and in proof-of-concept demonstrations [72].

The described implementation of Swether is a single contract solution, focusing predominantly on client services (i.e., electric charging and metering), and BC network communications. Although smart contract security was not our key objective at that time, and the development approach of the former version was not secure by design, basic security principles were regarded there, too. The code check with Mythril detected no issues.

We are running the system in the Ropsten public Ethereum network and in various Ethereum flavored private networks. The Swether device can be built into, e.g., an electric car charging station, and acts as an independent BC node, reporting the measurement status into the chain. The SC in the DApp enables booking of a charging station for a selected period, tracks the availability of charging stations, and handles the charging payments. In a typical user scenario [69], a client would book a plug in a Swether device for the desired time period or energy quantity, and confirm the booking with a transaction to the Ethereum network sent from a BC-enabled web browser or mobile application. The smart contract would validate the request and launch the events to the blockchain. The events would be intercepted by the Swether device. The smart contract enables additional payment and workflow features, such as escrow (to increase the security

of involved stakeholders), bidding (to automatically select among different available energy providers), and a system of loyalty tokens for the clients (additional incentives for service use).

The research presented in this article relies on a similar client service as the previous one, with the key focus on the design and implementation of an entirely new version of supporting SCs. Instead of a single contract solution, we now elaborated the BC part into a modular platform of various smart contracts to enable a secure, multi-service and multi-tenant DApp backend platform. We present it in the case of smart charging. But the platform can be easily adapted to another novel IoT and 5G application domains, such as collaborative robotics, smart watering, and alike.

The key security contributions of the presented SC platform are as follows. Some additional details about security measures are later given in Table 2.:

- A single contract solution is transformed into a modular one. In each module, methods are provided to freeze and update a smart contract if a security vulnerability is found or functional extensions are required. This facilitates the efficient management of security updates and upgrades. Modularization also lowers the deployment costs if the platform SCs are installed into a public DL network. Every module of the platform has twofold access control.
- At the level of message senders, we proposed and built Smart Contract Tunnels (SCT). This access control mechanism interconnects only the valid SCs in the platform, and prevents malicious SCs from accessing the platform methods.
- At the level of transaction origin, we provide additional, per stakeholder (based on the stakeholder's DL network address) access control to platform methods. Each stakeholder can be granted one of four permission levels, which separately control access to service-specific and platform management smart contracts and methods.
- Modularization and access management enable multi-tenancy and reuse of the deployed platform modules. Different platform modules can be deployed and managed by different stakeholders. The platform owner could, e.g., provide service agnostic modules for platform management and security features, and a service provider would deploy and manage service-specific SCs.

A. STAKEHOLDERS AND THEIR RIGHTS AND RESPONSIBILITIES

We anticipate three different roles of stakeholders in the use cases of the proposed platform. Stakeholders can be persons, legal entities, or devices. For participation in DL communication in the platform, each of them is identified by their DL account/address. The platform provider is the initiator of platform deployment. Commonly, a platform provider would be the owner of the contract that is deployed. Charging station owners are stakeholders registered in the platform. Charging stations are service/application-specific stakeholders; they are registered, too, but in a service-specific module. Both are registered in the system because this is fundamental to control their access to service-specific SCs and methods. The charging station owner is the proprietor of a charging station that he would like to make available to clients via a decentralized system. A charging station is an actual physical IoT device, which has capabilities for DL communication and transaction creation. Unregistered clients are, e.g., electric car owners who would like to use the charging station and the corresponding decentralized application provided by the platform.

This is the fundament for multi-tenancy, where system and platform operation, service management, and service provisioning require the collaboration of various stakeholders in a frequently competitive constellation.

The permissions in the platform are managed on per smart contract basis and can have four levels: owner, admin, registered users, and unregistered user. Every stakeholder, including an unregistered client, is identified by its DL address. The term registration in our case refers to the mapping of these DL addresses to the corresponding permission levels. Unregistered users have no such mapping and therefore cannot use the functions of registered users. They can, of course, use the functions for unregistered users if they have a valid DL account/address that is required for DL communication. Each SC can have an SC owner and one or more SC admins. An SC owner has the right to appoint new SC admins, and the right to initiate migration procedures from a previous to a new version of an SC. An SC admin can add newly registered users to the SCs. Service-specific smart contract methods are accessed by registered users. A charging station owner, for example, is a registered user who can access some of the ChargingStationDirectory methods.

In most cases, modules are owned or administered by the same stakeholders, i.e., the platform provider or stakeholders assigned by the platform provider to these roles. The methods used by owners or admins are used for platform administration, like registering users, handling freezing of operation if the need arises, and handling the migration of modules if the smart contract in the module is updated. The only module that cannot be administered and has no owner is the self-owned module SmartContractIndex. The reason for this is that this module is not expected to change since it is only the address resolution service for modules to be able to interoperate. In the SmartContractIndex, other modules serve as registered users and administer their own information. The information stored in the SmartContractIndex module is available to everyone to provide transparency.

SmartContractAdministration also treats other modules as registered users since they use the SmartContractAdministration to administer their access control, and are the only ones that need to know which users are allowed to access their methods.

TABLE 1. Mapping of a stakeholder to corresponding per-module access levels.

Module	Owner	Admin	Registered user	Unregistered user
SmartContractAdministration	Platform provider	Admin of a particular module	Module	/
SmartContractIndex	/	/	Module	Public access
ChargingStationDirectory	Platform provider	Platform provider or stakeholder assigned by platform provider	Charging station owner	Charging stations Charging service customer
ChargingStationCore	Platform provider	Platform provider or stakeholder assigned by platform provider	Charging service customer, registered	Charging service customer, unregistered
Accumulator	Platform provider	Platform provider or stakeholder assigned by platform provider	Charging station owner	/
Loyalty	Platform provider	Platform provider or stakeholder assigned by platform provider	Charging service customer, registered	Charging service customer, unregistered
Escrow	Platform provider	Platform provider or stakeholder assigned by platform provider	Charging station owner	Charging service customer

TABLE 2. Security measures taken and their impact.

Measures taken	Security impact
Multi-contract solution with leader-member architecture	– Unregistered users can only access one leader SC (ChargingStationCore) – All the remaining SCs are members, method calls are allowed only to the SCs of our solution and SC owners – Leader SC serves as the front-end for unregistered users; – Member SCs can be accessed by other SCs according to the permissions set in SmartContractAdministration – Leader and member SCs can have different owners, who have the right to access them. This enables multi-tenancy at the level of service provisioning.
System management smart contracts:	– A set of system-provided management services
SmartContractDirectory	– Mapping of leader and all member SC addresses in the current deployment. Unlike in ENS, this mapping is hidden for unregistered users.
SmartContractAdministration	– Common access control for all the SCs in the solution. Apart from unregistered (anonymous) users of the service, which are not managed by this access control, SmartContractAdministration records owners, administrators, and registered users of a particular SC in our solution.
Auxiliary smart contracts:	– A set of system-provided application services
Accumulator	– Enables the registered users to retrieve their funds from the Escrow SC. This can reduce gas fees for the registered users' transactions.
Escrow	– Keeps the funds of the communicating entities till predefined criteria are met. Prevents a malicious user from exploiting another user.
Loyalty	– Ensures an ERC20 based loyalty system in the form of a non-fundable token or a stable coin.
Validations:	
Input validation	– Prevents under- and overruns in integer calculations.
Origin validations	– Validates the origin of the transaction and the call and thus supports the use and validation through SCTs.
SC status validations	– Validates whether the smart contract has been deprecated, i.e., upgraded or updated.
Initial setup validations	– Validates that all of the steps of the initial setup were completed and the SC is therefore operational.

