Journal of Systems Architecture 153 (2024) 103177

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

VelogCPS: A safe blockchain network for cyber–physical systems leveraging block verifiers

Marisol García-Valls *, Alejandro M. Chirivella-Ciruelos

Departamento de Comunicaciones, Universitat Politècnica de València, Valencia, Spain

ARTICLE INFO

Keywords:
Cyber physical system
IoT
Blockchain
Verification
Block verifier
Smart contract
Authenticity
Security
Safety
Real-time
Time sensitive system
Middleware

ABSTRACT

Non-functional requirements related to safety, security, and timeliness have made cyber–physical systems (CPS) initially reluctant to their integration with blockchain technology. Despite the multiple advantages of blockchain like improved data security and traceability, the main reasons that have slowed down its adoption in CPS still remain. Examples of these are the inherent overhead of accessing the distributed ledger and the security incidents that a number of blockchain networks have suffered since its inception. This paper presents VelogCPS, a novel middleware that guarantees that logic and data managed by blockchain networks of cyber–physical systems are verified and generated by legitimate sources. Thus, VelogCPS avoids a kind of security incidents that impact the authenticity and integrity of the logic and data managed in blockchain networks. By authenticity we refer to provenance authenticity of the involved smart contracts, i.e., the perfect matching between the advertised source-code and the version deployed to the network. Our framework provides a safe blockchain network as it ensures that the entities that participate to a CPS use solely authentic logic. We do this by leveraging block verifier services and enforcing them through the operation workflow. As a result, the middleware guarantees that the participating entities use and share authentic logic. The proposed framework is validated through its implementation on a real blockchain network, employing actual smart contract verifier logic, and through the exhaustive analysis of the temporal behaviour and overhead of the major operations; the obtained results ensure its utility for time-sensitive sytems like CPS and IoT.

1. Introduction

Constant improvement of hardware and software technology has enabled the development of increasingly complex applications that span across heterogeneous execution substrates and distinct domains. Cyber–Physical Systems (CPS) are a kind of such complex systems. CPS are abundant in the use of sensing and actuating equipment, comprising highly heterogeneous hardware nodes running diverse logic, and they may conform to complex software computational paradigms.

Cyber–physical systems merge the physical and cyber (computational) domains. This mixture gives birth to new applications that express also new economic models arising in the emerging global markets. This is the case of virtual marketplaces, distributed and global supply chains, connected global eHealth, and demand-driven markets, just to name a few. These new trends often imply a global cost and revenue model over the basis of a cost-effective business structure.

As a realization of distributed ledger technology (DLT), blockchain enables trust and decentralization in collaborative business models. Though initially created for cryptocurrency transactions, blockchain soon captured the attention of the scientific and engineering community for a broad range of application domains, e.g., government [1], IoT ecosystems [2], eHealth [3,4], voting [5], or industry [6]. Such complex systems have proven to benefit from the emerging software developments that support the realization of trusted, though decentralized, processes that are intensive in data access and sharing. This is also the case of CPS and IoT.

Though it has some amazing inherent benefits, blockchain technology is not perfect. In deed, a number of attacks have successfully hit some blockchain networks in the recent past. This has caused great economic losses and trust erosion especially to those users belonging to most critical domains. This is the case of the DAO (Distributed Autonomous Organization) incident [7]; over three and a half million Ether (the currency used in Ethereum [8]) were drained by a cyber-attacker by exploiting a vulnerability of the system. Another paradigmatic incident happened in 2017 with the Parity Wallet hack [9] that was attacked in Ethereum, causing around 30 million USD loss.

These incidents have occurred because a number of factors may turn *smart contracts* (their underlying software concept and building block)

* Corresponding author.
E-mail addresses: mgvalls@dcom.upv.es (M. García-Valls), alchici@dcom.upv.es (A.M. Chirivella-Ciruelos).

https://doi.org/10.1016/j.sysarc.2024.103177
Received 23 June 2023; Received in revised form 14 April 2024; Accepted 13 May 2024
Available online 18 May 2024

into vulnerable programs. A part from the potential vulnerabilities that execution platforms such as EVM (Ethereum Virtual Machine) contain, programming the source code of a smart contract in a clean, error-free manner is a hard task. This job requires extensive training in the target programming language. Introducing a single bug in the source code may result in losses worth millions USD. Additionally, a blockchain network does not guarantee *smart contract authenticity* per se. In fact, cyberattackers may employ different techniques to break the one-to-one association between an advertised smart contract and its corresponding bytecode that runs in the blockchain network.

The global ledger of a blockchain network stores a number of running smart contracts, ready to receive invocations from users. Users can benefit from the functionality of the running smart contracts by invoking their available functions. To be reachable by network participants (i.e., users), smart contracts are advertised in some network dashboard. By looking up in the dashboard, participants can find smart contracts of interest and request the functionality that they provide. The functionality of a smart contract is requested by invoking the available functions that it contains. Nevertheless, if a successful attack has occurred in the network, the running version of the involved smart contract and its corresponding advertised source code may not be the same. If both items (i.e., source code and running code) fully coincide, then the smart contract is considered *authentic*.

In essence, *smart contract authenticity* [10] refers to the accreditation as truthful and being a true reflection of the characteristics and requirements that are advertised in the network dashboard. As a result, verifying the authenticity of a smart contract is the process of checking that a smart contract source code and the corresponding version that runs on the global ledger are the same.

As a result, to achieve 100% certainty about the correctness of a smart contract, it must be checked in a number of orthogonal plains. On the one hand, they should be checked for errors in order to ensure their correctness, that their execution is secure, and that their safety properties (if any is required) are met. On the other hand, it is of paramount importance that the authenticity of a running smart contract is ensured in order for potential users to be safe when using it. As mentioned before, we employ the term *authenticity* as explained in [10]: Authenticity involves certainty about the smart contract provenance. This implies that there must be a one to one relation between the running version of the smart contract and the advertised source code.

As cyber–physical systems must comply to stringent safety and security requirements, smart contract authenticity is an essential characteristic in them. We address the complexity of cyber–physical systems considering that they are conformed by a set of interconnected subsystems or participant entities/nodes, namely *cells*. Consequently, all cells may require certainty about the authenticity of the data and logic hosted in the distributed ledger prior to using them.

1.1. Attack model

We assume potential presence of malicious actors with the objective of intentionally altering the functionality offered by smart contracts. Attackers may attempt to modify the executable version of a smart contract that, when successful, appears transparent to the system participants. Following, we explain how this may happen.