Charging station owners are registered entities in the ChargingStationDirectory, which allows them to administer their charging stations. This entails the registration of charging stations, updating information about the charging stations and removing the charging stations that are no longer operational. They are also registered users in the Escrow module since they are the recipient of the funds that pass through this module and have to be a verified stakeholder on the platform. Being a registered user in the Accumulator module allows one to verify the charging station owner address so that they can redeem the funds that have accumulated in their virtual wallet.

Charging stations are unregistered users of the ChargingStationDirectory. This allows them to update their status, which is usually when charging is finished and they want to be made available for booking again.

Charging service clients can be present as registered users in the ChargingStationCore or the Loyalty module. This allows them to collect and redeem the loyalty benefits of using the service if such a service is provided. Having them

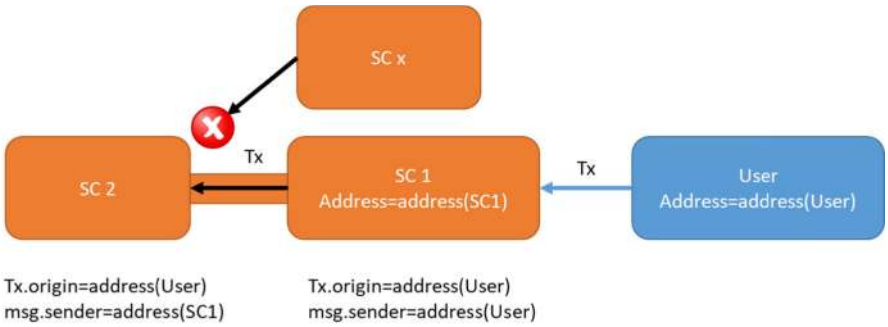

FIGURE 2. Smart Contract Tunneling. To limit the possibility of SC method access to only the SCs explicitly listed in the index SC, we build logical tunnels among the SCs of the current platform implementation, and limit the access rights only to those attached by these Smart Contract Tunnels (i.e., SC 1, and not SC x). In SC 2 we verify both the transaction origin and the message sender. During a transaction pass-through via a Smart Contract Tunnel between SC 1 and SC 2, the transaction origin remains unchanged, but the message sender refers to the sending SC address. Based on its address and SC index, SC x has no access rights to the SC 2 methods so it cannot send or pass the transactions to SC 2.

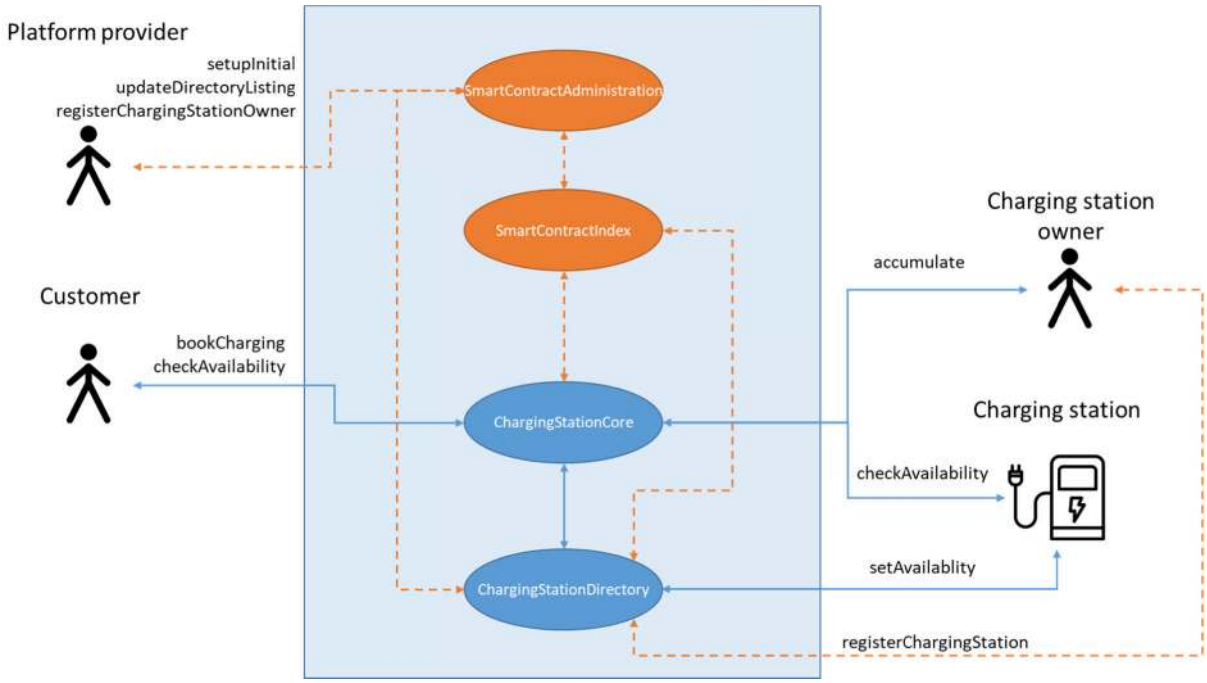

FIGURE 3. Basic platform architecture: basic application-agnostic and application-specific modules, and the stakeholders. The platform provider initially deploys the platform, manages the platform and modules, and appoints additional module admins. A charging station owner registers and manages their charging stations, and accumulates their earnings. A charging station provides electricity charging for the customer, reports availability status and provides billing information. A charging station customer checks the availability of charging service, books it and is charged for it. All these interactions are managed through the smart contracts in the platform.

registered in both modules allows them to unlock additional registered-user-specific methods.

Unregistered charging service clients are still able to use the service so they are present in the ChargingStationCore and the Loyalty module, but they have limited access to the methods in both. Until they register, they are unable to redeem their loyalty rewards, but we still keep track of them in case they want to be registered in the future.

Both registered and unregistered charging service clients are present in the ChargingStationDirectory as unregistered users, where they can query for which charging stations are currently available for booking. They are also both present in the Escrow module when they fund the escrow wallet in order to use a service as an unregistered user of the module.

All of the modules are further explained in subsection IV.C.

B. DESIGN AND PLATFORM ARCHITECTURE

The expected security enhancements, modularization, and multi-tenancy directed the key design requirements.

Modularization simplifies the structure and logic of a particular contract, compared to the single contract solution. At the same time, it enables transparent, partial updates of the platform modules with minimal service interruption. Modularization is the fundament for multi-tenancy, too. The access rights can now be set per module (SC), apart from

the access control related to particular methods. However, modularization introduces inter-contract calls. To exclude additional vulnerabilities as a consequence of inter-contract communications, we manage the access rights for the modules in the platform. This effectively limits the possibility of SC method access only to the SCs explicitly listed in the platform index SC. In this way, we build logical tunnels among the SCs of the current platform implementation, and limit the access rights only to those attached by these Smart Contract Tunnels (SCTs).