A smart contract is written in some high level programming language. Examples of most commonly used languages for this purpose are Solidity (the most popular one), Vyper, Yul, Cairo, or Rust. This source code must eventually be compiled into bytecode, becoming an executable program. Once the source code has been transformed into an executable version, it can be deployed to the blockchain, and run on some execution run-time environment like the Ethereum Virtual Machine (EVM) [8] or Hyperledger Fabric [11].

As a result, there is a transformation of the original source code version of a smart contract to an executable version. A successful attack on the *authenticity* property of a smart contract yields two situations. On the one hand, the dashboard could be attacked causing a modification of the advertised code. On the other hand, the executable version may be modified resulting in a different executable file being uploaded to the distributed ledger. As a consequence of a successful attack, a user cannot trust that the running version of a smart contract is exactly the same as the advertised source-code version.

1.2. Contribution

Our work contributes to ensuring that only authentic logic is used over the global operation space of the cells of a cyber–physical system. For this, the involved pathways to ensuring authenticity in smart contracts are analysed. The first one is related to the deployment of bytecode to the network; and the second one relates to the process of verifying the contract. For this, we design and develop a middleware named VelogCPS that embeds *verification of logic (i.e., the smart contracts) hosted in CPS cells*, achieving a solution that ensures that any execution of a smart contract function is performed only if the running logic is found to be authentic. The middleware leverages on block verifiers for performing the verification tasks.

Providing trust to the cells in this way avoids using third party regulatory agencies that would break the trust model of the blockchain. VelogCPS middleware is designed as a modular intermediation point that transparently grants access to legitimate logic. It does so without altering the mechanisms of the blockchain substrate like block mining, block validation, and network consensus.

This work does not address the verification of the correctness of smart contracts for which formal verification techniques, source code verification methods, and software analysis are employed. Such techniques and methods have been long studied in the past, yielding a large number of contributions on formal methods, model checking, theorem proving, or run-time verification, that are out of the scope of this paper.

1.3. Paper organization

The document is organized as follows. Section 2 describes the context and related work. Section 3 describes the requirements for the framework and the attack model. Section 4 provides a theoretical and practical description of the framework that provides a safe blockchain network for cyber-physical systems; VelogCPS middleware is presented, describing its design, functionality, and programming interface Section 5 validates the contribution through its implementation over an actual blockchain network, the design of a set of experiments and their realization that shows the temporal behaviour of the middleware, and the cost–benefit relation of the authenticity assurance strategy. Eventually, Section 6 concludes by reviewing the benefits of our approach.

2. Background

The introduction of the concept of *smart contract* has boosted the power of DLT and blockchain as a technology to implement decentralized applications (dApps). A smart contract is a program that runs in the blockchain network and describes the terms of an agreement. A smart contract stored in the distributed ledger runs automatically when the terms are met, being capable of changing the state of the contract itself and, therefore, of the global ledger. Once stored in the ledger, the program logic is immutable.

A potential user of a smart contract has no means to know whether a given running smart contract truthfully implements the source code that the user has consulted. In an ideal network, this would have no negative consequences. However, in a real context where attacks may occur, this model needs to improve its security. As a consequence, the blockchain technology community has realized the importance of *smart contract verification* as a way to build trust between smart contract users

and developers. In fact, it is the corner stone to build trust on advertised smart contracts.

The use of the term *verification* connects the minds of the general audience to formal verification techniques and software analysis. This effect happens also when referring to *verification* in the context of smart contracts. The reason is the still relative novelty of blockchain technology and the scarce number of works on *provenance verification* of smart contracts in blockchain networks.

However, formal verification and software analysis is orthogonal to the domain of our work. Our work does not address source code verification nor model checking techniques. The present work targets a novel way of looking into the needs of cyber–physical systems that rely on blockchain substrates and on the logic that is stored in them. Only few works have been produced so far, including the development of *block verifiers* that are later indicated in this section. Our contribution leverages on these verifiers that are remote services offered to smart contract developers to build trust on the provided functionality and logic.

The existing block verifiers are nowadays still limited to offering off-line smart contract verification that takes place at the initial phase of the programming and advertising.

In this section, we survey both aspects: formal verification and software analysis together with smart contract verification. Additionally, and to better draw the overall context, we describe other areas of credential management and also works on construction of global operation spaces. We anticipate a scarce number of works and technologies that constitute the reduced background of our work, as explained above in relation to its novelty.

2.1. Blockchain-based global operation spaces

In the recent years, a number of works have been released that address the integration of blockchain technology with IoT applications [2] in some form. These are applied to a number of domains like eHealth [3,4], agriculture [12], or electronic voting applications [5], among many others.

On the usage of the blockchain as a global space, we find CoTwin [13] that develops a global operation space for improving digital twin models based on deep learning over a blockchain substrate; or [14] that describes a model for document verification for government systems to achieve decentralized sharing of authenticated government documents across private bodies and organizations. A complementary approach is presented in [15] that describes an automated credential verification system based on the blockchain to handle academic certificates; the system consists of a web app having a front-end for registering and requesting verification, and a backend part with two functionalities like an OCR module to collect certificate details and a blockchain module to send and verify data that is located in the blockchain.

Targeting the construction of a global operation space, we find [16] that describes a solution for document sharing; here, documents have a unique hash key used to validate its authenticity.

In the area of control for cyber–physical systems and of autonomous robotic devices, [17] describes a model to support verification of critical data shared and exchanged by nodes; it is architected through the configuration of discrete multiple-valued logic to provide data protection against illegitimate modification attempts.

In the field of manufacturing and production, [18] conceptualizes the interaction in the supply chain among partners, defining validation rules for transactions and providing trust.

Though still far, the above have been found to be the closest to our target system in the sense that they attempt to provide a global operation space where the managed data objects are trusted. To the best of our knowledge, there is no contribution that provides a means to ensure that the logic deployed to, and executed from, a blockchain-enabled global space is authentic at all times. To illustrate this statement, most related works have been identified as representatives of this kind.

2.2. Smart contract verification services

This section presents the set of contributions most related to our work that use 4 same meaning for *smart contract verification* [10] as the one used in our approach. After surveying the research landscape, the scarce research works that have been found are limited to block verifiers (a.k.a verification services) and a few others to analysing their behaviour and vulnerabilities.

Block verifiers should not be confused with block validators. The latter are mandatory participants in a network because they are in charge of validating the correctness of the block mining process.

However, *block verifiers* are a later add-on that is optional to a blockchain network. Block verifiers have appeared given the progressive growth of the number of available smart contracts on the public networks. A running smart contract can be used by anyone in a public network, but using it is (in the end) a matter of trust since its bytecode cannot be read. A block verifier is a service that answers the question of whether the source code of an executable version of a smart contract matches exactly a given (other) source code. Provenance authenticity is a real challenge in cyber–physical systems because the trustability of the logic contributed by all cells must be guaranteed. As a consequence, we believe that the process of block verification (or smart contract verification) has to be present in a complex cell-based system.

Currently, only some external platforms exist for verification of smart contracts like Tenderly [19], Sourcify [20], Blocksout [21], or Etherscan [22], among others. These platforms provide an own dashboard containing the set of smart contracts that have obtained a verification seal. However, this process is decoupled from the usage of the particular blockchain that might have been configured.

The design and development of solutions to improve the block verification schemes opens a wide research space. As indicated in [23], the community is just starting to learn about a few types of vulnerabilities that affect some smart contract verification services which can break the verification; this results into spreading malicious smart contracts. The work of [23] is the most relevant on the literature about the vulnerabilities of block verifiers. They propose a set of security properties to be satisfied by block verifiers and identify the types of potential vulnerabilities that can break the verification when the security properties are violated. The difference with respect to our present contribution is that the work is a security analysis of Ethereum smart contract services, precisely for Etherscan [22], Sourcify [20], and Blockscout [21]. Our contribution does not focus on the identification of security issues inside block verifiers. We leverage on block verifiers to provide a framework that transparently ensures cells that they are using authentic logic. For this, we assume that the verification process of the block verifier entities is not broken.

In a previous work [24], we have used block verifiers through the instrumentation of the Hardhat [25] plain scripting facilities to deploy smart contracts on public blockchain networks. We had engineered a scheme to automate smart contract verification tries and assess the interaction with some of the major block verifiers. To the best of our knowledge, this has been among the first published works on the initial determination of the temporal characteristics of block verifiers and their usability for smart contract verification. We extended this work in [10] where we designed a middleware to automate the deployment of smart contracts on real networks. However, that work focused solely on the assessment of the temporal behaviour of the verification process. In the present contribution, we substantially improve our previous works by designing a middleware framework targeted at cells that work cooperatively against a shared space based on blockchain technology. With the knowledge gained in [10], we design a framework that transparently checks the logic requested by cells and ensures that it is authentic. When a cell wishes to use a given logic, it does so through our framework VelogCPS. The framework performs the needed operations to determine whether the logic has been previously verified. It only allows the cell to use a logic if it has been verified or if

VelogCPS can verify it on-the-fly without detecting any authenticity breach during the process. This is a large step forward because if a cell ecosystem within a cyber–physical system uses VelogCPS, it will be guaranteed that the logic used in this ecosystem is authentic. This means that the logic provenance is the expected one, i.e., it comes from a legitimate source.

2.3. Software analysis

From a software analysis perspective, verification refers to checking the correctness of a smart contract; this consists of determining whether the programmed logic adheres to the initial specifications. Also, it implies the determination of whether the possible executions of the contract produce the expected behaviour and outcome.

There are two main ways to achieve the needed certainty level. On the one hand, formal verification techniques (that are based on mathematical methods) enable the determination of predefined properties of the model. For example, a given output value is one from the expected set of valid values. On the other hand, source code verification techniques ensure that the programmed code is correct; e.g., absence of attempts to access undefined memory positions, or absence of race conditions, among others.

A considerable number of vulnerabilities in smart contracts are caused by lack of coding proficiency that results in the introduction of code bugs. In a blockchain network, it is not possible to modify a smart contract after it has been deployed. A new logic can be deployed to overcome such a situation, but the former will stay immutably in the global ledger. Fighting bug coding can be done by investigation of methods to detect vulnerabilities for improving the security; these methods typically consist of analysing the code against a list of predefined well-known vulnerability patterns. In this respect, different techniques have appeared These are well summarized in [26] that surveys smart contract *code* verification contributions. Other approaches like [27] present the landscape of verification for blockchain applications but, once more, solely focused on software analysis techniques to study the correctness of smart contract code. Nevertheless, such works focus solely on *formal analysis of* source code, which is out of the scope of our work.

2.4. Credential management and authentication

Our contribution does not address credential verification involving identity management, password authentication mechanisms (e.g., through smart card usage, etc.), or two-factor authentication schemes, among others.

There is a long list of challenges in designing multi-factor authentication schemes. Among these, [28] addresses real-time aspects of cyber–physical systems by targeting wireless sensor networks for real-time data access. It uses the imbalanced computational nature of the RSA cryptosystem, and it is targeted to scenarios where sensor nodes are energy bottlenecks.

Other authentication schemes like [29] develop quantum-attack resistant approaches to smart-card password authentication schemes.

Also, privacy-preserving authentication schemes based on blockchain technology have been developed like [30]. That work relies on an elliptic curve to develop proper signature schemes; whereas the blockchain is used to provide a more reliable service. However, the overall scheme is proven in a fully simulated environment.

These are orthogonal domains with a substantial mass of prior works. All of them concentrate on solving a particular and different problem.

2.5. Summary and missing gap

As derived from the above arguments, and to the best of our knowledge, there is a lack of contributions on ensuring smart contract authenticity for complex systems like CPS. In this paper, we contribute to a complementary yet different direction to the existing areas. We target the need of distributed CPS cells to construct an efficient global execution space that is trusted by all participants. For this, our solution leverages block verifiers, global operation spaces for CPS and focuses on the actual integration of mechanisms and strategies that improve the security of the global system operation, where participants have the guarantee that they are using purely authentic logic.

For this, we design VelogCPS framework that offers remote cells a blockchain-based global operation space with built-in assurance about the authenticity of the logic that is hosted and executed on the network. This results in the overall security being improved. To meet the time-sensitive expectations of cyber–physical systems, the resulting scheme is validated over a real deployment, showing that the framework has a timely behaviour and a controlled execution overhead.

3. System model

3.1. Overview and requirements

The target scenario is that of a complex cyber–physical system that interconnects distributed cells, namely CPS cells, as illustrated in Fig. 1. A *cell* is a full fledged system. Examples are a manufacturing premises, i.e., a manufacturing cell within a globally connected system for a given brand manufacturer; a railway station within a connected regional railway system; or a hospital department of a larger hospital facilities, among others.

All cells participate equally to the global system. They are interconnected and able to share data and operation logic. The operation logic is the set of procedures that dictate how the system functions to achieve its goals. For the sake of realization, the global operation logic is broken down into a set of smaller, highly focused logic pieces. Each piece is named *logic*. For example, the global operation logic may be the management of the logistics of a car manufacturer across all its sites around some geographical area. In this example, *logic* pieces are the processes or algorithms to detect shortage of raw material, the process to annotate that a new car or asset has been produced, or the algorithm to determine the total amount to be paid to a given cell in charge of supplying a particular asset to other cells. In summary, cells are interconnected and participate in the system operation. Cells can read and update the global space and the logic contained in it.