Multi-tenancy enables that unregistered clients only access a single application-specific contract and have no direct access rights to platform management smart contracts. This effectively protects the remaining SCs from direct access of anonymous BC users. When the unregistered service clients send a transaction to the service SC, it passes it to the subsequent SCs in the platform (pass-through registration). So the stakeholder identification in SCs can still rely on the transaction origin address.

Transaction origin (tx.origin) is a global variable in Solidity, which returns the address of the original external account that started the transaction, whereas message sender (msg.sender) refers to the immediate account that invokes the function. In a single contract solution message, the sender and the transaction origin are the same address. The difference in the multi-contract solution is depicted in Fig. 2. When a user (identified by its account address) sends a transaction to smart contract SC 1 and it passes through the transaction to SC 2, then in SC 2 the transaction origin (still) identifies the user, but the message sender points to SC 1. To additionally limit access to SC 2 and prevent malicious passing of transactions to SC 2 by, e.g., an attacking SC x performing a man-in-the-middle attack, we grant access to SC 2 methods only to SC 1.

In a practical deployment, ENS would be applied to map the service SC address to a recognizable name for the service clients. All the remaining contracts would only be accessible by their registered owners and administrators, and by the explicitly listed platform contracts.

1) ARCHITECTURE

The platform smart contracts have three different scopes. **Basic application-agnostic contracts** are service and application independent. They provide key platform functionalities required in any multi-contract solution. On the other hand, **application-specific smart contracts** provide unique functions for service operation, in our case, e.g., methods for smart electric charging. In a solution for different (5G) verticals, these contracts would be completely different. The classification as agnostic or specific is rather straightforward, but there are some contracts where the positioning is more debatable. We call these **auxiliary contracts**. They are mainly involved in the service provisioning part and thus directly provide functions required by the application. However, their functions might be so common and appreciated in various verticals that it makes sense to provide them system-wide, too.

The basic architecture of our solution is depicted in the use case diagram in Fig. 3. In the basic architecture, we implemented two basic platform-agnostic contracts, the SmartContractIndex and the SmartContractAdministration, and two service/application-specific contracts, the ChargingStationCore and the ChargingStationDirectory. Different stakeholders have the right to access only some of the SCs and their functionality, according to the access rules in the SmartContractAdministration.

Fig. 4 gives the extended architecture of the platform and smart contracts. The Accumulator, Loyalty, and Escrow smart contracts are auxiliary contracts. These are service-related smart contracts, but might be utilized in many various services. They are therefore provided as a part of the platform similarly as the service-agnostic SmartContractAdministration and SmartContractIndex. The auxiliary contracts in the current deployment are listed in the SmartContractIndex, and their administration is managed by the SmartContractAdministration in the same way as any other contract in the platform.

2) BASIC APPLICATION-AGNOSTIC SCS

SmartContractIndex maps the leader and all member SC addresses in the current deployment. If one of the contracts in the deployed system is upgraded and replaced, the SmartContractIndex reflects this change and transparently directs the remaining contracts to the updated address.

SmartContractAdministration provides common access control for all the SCs in the solution. Apart from unregistered (anonymous) clients of the service, which are not managed by this access control, the SmartContractAdministration manages and records owners, administrators and registered users of a particular SC in our solution, and provides the other SCs with corresponding access rules in inter-contract calls.

3) APPLICATION-SPECIFIC SCS

ChargingStationCore is the key service logic contract and is thus application specific. It implements all the methods needed for smart-charging service flow, and communicates with the following stakeholders: charging station owners, charging service clients, charging stations, and charging administrators.

ChargingStationDirectory is a dynamic mapping service for the ChargingStationCore. It keeps the current list of the charging stations and active charging service clients in the system.

4) AUXILIARY SCS

The accumulator module is a platform provided wallet that allows charging station owners to accumulate the Ether they are paid in exchange for the charging their charging stations provide. Instead of transferring every charging payment to the owner right away, the owner can accumulate the Ether in the accumulator and transfer the desired amount at

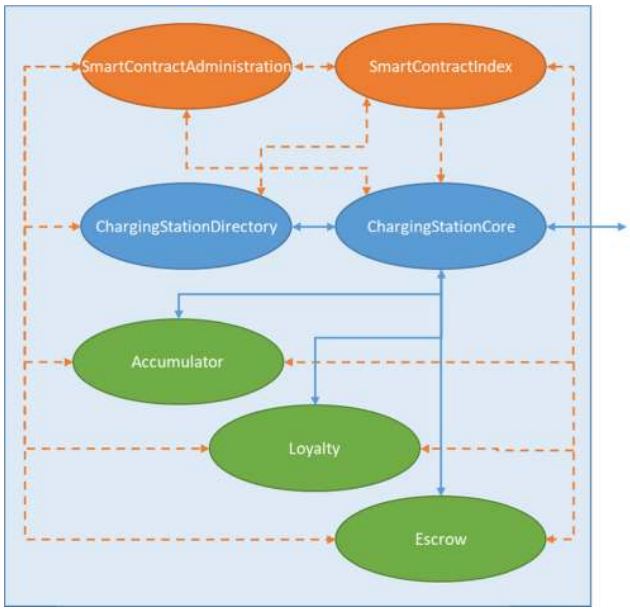

FIGURE 4. Extended platform architecture: basic application-agnostic, auxiliary, and application-specific modules. Auxiliary modules provide additional functions relevant for charging service provisioning. However, their functions are common and relevant for other application verticals, too. Therefore, auxiliary modules (e.g., Accumulator, Loyalty, and Escrow) are provided system-wide. They are managed through the SmartContractAdministration and SmartContractIndex (dashed lines). Their methods are accessed by clients through the ChargingStationCore with a pass-through via the Smart Contract Tunneling mechanism (solid lines).

any time of their choosing to their own or someone else's address.

The loyalty module is the ERC20 standard non-fungible dynamically generated infinite supply loyalty token provider. It allows the allocation of loyalty tokens as a loyalty award to the platform clients. After the charging is completed, the escrow module can allocate the tokens to the clients' accounts in the appropriate amount. The tokens are automatically burned when they arrive back to the accumulator address. An infinite supply of loyalty tokens ensures that every client can be fairly awarded for their use of the platform, and that the value of the token is dictated by the platform provider instead of the clients.

The Escrow module makes it possible to ensure that the charging was provided before awarding the provider the payment. This allows us to operate a trustless platform where only the delivery of the service results in earning a financial award. At the same time, any unused Ether can be refunded back to the client that initiated the charging.

C. MODULES AND METHODS

Some methods are present in all of the modules since they provide interoperability of the modules with SC tunnels. These methods are:

–the **setupInitial** method is used by the platform provider to register the module in the SmartContractDirectory and the SmartContractAdministration. This enables the module and is crucial for smart contract tunneling to work.

–the **updateDirectoryListing** method is used to update the pairing of a module with a smart contract address. It makes it possible to migrate to an updated smart contract, and is necessary for seamless smart contract tunneling reconfiguration.

–the **freezeAndMigrate** method is used to pause or stop smart contract activity, and provide a smooth update to a new version of the SC.

There are methods that are specific to certain modules and only available to the stakeholders with the right access level:

–the **registerChargingStationOwner** method is only available to the platform administrator in the ChargingStationDirectory. It allows an administrator to register a charging station owner, who can then register their charging stations and manipulate their data.

–the **setAvailability** method is a ChargingStationDirectory method that is called directly from the charging station. It is used for billing the client for the service provided by the charging station, and enables charging stations to update their availability status when they complete charging.

The ChargingStationCore module is the single point of interaction for all of the client-facing methods, for both registered and unregistered clients. This module serves as a proxy module that enables the passing-through of the method calls via secure Smart Contract Tunnels to the corresponding method implementing modules. It thus ties all of the other modules together, and in this way secures and simplifies the clients' interaction with the platform. In the basic platform architecture, methods accessed through the Smart Contract Tunnels are:

– **bookCharging** is a pass-through method with a final endpoint in the ChargingStationDirectory module, which allows the client to book charging using a registered charging station.

– **accumulate** is a pass-through method with a final endpoint in the Accumulator module, which allows charging station owners to collect the earnings produced by their charging stations.

– **checkAvailablity** is a pass-through method with a final endpoint in the ChargingStationDirectory used for checking the availability of charging stations for charging.

– **registerChargingStation** is a pass-through method with a final endpoint in the ChargingStationDirectory, which enables charging station owners to register their charging stations.

D. MANAGEMENT AND SERVICE OPERATION FLOW

We distinguish three different workflows in the BC platform of the described DApp: the initial platform deployment, platform management and upgrades, and the basic or extended service operation flow.

1) INITIAL DEPLOYMENT

Initial deployment is the first step after the development of the SCs building the platform is completed. We must

deploy the four contracts of the basic architecture. The auxiliary contracts are optional and are deployed if the service logic requires them. First, the SmartContractIndex is deployed. As it has a very limited functionality and has no advanced functions implemented, it is not meant to be updated or administered. It, therefore, is a self-owned contract and has no owner set after the deployment. Second, the SmartContractAdministration is deployed. The owner of this SC is the deploying address, usually the platform provider's address. We have not chosen to specify an arbitrary owner address with an input parameter during the deployment since the smart contract source code verification on Etherscan, which is done after the deployment, does not support well the passing of additional arguments to the constructor of the contract. Therefore, we opted for the initial setup function instead of passing the constructor arguments when deploying a smart contract. Next, the variable pointing to the SmartContractIndex address is set in the SmartContractAdministration to limit inter-contract calls. Finally, the service-specific smart contracts are deployed. The deployment order for them is arbitrary. After the deployment, the initial setup of these SCs sets the owner and admin address. In the SmartContractIndex, we point to both newly deployed and initialized service-specific contracts. With this, the platform is ready for operation.

2) BASIC SERVICE OPERATION

A service client of the platform does not need to be registered. The service is provided through the ChargingStationCore. The only payable transaction the client has to make is the booking transaction. The operation of the service is as follows:

(1) The client can view available charging stations and plugs on them by querying the ChargingStationCore, which in turn queries the ChargingStationDirectory.
(2) Data about charging stations and their plugs also contains a charging price per unit of charging. We opted for a price per second of charging.
(3) Once the clients decide which charging station they want to book, they initiate a transaction to the ChargingStationCore, which contains information about which charging station and which plug they want to use, as well as the desired duration of charging. The transaction has to be accompanied by the appropriate amount of Ether required for the desired charging parameters.
(4) The ChargingStationDirectory validates that the charging plug on the desired charging station is still free and that the amount of Ether sent with the transaction covers the specified duration of charging.
(5) With validation complete, the ChargingStationCore emits an event notifying everyone listening that the plug was booked and that charging started, and removes the plug from currently available charging plugs. It also forwards the Ether paid for the charging to the charging station owner.
(6) After the charging on the plug is finished, the charging station lets the ChargingStationDirectory know that the plug is available again through the ChargingStationCore. The transaction is validated in the ChargingStationDirectory, which only allows charging stations to modify their own states, and forbids them to modify the states of other charging stations.
(7) As the plug is made available again, the whole process can be repeated.

3) EXTENDED SERVICE OPERATION

Extended service with auxiliary modules implemented has the same initial setup flow as for the basic service modules. Basic service operation is slightly modified to utilize the auxiliary modules. Steps (1) and (2) from the basic service operation remain the same, but further steps are augmented:

(1) Once clients decide which charging station plug they want to use, they send a transaction to the ChargingStationCore, specifying which charging plug they want to use, and the required amount of Ether for charging.
(2) The ChargingStationCore initiates escrow with the Escrow auxiliary module storing the Ether meant for charging until the charging is complete. At the same time, it communicates that charging has to begin with the ChargingStationDirectory.
(3) Once charging has commenced, the charging station sends a transaction to the ChargingStationCore. The core lets the ChargingStationDirectory know that the plug is available again and a new client can book it. At the same time, it lets the Escrow module know that the funds tied to this charging event can be forwarded to the Accumulator module.
(4) The Escrow module forwards the Ether it holds to the Accumulator module and assigns it to the charging station's owner's address. It simultaneously sends a call to the Loyalty module, notifying it about how much the client has paid for the charging.
(5) The Loyalty module assigns ERC 20 standardized loyalty tokens to the client based on how much the client has spent for charging. These loyalty tokens can be redeemed for a certain amount of units of charging at any later time.
(6) The charging station owner can at any time transfer the Ether tied to their own account in the Accumulator module to their own or an arbitrary address.

4) MANAGEMENT PROCEDURES

There are two key platform management procedures: managing access control, and upgrading an SC on the platform. Per SC access control is managed in the SmartContractIndex by the SmartContractIndex owner. It points to the valid addresses

of the SCs in the platform to allow inter-contract communications. Per stakeholder access control of the methods in an SC is done by the SmartContractAdministration. This provides an interface for other platform modules to register various levels of stakeholders that can interact with the module. Every time a limited access function call is to be executed, a module verifies if a user is authorized to execute it. This is achieved with per module access permission and a combination of inherent transaction proprieties; message origin and transaction origin. The SmartContractAdministration module verifies the module that required access control by using msg.sender, and then checks if the user indeed has access to the desired function by using tx.origin.

If any of the SCs in the platform, apart from the SmartContractAdministration, has to be upgraded, the following steps are made. First, the SC administrator temporarily freezes the SC to disable the service providing functions and prevent possible exploitation. Only the functions needed for the update remain active. Then, a new updated version of the SC is deployed to the BC network. Next, the data and funds are migrated from the old to the new SC, and finally, the address of the new SC is added to the SmartContractIndex, thus effectively replacing the pointer to the outdated SC. The platform is now operating with an upgraded SC.

E. IMPLEMENTATION: LESSONS LEARNED

In the current version of the platform, we do not apply any external security libraries because the current ecosystem of blockchain libraries is very volatile. Ensuring that our SCs are secure on their own introduces another layer of security we have to address, but it is a viable long-term solution.

We executed several security code analyses during the development to monitor potential risks due to the modular architecture and implementation. The code was initially analyzed with the Mythril CLI tool, which showed to be a very time-consuming process, especially when inter-contract calls were being added. After applying MythX instead, the security analysis was quicker, but inter-contract calls turned out to be not validatable.