In cyber–physical systems, the *execution time* of their constituent processes, operations, and functions is considered a key parameter. In fact, the execution time of its functionality or processes determines the correctness the system. Also, the security requirements over the global space must be fulfilled. For this reason, the global operation space must be a safe area and its content must be authentic. When accessing the global operation space, cells will use solely authentic logic, i.e., smart contracts that have been developed and deployed by a legitimate participant and whose integrity has not been breached. The logic hosted in the blockchain must be from a legitimate source, i.e., from a participant cell.

To summarize, it follows the list of requirements:

Req1. The solution must target the complex and distributed nature of cyber–physical systems, supporting individual cells with equal participation opportunities.

Req2. All cells are interconnected and share the global system's operation logic. Cells may dynamically produce information as well as logic that can be contributed to the global space. All cells may be able to access the information and logic produced by other cells.

Req3. The solution must be a secure and fraud-preventive infrastructure. It must be possible to trace the hosted logic pieces to know which cell (if any) has produced each piece.

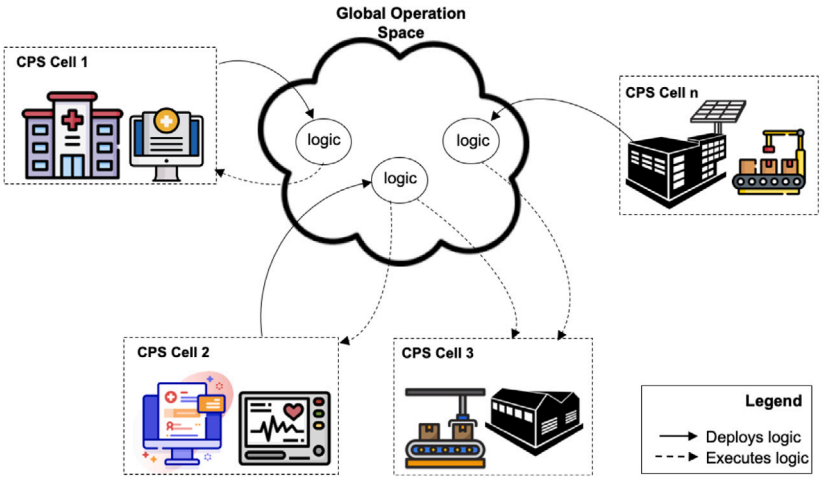


Fig. 1. Cyber-physical system overview, containing distributed cells.

Table 1
Initial notation.

Symbol	Description
l	A logic or smart contract stored in the distributed ledger
c	Source code of logic l
b	Bytecode produced by the compilation of c
f^{co}	Compilation function that generates a bytecode from a source code
d	Deployment address of a given bytecode
r	Request from a user willing to run a deployed logic or smart contract
p	Request parameters to run a smart contract

Req4. The framework must provide a higher security scenario and network that ensures that the logic contained in it is authentic. For this, the deployment of logic to the global space, and execution of logic in the global space, can only be done over authentic logic. Otherwise the user cell should be notified of a security breach and will be prevented from using it.

Req5. This requirement relates to the real-time characteristics of cyber–physical systems; these are time-sensitive systems, such that their response time affects their correct operation. It should be possible that cells trade-off the security level of the network for the execution overhead of utilizing the network. Consequently, any cell can decide to obtain faster execution rates at the cost of less security in the environment or viceversa. For this reason, the framework must be flexible to support a partial downgrading to a less secure network level when requested by cells. In our framework, higher security levels imply additional overhead; whereas lower security levels are implemented by faster framework operations.

3.2. Sample attack on deployed logic

The blockchain paradigm has revolutioned the way in which trusted transactions take place. Precisely, it allows developers to automate the execution of agreements without the intervention of intermediaries. All nodes of the network can have immediate certainty about the outcome of the transaction.

Smart contracts are the key concept lying at the heart of this technology. They are programs stored in the blockchain network (i.e., in its global ledger) and run when certain specified conditions hold. The global ledger is inmutable and contains the terms of the agreements.

Nevertheless, a number of attacks have been successfully performed on this technology as exemplified in what follows. For the sake of clarity, Table 1 advances part of the notation that will be introduced in the following section.

Smart contract deployment. Let us imagine that a given logic l is to be uploaded to the blockchain network. Any logic l is instantiated by a particular source code c. The source code is, in turn, implemented in some high-level language like Solidity. The source code has a corresponding bytecode version b that is its executable version running in the blockchain network. This association is expressed as $l = (c, b)$.

In a blockchain network such as Ethereum, when a smart contract code c has to be deployed to the network, it should be first compiled. Then, its corresponding bytecode b is generated by a compilation functionality, expressed here as $b = f^{co}(c)$, being f^{co} the compilation function. The logic of a smart contract can then be deployed in the network (what is truly deployed is its associated bytecode b). Precisely, deploying a smart contract c to the network results in that the blockchain infrastructure deploys its corresponding bytecode b at a given *address* d. As a result, any given source code is, by this construction, univocally associated to a given deployment address d where its associated bytecode b is running. For this, we need to enhance the logic specification as $l = (c, b, d)$.

Attack scenario. It is now clear that a deployed contract source-code c has a 1-to-1 relation to its derived bytecode b, and to its deployment address d.

In this context, when a cell node k wishes to execute a given logic, it submits an invocation request r containing, among other arguments, the deployment address and the corresponding invocation parameters p, such that $r = (d, p)$.

In this situation, a cyber criminal may have successfully attacked the system by either manipulating the content of the smart contract address in the advertisement dashboard; by uploading a malicious bytecode b' such that $b' \neq f^{co}(c)$; or by manipulating the advertised source code of the smart contract, among other possibilities. As result of a successful attack, the code running on the blockchain network does not anymore correspond to the expected contract bytecode b.

4. VelogCPS approach

This section describes the design of VelogCPS middleware framework. The middleware manages the requests of the participant cell nodes in order to ensure that these operate in a secure environment in what concerns the authenticity of the logic and data hosted in the blockchain network (i.e., in the global operation space). To clarify our approach, the way in which smart contract verification works is described first. Then, the system model and the used notation are presented. Lastly, the software design of the middleware is presented.