Nevertheless, this provided some interesting insights, such as tx.origin dependence. We are aware of the potential risk of authorization through tx.origin (see Section III.B for details). A call could be made to a vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction, which in this case is the authorized account [73]. The proposed remediation is to use msg.sender instead. We actually do both (see Section IV.B for details). We use msg.sender and SmartContractAdministration for the authentication of valid senders (i.e., only the SCs that build the platform), and to reject possible malicious senders. As transactions can only be sent by trusted SCs, we then use tx.origin for user authorization.

The SmartContractIndex uses an unsigned integer (UINT256) for the indexing of SCs in current and former platform deployments. In Solidity, an overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type; in the event of a UINT256 unsigned integer this is 2^{256}. It is recommended to use safe math libraries for arithmetic operations [74]. But in our case, this parameter is not used in calculations, and merely serves as an index and a counter of the SCs in the platform. It is highly unrealistic (also due to related deployment costs) that any practical DApp implementation would require so many SCs. In case we approached this limitation, we could implement a round-robin index, which overwrites the oldest inputs and thus never exceeds the predefined size.

In some SCs we knew about some SWC violations, like external calls to user-supplied addresses (SWC ID: 107) in the ChargingStationDirectory. This was necessary because it was a means of interconnecting the modules of our platform. Introducing such an SWC violation appears to be acceptable when there are security mechanics implemented that target the point of SWC introduction. In our case, only the SmartContractIndex address is provided at the time of the initial setup of the module.

F. EVALUATION

Before the modularization of the single smart contract architecture Swether platform, the smart contract was very long, about 1000 lines of code. It was difficult to maintain. Anyone other than the original developers would need a significant amount of time to be comfortable enough with the source code to make any changes. The management, adding new features, and security updates started to become inefficient and unreliable. Having one smart contract containing access control, storage of information, and business logic resulted in an intransparent API, which was hard to be documented. Any kind of interaction would prove to be extremely time consuming.

After the modularization rework of the Swether platform, the source code of any module does not exceed 200 lines. Modules are logical units that are easier to understand, maintain, and update. Whereas modularization introduces new challenges, like the need for smart contract tunneling, and adds a need for inter-modular interactions, the clear division of overall business logic, information storage, and client-service level functionalities allows for smoother operation of the platform. Security updates can be performed faster, and the impacted modules hot-swapped without the service being disrupted for a significant amount of time. The clear definition of functionalities provided by the modules also enables us to faster identify the faulty part of the platform, should a problem arise.

With upgradeability in mind, we built the reworked platform in a way that every module is notified of a certain module update. Thus, they can continue operating without the need of the platform administration to reconfigure SCTs for each module that interacts with an updated module.

Inter-SC calls do not introduce any additional latency since the transaction is still executed in a single block. This kind of operation increases the processing demand for EVM, but

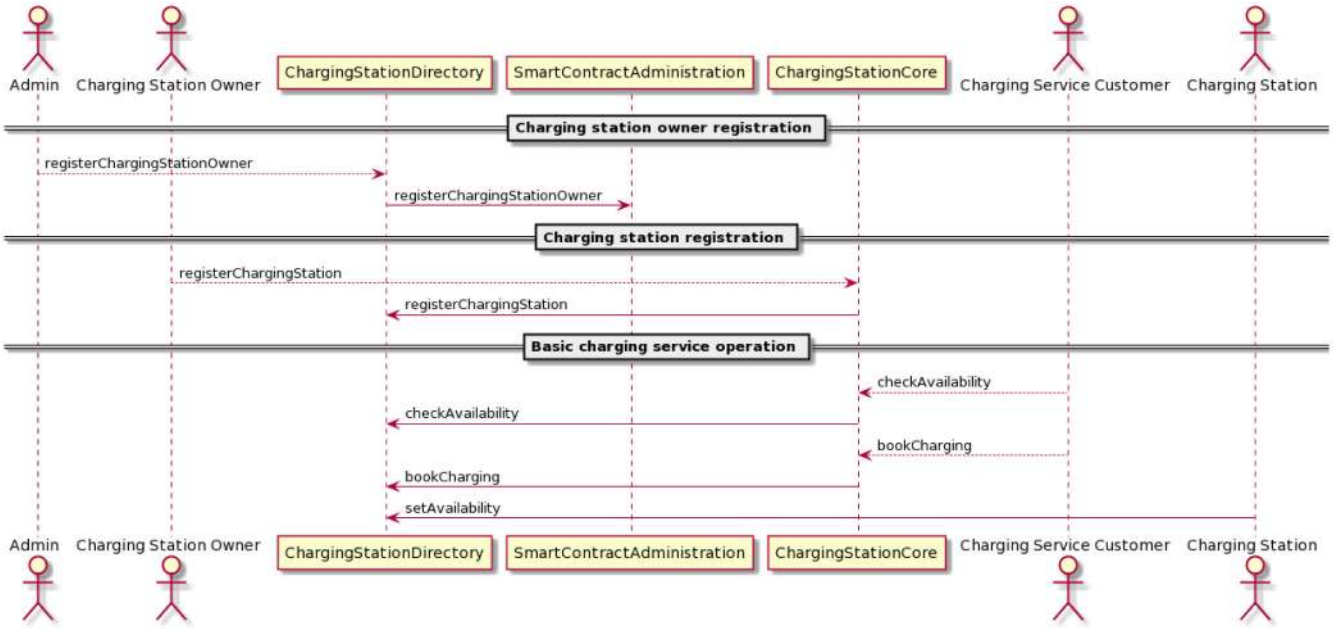

FIGURE 5. Message flows in the initial setup of the platform and the basic charging service operation. For a charging station owner registration, an admin calls the registerChargingStationOwner pass-through function, which passes through the ChargingStationDirectory to the SmartContractAdministration module. There, the admin's access rights for the ChargingStationDirectory are first verified. Next, a new charging station owner is registered as a registered user of the ChargingStationDirectory module. The newly added charging station owners can now register and administer their charging stations. For a charging station registration, a charging station owner calls the registerChargingStation pass-through method, which passes through the ChargingStationCore into the ChargingStationDirectory. The ChargingStationDirectory checks if the charging station owner is a registered user; then the charging station is added to the ChargingStationDirectory and becomes available to charging service customers. In the basic charging service operation, charging station customers query the platform with the checkAvailability method. The query passes through the ChargingStationCore to the ChargingStationDirectory, which returns a list of all available charging stations. Charging station users select and book their preferred charging station with a bookCharging pass-through call. It initiates the charging at the charging station, and removes it from the collection of currently available charging stations. When the charging is complete, the charging station calls the setAvailability method at the ChargingStationDirectory to update its availability and be added back to the collection of the available charging stations.

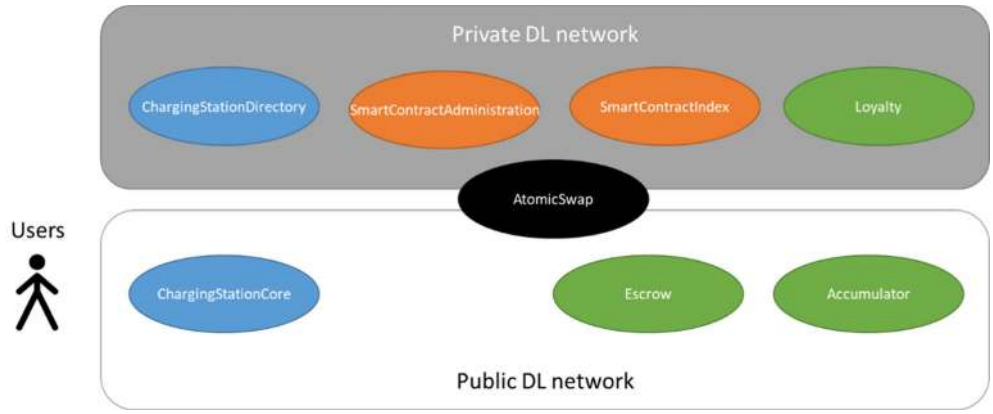

FIGURE 6. Platform deployment for performance, scalability, and cost optimization on a private and a public DL network that are interconnected through an atomic swap. In the public network, the ChargingStationCore, Escrow, and Accumulator smart contracts are deployed because they need to be accessed by unregistered clients or might require an exchange of cryptocurrencies. The remaining platform contracts are deployed in a permissioned private network for additional security and performance.

does not increase the transaction price proportionally to the increase of new calls. If there are three inter-SC calls in a single transaction, the price for a transaction does not increase three times, but only by a fraction of the original cost.

After analyzing the new solution with the MythX and Mythril tools, we found no unexpected SWC issues. Any SWC violation that was reported was introduced consciously, and was handled by own security mechanics. They alleviated the security vulnerabilities that these, by-design implemented SWC violations, could introduce.

Since we have redesigned the single-SC solution into a multi-SC platform, the initial deployment of the platform

requires more transactions than were required in the original service. Whereas the initial SC only required one transaction to be successfully deployed, the new modular design requires four transactions to deploy all of the modules, and then three additional calls to link these modules amongst themselves and establish SCTs.

The deployed modules enable the platform to be operational, but not yet usable. To make the platform usable, we still need to execute three more transactions. These transactions are the registration of a charging station owner executed by the platform administrator, registration of the charging station executed by the previously registered charging station owner, and registration of a charging station plug on the registered charging station, which is again executed by the CS owner. These steps are analogous in both, the former single-contract and the new multi-contract solution. Now our platform is usable, which means that every client needs only one transaction to book and use charging. We do not consider the use of view functions as transactions since they are only an observation of the state of the smart contract, and do not require any transaction fees.

Introducing auxiliary modules would increase transaction counts in each step of the platform setup and use. Each module would need an additional deploy and setup transaction, which means that every new module would increment the number of transactions required for the initial setup, but not by more than two. Depending on the module, there might be one or two additional transactions required to render the module operational. But a bigger increase in the number of transactions – or rather inter-SC calls – would be introduced in extended service operation.

V. MODULAR SMART CONTRACT PLATFORM IN A 5G ARCHITECTURE

We anticipate several levels of integration between 5G and DLT. These levels mostly differ in the architecture and role of the underlying DL networks. The most loose coupling accesses DLT systems through 5G access and services. This is an over-the-top (OTT) like approach, where DL is not integrated into 5G. The positive side of it is that it immediately supports the current public DL networks, too. It is envisioned that future 5G mobile operators would benefit more by using a permissioned consortium or private blockchain networks [5]. The reasons for the introduction of permissioned networks are security, performance, and transaction costs. The introduction of private networks is even more justifiable if the DApps do not require public cryptocurrencies.

Fig. 6 illustrates one possible approach to the deployment of the modular smart contract platform presented in this article. We apply a hybrid approach with one public and one private BC network. In the public network, the ChargingStationCore, Escrow, and Accumulator smart contracts are deployed because they need to be accessed by unregistered clients and require an exchange of cryptocurrencies. The remaining contracts are deployed in a permissioned private network managed by the platform provider or a dedicated consortium. The contracts in the private network do not require public cryptocurrencies. The private network ensures appropriate performance in terms of transaction throughput and delay, and minimizes transaction costs. The atomic swap is a function that interconnects public and private networks at the protocol and semantic level. In this way, service process flows in the private and in the public network remain synchronized, as if the entire platform were implemented in one network.

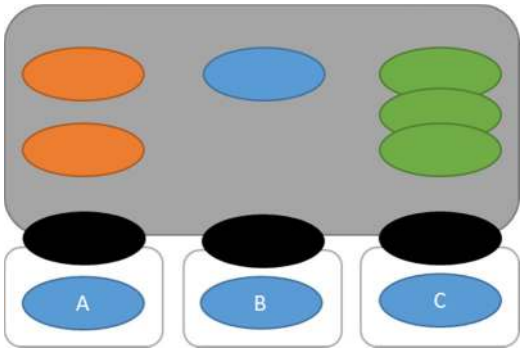

FIGURE 7. Network operators A, B, and C provide DL-based services through their edge private networks, whereas basic application-agnostic and auxiliary smart contracts are accessed via atomic swap in a separate and dedicated private network for performance, scalability and service availability. In this way, the DApp and the service it is providing span over multiple network operators and connect them to a common platform backend.

Additional simplification would be possible if no payments in public cryptocurrencies were required. This, of course, does not exclude accounting and compensation for the provided services. Instead of a cryptocurrency, a token in the private network would implement, e.g., a stable coin for accounting. In this case, the Escrow smart contract would be deployed in the private network, too. The Accumulator would no longer be necessary as it mainly serves to reduce transaction costs in a public DL network. The only contract in the public network would thus be the ChargingStationCore, i.e., the key service contract. Another step would be possible if we could relinquish unauthorized client access. Then the public network could be replaced by a private, too. We could still keep two private networks, a platform-centric and a client-centric one, to support different performance, security, and scalability requirements of the overall business logic and client-service level functionalities.

The hybrid approach can be further generalized to support multiple network operators that provide the DL-based service through their edge private networks, as depicted in Fig. 7. This scales the proposed platform over multiple networks. But unlike in the OTT approach, the system is integrated into 5G. Each network operator (A, B, C) runs its private DL network and deploys the service-specific SC there, and the atomic swap links all the network operators to a common platform backend. In this way, the service integrated into the 5G environment can be provided to the clients independently of their current 5G mobile provider.

VI. CONCLUSION

We extended the multi-contract diamond-cut architecture not only to be upgradeable, but to provide high security and access right management through Smart Contract Tunnels, combined with per-user authorization. Even when partially deployed on a public BC network, the solution does not importantly increase the transaction costs compared to a single contract solution. This builds a platform that can meet the service needs, and can be scaled to modern 5G environments.

Currently, we are adapting the presented platform for a use case in collaborative robotics, where autonomous robots interconnect through a mobile network to negotiate through a hybrid blockchain network for the resource they require or provide.

REFERENCES

[1] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5G wireless networks: A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 1617–1655, 3rd Quart., 2016, doi: 10.1109/COMST.2016.2532458.

[2] D. C Nguyen, P. N Pathirana, M. Ding, and A. Seneviratne, "Blockchain for 5G and beyond networks: A state of the art survey," 2019, *arXiv:1912.05062*. [Online]. Available: http://arxiv.org/abs/1912.05062

[3] I. Ahmad, S. Shahabuddin, T. Kumar, J. Okwuibe, A. Gurtov, and M. Ylianttila, "Security for 5G and beyond," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3682–3722, 4th Quart., 2019, doi: 10.1109/COMST.2019.2916180.