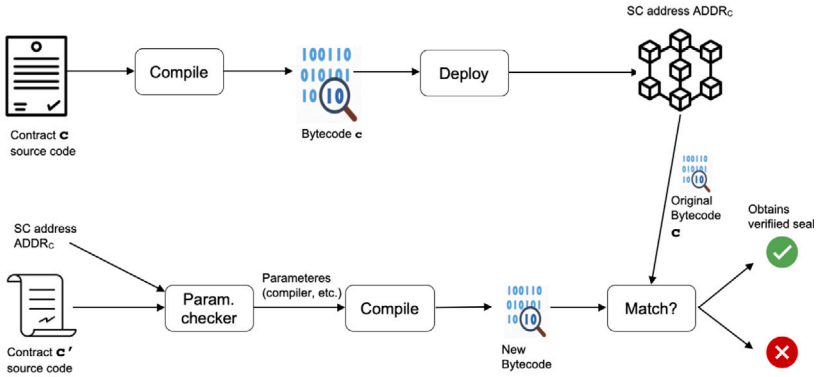


Fig. 2. Logic verification process in the blockchain: Verified SC vs non-verified.

4.1. Smart contract verification

Smart contract verification is the process that checks whether two pieces of code are, in deed, the same code. Since a source code c has its equivalent compiled executable b, it must be checked that c corresponds exactly to b when compiled; or viceversa, checking that b generates c when decompiled.

Fig. 2 shows the verification process of smart contracts. The top part of the figure presents the steps to deployment of a non-verified smart contract, starting from its source code c; whereas the bottom part shows the verification process of a given smart contract source-code c'.

Let us imagine that a cell node k found source code c' and wants to use it; c' is deployed in the network at address d in the form of executable b. If node k needs assurance about the authenticity of c', then it should be checked that c' is, in deed, the source code that generated the bytecode that is running in the blockchain network, namely b.

According to [31], smart contract verification is the process that guarantees that the blockchain bytecode corresponds to a given high-level language source code. It implies compiling the source code and checking whether it matches the bytecode stored in the blockchain network; or decompiling the bytecode and checking whether it is exactly the same as the advertised source code. In essence, and as described in Fig. 2, the verification process obtains the bytecode at the specified deployment address; then, it decompiles the bytecode; and lastly, it checks if it is exactly the same as c'. If both match, the verification is successful and it is guaranteed that $c = c'$. Then, node k can rest assured that the smart contract is authentic: its origin and provenance are legitimate ones and it has not been modified in any sense.

The verification mechanism requires that some particular information are provided to identically replicate the compilation process. Such information is, among other, the version of the compiler and optimizer (if the latter was also used), and constructor parameters, if needed.

4.2. Model and notation

The notation used in the description of the system model is shown in Table 2, that enhances the initial one provided in Table 1.

A cyber–physical system can have multiple (n) cells. Each cell i has a main server or gateway N_i that contains the functionality that allows its integration with the rest of cells that take part in the global system. Each cell i may develop a given number (m_i) of logic pieces. The set of all logic pieces of the cell is L_i, such that $L_i = (l_{i,1}, l_{i,2}, \ldots, l_{i,m_i})$. As a result, $l_{i,j}$ is the j^{th} logic developed by cell i. Each cell i can deploy a logic $l_{i,j}$ that it previously developed to the global operation space enabled.

In the blockchain, the logic pieces are realized by means of smart contracts. The specification of a logic piece $l_{i,j}$ is now enhanced and

Table 2
Notation.

Symbol	Description
$l_{i,j}$	j^{th} logic (or smart contract) produced by cell i
L_i	Set of logic pieces developed by cell i
$c_{i,j}$	Source code corresponding to $l_{i,j}$
$b_{i,j}$	Bytecode corresponding to $l_{i,j}$
$p_{i,j}$	Set of feasible invocation parameters for logic $l_{i,j}$
$u_{i,j}$	Indication of whether it is a verified contract
f^{co}	Compilation function to generate $b_{i,j}$ from $c_{i,j}$
N_i	Main node of cell i that is a participant of the global system

expressed with the following components $l_{i,j} = (c_{i,j}, b_{i,j}, d_{i,j}, u_{i,j})$ such that $c_{i,j}$ is the source code; $b_{i,j}$ is the executable code or bytecode; $d_{i,j}$ is the address associated to $b_{i,j}$ upon its deployment; and $u_{i,j}$ is a boolean value expressing whether the logic is verified or not. If the logic is verified, it means that the code has been compiled, deployed to the blockchain, and it has undergone a successful verification process afterwards that guarantees its legitimate provenance.

Any cell k can utilize and/or access any logic $l_{i,j}$. For this, the node elaborates a request containing the deployment address $d_{i,j}$ of the desired bytecode. Optionally, the request may also include additional invocation parameters p^r such that $p^r \in p_{i,j}$. Thus, a request or invocation to execute a given logic $l_{i,j}$ in the blockchain is described as $(d_{i,j}, p^r, v)$ where p^r is the parameter (sub)set of the r^{th} invocation triggered by cell k. Parameter v is a boolean value, expressing whether the logic should be a verified one or not. E.g., if v is *true*, then the code running at address $d_{i,j}$ will only be executed if such code is verified.

It is possible that a node k elaborates requests with different authenticity requirements with respect to the logic $l_{i,j}$ that it wants to use. For this, node k indicates the verification requirements of each request by setting parameter v to a proper value. A value *true* indicates that the invocation has a high authenticity assurance requirement, so that the invocation will only take place if the logic is previously successfully verified. A *false* value for parameter v indicates that there is no need to check the authenticity of the logic.

Once a logic has been deployed as a verified contract, the middleware stores it in its internal *Sealed DB*. This data base contains all logic pieces deployed to the blockchain that have been previously verified.

This system model is enabled through VelogCPS middleware (see Fig. 3), that follows a component-based software design and the principles of separation of concerns. VelogCPS supports the creation of the global operation space as well as its controlled execution flow to guarantee the usage of a secure environment in what concerns smart contract authenticity. The middleware interconnects the cells of the cyber–physical system to the global space, allowing them to verify logic, deploy it, invoke the execution of deployed logic, and filter its execution depending on its authenticity assurance parameters (i.e., of its required network security level).

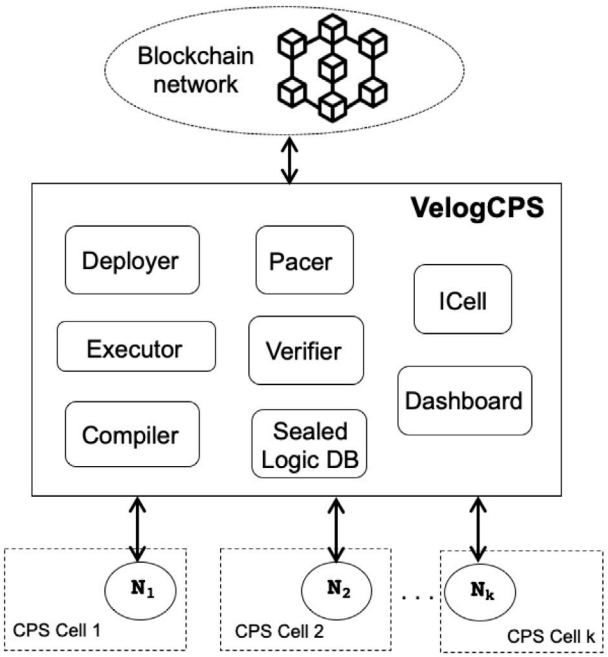


Fig. 3. Design of VelogCPS.

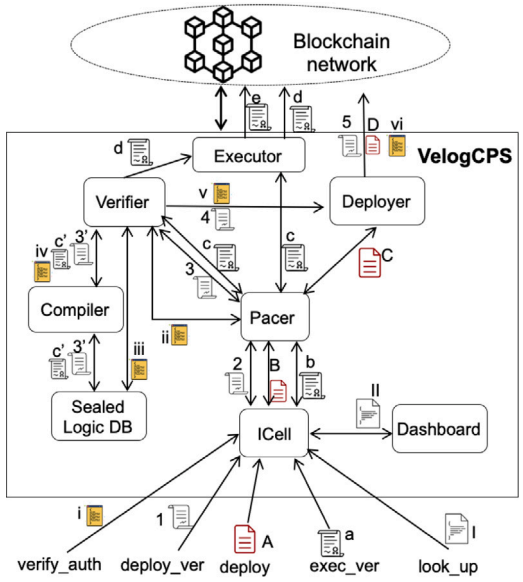


Fig. 4. Overview of interactions across the middleware components.

The main components of the middleware framework are described below:

- *ICell*. This component (*Interface to Cells*) intercepts the requests from cell nodes, providing them with the following interfacing functions: `verify_auth`, `deploy_ver`, `deploy`, `exec`, `exec_ver`, and `look_up`. They are described later in detail. This component performs the initial check on the request format and accompanying parameters provided by the nodes.
- *Pacer*. This is the core component responsible for controlling the operation flow within the middleware processes. Upon interception of a request by the *ICell* component, it forwards the request parameters to the *Pacer*. After analysing the request type, the appropriate workflow is initiated. Deployment requests are eventually handed over to the *Deployer* component (and *Verifier*, if needed). The execution of a logic is eventually passed to the *Executor* component; and verification requests are passed to the *Verifier* component. When the components finish their operation, they provide the response back to the *Pacer* that prepares a response to the cell node reflecting the final outcome of its request. The answer is circulated back to the user cell through the *ICell* component.
- *Deployer*. This component is responsible for deploying smart contract logic to the blockchain network. It accesses the distributed ledger and generates a transaction that ends with the logic being sent and stored in the global ledger.
- *Sealed DB Interface*. The middleware has a database (namely *Sealed DB*) that stores the smart contracts that have already been successfully verified in the system.
- *Verifier*. The functionality of the *Verifier* component is shown in Fig. 2. This component receives a verification request over a particular smart contract source code to be compared against a running logic. It compiles the code to generate a bytecode and compares it to the running code with decompilation. The *Verifier* makes use of the Sealed DB to store a logic when it is verified. Additionally, this could be interfaced to an external global block verification engine.
- *Executor*. This component receives requests to execute a given logic. It performs the actual invocation to the running bytecode on the blockchain.
- *Deployer*. This component contains the needed functionality to carry out the deployment of bytecode-form logic to the blockchain network.
- *Dashboard*. The middleware uses this component to make visible the available logic pieces to participating cells. A cell interested in using some logic will look into the *Dashboard* so that the cell can analyse the code and determine if the functionality provided by the logic is of interest. If so, it will be able to use it by invoking the functions that such logic offers.

4.3. Middleware functionality

The innerworkings of the middleware are explained here through the needed algorithms that illustrate its rationale and procedures. Fig. 4 shows the overall behaviour.

Let us imagine the initial situation in which a generic cell node k comes across some logic $l_{i,j}$ that is advertised in the global space. Finding a given logic is facilitated by the middleware through the `look_up` function provided by the *ICell* component; this function triggers a read in the system *dashboard* where logic is advertised. An example of logic may be a mechanism to reassign resources among its local nodes or a discount strategy for asset negotiation. The cell node checks the source code $c_{i,j}$ that corresponds to such a logic, analyses its functionality, and decides that it wants to use it.

Algorithm 1 shows the process to verify any given logic. This functionality is provided by the middleware as function `verify_auth`. The algorithm is executed upon a request with parameters $(c_{i,j}, p^r)$. The middleware intercepts this invocation request in its *ICell* component, and starts the process to ensure that this logic is authentic. First, it is important to remcall an inherent characteristic of smart contract verification: For a logic to be verified, it should have been first deployed to the network at a given address $d_{i,j}$ (its deployment address).

After an initial check over the validity of the provided parameters $p_{i,j}$, it is necessary to determine: (a) if the smart contract $c_{i,j}$ has already been deployed to the blockchain network (i.e., it is already running in some nodes); and (b) if such logic has a verification seal (i.e., it is stored in the Sealed DB). Algorithm 1 will result in an attempt to verify source code $c_{i,j}$ if it has not already been verified nor deployed. If $c_{i,j}$ is successfully verified, then the algorithm will return a success code as boolean value *true*. If the smart contract cannot be verified, then an authenticity problem is detected and a boolean value *false* will be

Algorithm 1 Verify authenticity of a logic $c_{i,j}$: function `verify_auth`

```
Require: ICell receives request r = (c_{i,j}, p^r)
 1: if p^r ∉ p_{i,j} then return error                ▷ Request format problem
 2: else
 3:     i ← 0
 4:     verified ← false
 5:     d_{i,j} ← get_deploy_addr(c_{i,j})            ▷ If logic not deployed, cannot check
 6:     if d_{i,j} == null then                       ▷ Not yet deployed; deploy & check
 7:         b_{i,j} ← compile(c_{i,j}, p^r)
 8:         d_{i,j} ← deploy(b_{i,j}, p^r)
 9:         b'_{i,j} ← get_bcode(d_{i,j})
10:         cd ← decompile(b'_{i,j}, p^r)
11:         resp ← match(c_{i,j}, cd)
12:         if resp == true then                      ▷ Verification success, store in DB
13:             q ←length(cvstore)
14:             cvstore[q] ← d_{i,j}
15:             u_{i,j} ← true
16:             return resp                           ▷ Return OK: deployed & verified
17:         else
18:             return error                          ▷ Authenticity problem detected
19:         end if
20:     else                                          ▷ If logic is deployed, look first in sealed DB
21:         while i <length(cvstore) and verified == false do
22:             if cvstore[i] == d_{i,j} then
23:                 verified ← true
24:             end if
25:             i ← i + 1
26:         end while
27:         if verified == false then                 ▷ Deployed but not in DB, verify
28:             b_{i,j} ← get_bcode(d_{i,j})
29:             cd ← decompile(b_{i,j}, p^r)
30:             resp ← match(c_{i,j}, cd)
31:             if resp == true then                  ▷ Verification success, store in DB
32:                 q ←length(cvstore)
33:                 cvstore[q] ← d_{i,j}
34:                 u_{i,j} ← true
35:                 return resp                       ▷ Return OK: verified
36:             else
37:                 return error                      ▷ Authenticity problem detected
38:             end if
39:         else
40:             return true                           ▷ Return OK: deployed & verified
41:         end if
42:     end if
43: end if
```

returned. Function `get_deploy_addr`($c_{i,j}$) (shown in line 5) is used that returns the deployment address of the running bytecode or *null* if it has not yet been deployed.