[4] R. Khan, P. Kumar, D. N. K. Jayakody, and M. Liyanage, "A survey on security and privacy of 5G technologies: Potential solutions, recent advancements, and future directions," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 1, pp. 196–248, 4th Quart., 2020, doi: 10.1109/COMST.2019.2933899.

[5] A. Chaer, K. Salah, C. Lima, P. P. Ray, and T. Sheltami, "Blockchain for 5G: Opportunities and challenges," in *Proc. IEEE Globecom Workshops (GC Wkshps)*, Dec. 2019, pp. 1–6, doi: 10.1109/GCWkshps45667.2019.9024627.

[6] A. Sunyaev, "Distributed ledger technology," in *nternet Computing: Principles of Distributed Systems and Emerging Internet-Based Technologies*, A. Sunyaev, Ed. Cham, Switzerland: Springer, 2020, pp. 265–299.

[7] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," 2019, *arXiv:1906.11078*. [Online]. Available: http://arxiv.org/abs/1906.11078

[8] *Ethereum*. Accessed: May 14, 2020. [Online]. Available: https://ethereum.org

[9] *Hyperledger*. Accessed: Apr. 9, 2019. [Online]. Available: https://www.hyperledger.org/

[10] H. Hashgraph. *What is hashgraph consensus*. Accessed: May 6, 2020. [Online]. Available: http://www.hedera.com/learning/what-is-hashgraph-consensus

[11] *EOSIO—Blockchain Software Architecture*. Accessed: May 9, 2020. [Online]. Available: https://eos.io/

[12] *Corda|Open Source Blockchain Platform for Business*. Accessed: May 6, 2020. [Online]. Available: https://www.corda.net/

[13] R. B. Uriarte and R. DeNicola, "Blockchain-based decentralized Cloud/Fog solutions: Challenges, opportunities, and standards," *IEEE Commun. Standards Mag.*, vol. 2, no. 3, pp. 22–28, Sep. 2018, doi: 10.1109/MCOMSTD.2018.1800020.

[14] A. Muthanna, A. A. Ateya, A. Khakimov, I. Gudkova, A. Abuarqoub, K. Samouylov, and A. Koucheryavy, "Secure and reliable IoT networks using fog computing with software-defined networking and blockchain," *J. Sensor Actuator Netw.*, vol. 8, no. 1, p. 15, Feb. 2019, doi: 10.3390/jsan8010015.

[15] R. V. Rosa and C. E. Rothenberg, "Blockchain-based decentralized applications for multiple administrative domain networking," *IEEE Commun. Standards Mag.*, vol. 2, no. 3, pp. 29–37, Sep. 2018, doi: 10.1109/MCOMSTD.2018.1800015.

[16] K. Samdanis, X. Costa-Perez, and V. Sciancalepore, "From network sharing to multi-tenancy: The 5G network slice broker," *IEEE Commun. Mag.*, vol. 54, no. 7, pp. 32–39, Jul. 2016, doi: 10.1109/MCOM.2016.7514161.

[17] J. Backman, S. Yrjola, K. Valtanen, and O. Mammela, "Blockchain network slice broker in 5G: Slice leasing in factory of the future use case," in *Proc. Internet Things Bus. Models, Users, Netw.*, Nov. 2017, pp. 1–8, doi: 10.1109/CTTE.2017.8260929.

[18] W. Cai, Z. Wang, J. B. Ernst, Z. Hong, C. Feng, and V. C. M. Leung, "Decentralized applications: The blockchain-empowered software system," *IEEE Access*, vol. 6, pp. 53019–53033, 2018, doi: 10.1109/ACCESS.2018.2870644.

[19] A. Reyna, C. Martín, J. Chen, E. Soler, and M. Díaz, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Gener. Comput. Syst.*, vol. 88, pp. 173–190, Nov. 2018, doi: 10.1016/j.future.2018.05.046.

[20] X. Zheng, Y. Zhu, and X. Si, "A survey on challenges and progresses in blockchain technologies: A performance and security perspective," *Appl. Sci.*, vol. 9, no. 22, p. 4731, Nov. 2019, doi: 10.3390/app9224731.

[21] P. Cui, U. Guin, A. Skjellum, and D. Umphress, "Blockchain in IoT: Current trends, challenges, and future roadmap," *J. Hardw. Syst. Secur.*, vol. 3, no. 4, pp. 338–364, Dec. 2019, doi: 10.1007/s41635-019-00079-5.

[22] M. A. Khan and K. Salah, "IoT security: Review, blockchain solutions, and open challenges," *Future Gener. Comput. Syst.*, vol. 82, pp. 395–411, May 2018, doi: 10.1016/j.future.2017.11.022.

[23] A. S. Omar and O. Basir, "Identity management in IoT networks using blockchain and smart contracts," in *Proc. IEEE Int. Conf. Internet Things*, Jul. 2018, pp. 1–7, doi: 10.1109/Cybermatics_2018.2018.00187.

[24] C. H. Lau, K.-H.-Y. Alan, and F. Yan, "Blockchain-based authentication in IoT networks," in *Proc. IEEE Conf. Dependable Secure Comput. (DSC)*, Dec. 2018, pp. 1–8, doi: 10.1109/DESEC.2018.8625141.

[25] A. Stanciu, "Blockchain based distributed control system for edge computing," in *Proc. 21st Int. Conf. Control Syst. Comput. Sci. (CSCS)*, May 2017, pp. 667–671, doi: 10.1109/CSCS.2017.102.

[26] A. Dorri, S. S. Kanhere, and R. Jurdak, "Blockchain in Internet of Things: Challenges and solutions," 2016, *arXiv:1608.05187*. [Online]. Available: http://arxiv.org/abs/1608.05187

[27] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, "Blockchain for IoT security and privacy: The case study of a smart home," in *Proc. IEEE Int. Conf. Pervas. Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2017, pp. 618–623, doi: 10.1109/PERCOMW.2017.7917634.

[28] N. H. Kim, S. M. Kang, and C. S. Hong, "Mobile charger billing system using lightweight blockchain," in *Proc. 19th Asia–Pacific Netw. Operations Manage. Symp. (APNOMS)*, Sep. 2017, pp. 374–377, doi: 10.1109/APNOMS.2017.8094151.

[29] J. Kang, R. Yu, X. Huang, S. Maharjan, Y. Zhang, and E. Hossain, "Enabling localized peer-to-peer electricity trading among plug-in hybrid electric vehicles using consortium blockchains," *IEEE Trans. Ind. Informat.*, vol. 13, no. 6, pp. 3154–3164, Dec. 2017, doi: 10.1109/TII.2017.2709784.

[30] K. Wu, Y. Ma, G. Huang, and X. Liu, "A first look at blockchain-based decentralized applications," *Software: Pract. Exper.*, vol. 1, pp. 1–18, Oct. 2019, doi: 10.1002/spe.2751.

[31] *Bitcoin—Open Source P2P Money*. Accessed: Aug. 2, 2016. [Online]. Available: https://bitcoin.org/en/

[32] *Mainnet*. Accessed: May 6, 2020. [Online]. Available: https://docs.hedera.com/guides/mainnet

[33] C. Cachin. *Architecture of the Hyperledger Blockchain Fabric*. Accessed: Apr. 28, 2020. [Online]. Available: https://www.zurich.ibm.com/dccl/#program

[34] *Private, Public, and Consortium Blockchains - What's the Difference*. Accessed: Apr. 23, 2020. [Online]. Available: https://www.binance.vision/blockchain/private-public-and-consortium-blockchains-whats-the-difference

[35] A. M. Antonopoulos and G. Wood, *Mastering Ethereum—Building Smart Contracts and Apps*. Newton, MA, USA: O'Reilly Media, 2018.