From this moment on, one of the following situations may be encountered:

- The logic is *not yet deployed*. In this case, the logic is first compiled (function `compile`) to obtain the corresponding bytecode. After, it is deployed to the infrastructure (with function `deploy`). Once deployed, the logic can be verified. For this, the bytecode of the running logic is again retrieved with function `get_bcode` of line 9. Then, the decompilation is done (with function `decompile`) using the same compilation parameters p^r, e.g., compiler and optimizer versions should be used. The running code is authentic if the original source code and the source code coming from the decompilation of the running code are the same. The latter is checked with function `match`. Once matched, the verified smart contract is stored in the array of verified logic *cvstore* that maps the content of the Sealed DB component. In this case, function `deploy_ver` returns `true` as an indication that source code $c_{i,j}$ deployed at address $b_{i,j}$ is authentic, meaning that it is a verified smart contract. However, if function `match` would return value *false*, then it means that a problem was detected such that $c_{i,j}$ deployed at address $b_{i,j}$ could not pass the verification. As a result, the smart contract is not a verified one.
- The logic is *already deployed*. In this situation (starting at line 20), the logic is checked to see if it was also previously verified by finding it in the Sealed DB component of the middleware. Algorithm 1 illustrates this search as an iteration on the verified contracts *cvstore* that corresponds to the sealed data base component.
  If the logic is not in the Sealed DB, it means that it is not verified although it is already deployed. Then, the logic undergoes a verification process as illustrated in lines 27 through 38.

Algorithm 2 shows how the middleware handles the request from a generic node k to *run* a piece of logic. The source code of logic pieces is visible to the cell nodes; and this also includes their internal coding, function declarations, and function code. Then, any cell node can decide which function (and with what parameters) it wants to invoke.

In this context, when a cell node k finds an advertised logic $c_{i,j}$ that it wants to use, the node may trigger a request to execute it. This implies that node k can invoke the execution of any of the externally visible functions.

In the request, node k may include not only the function reference but also any other invocation parameters as expressed in p^r. Then, the request of node k has the form $r = (d_{i,j}, c_{i,j}, p^r, v)$ that contains the logic address, the invocation parameters, and a boolean value v. Parameter v expresses whether the logic should be a verified one or not. That is, if v is *true*, then the code running at address $d_{i,j}$ will only be executed if it has been verified, i.e., if it has a verification seal. By the same token, if v is *false*, the logic can be executed in a straightforward manner, i.e., without checking a priori if the logic running at $d_{i,j}$ is a verified one or not.

It should be noted that to verify the authenticity of a given contract, the source code should be passed together with the deployment address. Verification will check that there is no attack that has changed the running executable nor the advertised source code. This is done by function `verify_auth` explained in Algorithm 1.

Algorithm 3 illustrates the deployment process carried out by the middleware when triggered by the reception of an invocation to function `deploy_ver`. The middleware supports functions `deploy` and `deploy_ver`. The only difference between them is their verification awareness. Function `deploy` deploys any bytecode logic to the blockchain; whereas `deploy_ver` deploys with authenticity awareness. The latter checks if the logic has not been previously verified. If not verified yet, it is first verified inside `deploy_ver` function.

Precisely, Algorithm 3 shows the verification-aware process, as indicated by the boolean parameter v. The algorithm receives parameters $c_{i,j}$ that is the logic source-code to be deployed; and p^r or set of parameters for the deployment operation.

In this process, the compilation of $c_{i,j}$ is performed with the `compile` function. The compilation parameters are passed in p^r. Then, the process checks for the verification requirement v. If the deployment needs to be done for a verified logic, then function `verify_auth` is invoked. The latter continues the process as indicated in Algorithm 1. If `verify_auth` returns an error, it means that an authenticity problem was encountered, so the deployment does not proceed further and propagates this notification to the requesting cell node.

In case that the deployment does not require to work with verified logic (v is *false*), then this function downgrades to a less strict security level by performing a simple deployment using function `deploy`.

Algorithm 2 Execute logic: function `exec_ver`

Require: N_k requests to run $l_{i,j}$ with request $r \leftarrow (d_{i,j}, c_{i,j}, p^r, v)$

$r \leftarrow (d_{i,j}, p^r, v)$
if $v ==$ false **then**
  execute(r) ▷ Run standard logic
else
  $i \leftarrow 0$
  $verified \leftarrow$ false
  while $i <$length($cvstore$) and $verified == false$ **do**
    if $cvstore[i] == d_{i,j}$ **then**
      $verified \leftarrow$ true
    end if
    $i \leftarrow i + 1$
  end while
  if $verified == true$ **then**
    execute(r) ▷ Logic is verified: run it
  else ▷ Logic is not verified: verify before running
    $b_{i,j} \leftarrow$ get_bcode($d_{i,j}$)
    $cd \leftarrow$ decompile($b_{i,j}, p^r$)
    $resp \leftarrow$ match($c_{i,j}, cd$)
    if $resp ==$ true **then** ▷ Logic is verified: store and run
      $q \leftarrow$length(cvstore)
      cvstore[q] $\leftarrow d_{i,j}$
      $u_{i,j} \leftarrow$ true ▷ Just verified
      execute(r)
    else
      return error ▷ Authenticity problem detected
    end if
  end if
end if

Algorithm 3 Deployment of logic: function `deploy_ver`

Require: Node i issues deployment request $r \leftarrow (c_{i,j}, p^r, v)$

if $v ==$true **then**
  if verify_auth($c_{i,j}, p^r$) == true **then**
    return true ▷ Verified deployment OK
  else
    return error ▷ Authenticity problem detected
  end if
else
  $b_{i,j} \leftarrow$deploy($c_{i,j}, p^r$)
  if $b_{i,j} ==$ null **then**
    return error ▷ Deployment error
  else
    return true ▷ Deployment OK
  end if
end if

5. Validation

This section describes the validation of VelogCPS framework. The validation is based on the possibility of its realization and on the determination of the actual overhead as being appropriate for time-sensitive domains. Precisely, the questions answered in the validation are the following. Firstly, is it feasible to realize an intermediation framework in practice that provides a global operation space with assurance over the authenticity of the hosted logic? Secondly, is it possible to provide such an assurance to the participant cell nodes in an online manner and transparently? In the third place, is the framework suitable for time-sensitive domains? For the latter, we determine its temporal behaviour and incurred overhead. In the validation, the temporal overhead is analysed for different situations and middleware operations: communication from the cell nodes to public blockchain; the middleware operation added to the heavy operations of the blockchain such as the consensus protocol; the time required to deploy logic to the global space; and the time taken to execute a logic in the distributed ledger triggered by a requesting cell node.