[36] A. M. Antonopoulos and G. Wood. *Mastering Ethereum: What Is a Smart Contract*. Accessed: Apr. 2, 2020. [Online]. Available: https://github.com/ethereumbook/ethereumbook

[37] (Apr. 2018). *Linux Foundation Networking*. [Online]. Available: https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf

[38] N. Gopie. *What are smart contracts on blockchain*. Accessed: Apr. 2, 2020. [Online]. Available: https://www.ibm.com/blogs/blockchain/2018/07/what-are-smart-contracts-on-blockchain/

[39] *What is an ABI and Why is It Needed to Interact With Contracts*. Accessed: Jun. 5, 2020. [Online]. Available: https://ethereum.stackexchange.com/questions/234/what-is-an-abi-and-why-is-it-needed-to-interact-with-contracts
[40] *Ethereum Name Service (ENS)*. Accessed: Apr. 15, 2020. [Online]. Available: https://docs.ens.domains/
[41] A. Fischer. *IPFS & SWARMethersphere/swarm*. Accessed: Apr. 15, 2020. [Online]. Available: https://github.com/ethersphere/swarm
[42] *Provable*. Accessed: Apr. 15, 2020. [Online]. Available: https://provable.xyz/
[43] M. Pustišek, A. Umek, and A. Kos, "Approaching the communication constraints of ethereum-based decentralized applications," *Sensors*, vol. 19, no. 11, p. 2647, Jun. 2019, doi: 10.3390/s19112647.
[44] M. Saad, J. Spaulding, L. Njilla, C. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A systematic overview," 2019, *arXiv:1904.03487*. [Online]. Available: http://arxiv.org/abs/1904.03487
[45] *Blockchain Attack Vectors: Vulnerabilities of the Most Secure Technology*. Accessed: Jan. 3, 2020. [Online]. Available: https://www.apriorit.com/dev-blog/578-blockchain-attack-vectors?jve=1577524315
[46] *Ethereum Smart Contract Security Best Practices*. Accessed: May. 20, 2020. [Online]. Available: https://consensys.github.io/smart-contract-best-practices/
[47] T. Chen, "Understanding ethereum via graph analysis," in *Proc. IEEE Conf. Comput. Commun.*, Apr. 2018, pp. 1484–1492, doi: 10.1109/INFOCOM.2018.8486401.
[48] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, "Smart contract security: A software lifecycle perspective," *IEEE Access*, vol. 7, pp. 150184–150202, 2019, doi: 10.1109/ACCESS.2019.2946988.
[49] *CWE—Common Weakness Enumeration*. Accessed: Mar. 20, 2020. [Online]. Available: https://cwe.mitre.org/index.html
[50] *SWC—Smart Contract Weakness Classification and Test Cases*. Accessed: Mar. 20, 2020. [Online]. Available: http://swcregistry.io/
[51] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 254–269, doi: 10.1145/2976749.2978309.
[52] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *Proc. IEEE 24th Int. Conf. Softw. Anal., Evol. Reengineering (SANER)*, Feb. 2017, pp. 442–446, doi: 10.1109/SANER.2017.7884650.
[53] S. Rouhani and R. Deters, "Security, performance, and applications of smart contracts: A systematic survey," *IEEE Access*, vol. 7, pp. 50759–50779, 2019, doi: 10.1109/ACCESS.2019.2911031.
[54] (2020). *Leonardo*. [Online]. Available: http://leonardoalt/ethereum_formal_verification_overview
[55] *ConsenSys/mythril*, ConsenSys, New York, NY, USA, 2020.
[56] *MythX: Smart Contract Security Service for Ethereum*. Accessed: Jun. 2, 2020. [Online]. Available: https://mythx.io/
[57] *MythX Smart Contract Weakness Classification (SWC) Coverage and Status*. Accessed: Jun. 15, 2020. [Online]. Available: https://mythx.io/detectors/
[58] *Melonproject/Oyente*, Melonproject, San Francisco, CA, USA, 2020.
[59] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2018, pp. 9–16, doi: 10.1145/3194113.3194115.
[60] *SmartCheck*. Accessed: Jun. 2, 2020. [Online]. Available: https://tool.smartdec.net/
[61] *SmartCheck|KnowledgeBase*. Accessed: Jun. 2, 2020.[Online]. Available: https://tool.smartdec.net/knowledge
[62] *Securify: Security Scanner for Ethereum Smart Contract*. Accessed: Jun. 2, 2020. [Online]. Available: https://securify.chainsecurity.com
[63] P. Tsankov, A. Dan, D. Drachsler Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," 2018, *arXiv:1806.01143*. [Online]. Available: http://arxiv.org/abs/1806.01143
[64] T. Kaiser. *ChainSecurity's Chaincode Scanner: A powerful Security Checker for Hyperledger Fabric Smart*. Accessed: Jun. 3, 2020. [Online]. Available: https://medium.com/chainsecurity/release-of-hyperchecker-2dff2ebe30cc
[65] *Ethereum (ETH) Blockchain Explorer*. Accessed: May 6, 2020. [Online]. Available: http://etherscan.io/
[66] *Diamond Standard Issue Ethereum/EIPs*. Accessed: Jun. 15, 2020. [Online]. Available: https://github.com/ethereum/EIPs/issues/2535
[67] *OpenZeppelin*. Accessed: Jun. 2, 2020. [Online]. Available: https://openzeppelin.com/
[68] R. M. C. Forster. *Blockd Build: Active Smart Contract Security*. Accessed: Jun. 1, 2020. [Online]. Available: https://medium.com/blockd/blockd-build-active-smart-contract-security-3598170ad3a2
[69] M. Pustišek, N. Bremond, and A. Kos, "Electric Switch with Ethereum Blockchain Support," *IPSI TIR*, vol. 14, no. 1, pp. 21–28, Jan. 2018.
[70] M. Pustišek and A. Kos, "Approaches to front-end IoT application development for the ethereum blockchain," *Procedia Comput. Sci.*, vol. 129, pp. 410–419, 2018, doi: 10.1016/j.procs.2018.03.017.
[71] M. Pustišek, D. Dolenc, and A. Kos, "LDAF: Low-bandwidth distributed applications framework in a use case of blockchain-enabled IoT devices," *Sensors*, vol. 19, no. 10, p. 2337, May 2019, doi: 10.3390/s19102337.
[72] *Now Christmas Trees Are Going on the Blockchain—Decrypt*. Accessed: Jun. 7, 2020. [Online]. Available: https://decrypt.co/15313/now-christmas-trees-are-going-on-the-blockchain
[73] *SWC—Smart Contract Weakness Classification Test Cases*. Accessed: Jun. 7, 2020. [Online]. Available: https://swcregistry.io/docs/SWC-115
[74] *SWC—Smart Contract Weakness Classification Test Cases*. Accessed: Jun. 7, 2020. [Online]. Available: https://swcregistry.io/docs/SWC-101