5.1. Experimental setting

The experimental setting reproduces a global system at a reduced scale containing cell nodes and an actual implementation of VelogCPS with its constituent components. The cells run on actual hardware equipment to collect realistic measurements. Experiments are carried out with cell node logic running on Intel i9 13900, with 36MB LGA, 24 cores, 32 logical cores, 32 GB RAM, and running at 2 GHz. Acceleration is achieved with NVIDIA GeForce GTX 1650 with 896 cores in CUDA.

Cell nodes are connected to a blockchain network with Polygon [32] technology. Each cell node runs a blockchain client that is *geth* version 1.11.5-stable-a38f4108 virtual machine software. Logic is implemented solely in Solidity with compiler version 0.8.18+commit.87f61d96 in the *Deployer* component.

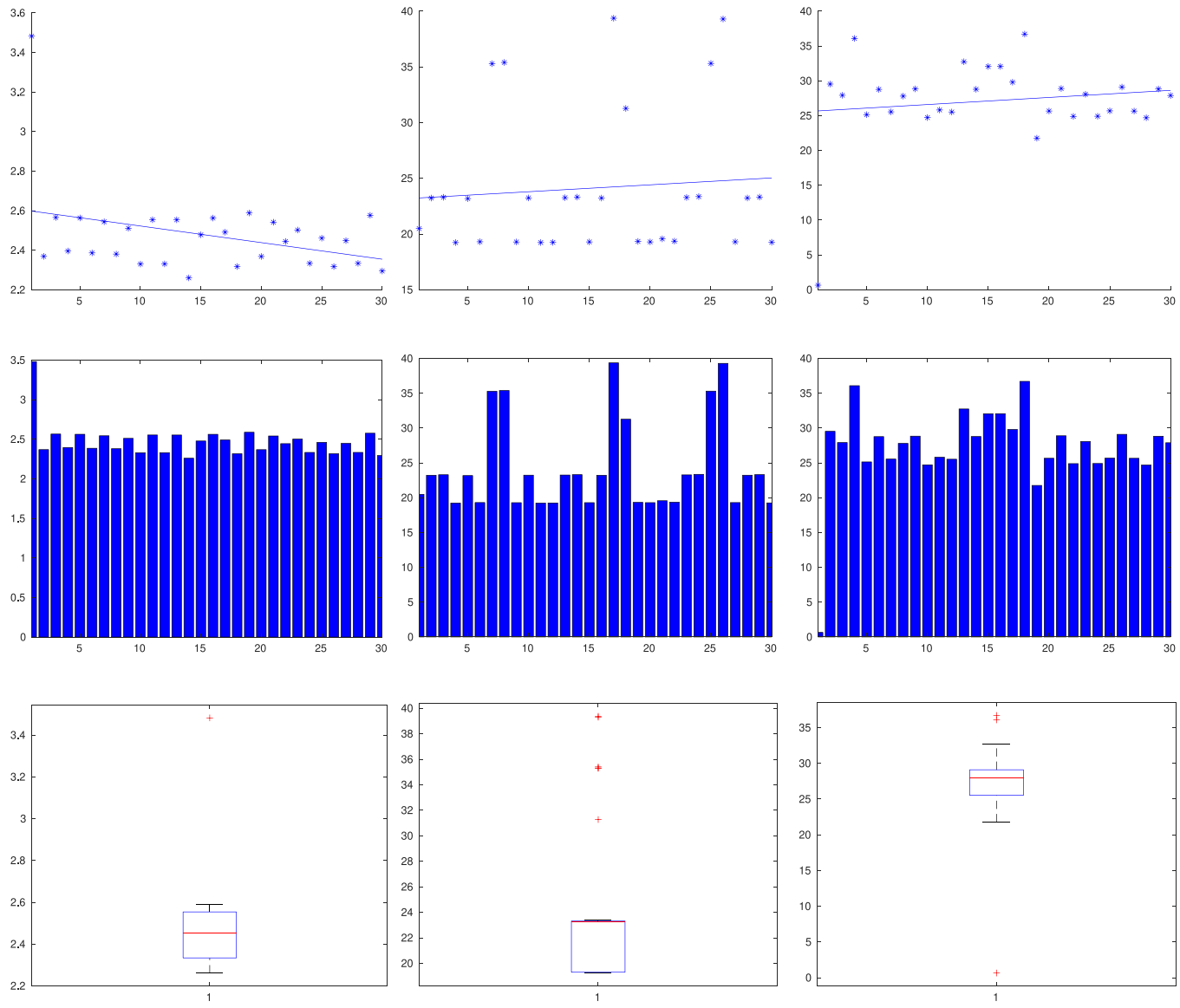
5.2. Results

A set of exhaustive experiments have been performed to validate our solution on the basis of an actual implementation of the middleware framework. We provide a feasibility validation and a determination of the framework overhead that proves its applicability in the context of time-sensitive systems. The obtained temporal cost show s the overhead of the process of providing a safe execution environment in the global operation space.

Determination of the logic deployment overhead. The first experiments present the overhead of the authenticity assurance support offered by the middleware. Precisely, Fig. 5 compiles the time taken to deploy logic pieces to the global space for the relevant situations. The first situation concerns exclusively the bare deployment of a logic. Second situation corresponds to a full deployment of a logic. Lastly, the third situation concerns the full deployment of verified logic. To ease their comparison, the temporal profiling for each situation is provided in one column. Table 6 summarizes the temporal results for this set of experiments.

Fig. 5(a) illustrates the first situation and plots the delay experienced by requesting nodes upon logic deployment. The experiment measures the time taken by the communication from the cell node to the blockchain where it deploys a given logic. This is the base situation corresponding to the bare deployment time of logic without waiting for the completion of the consensus protocol. At such a point in time, there is not a full assurance over the eventual annotation of the logic in the distributed global ledger. We instrumented the experiments to collect the bare deployment time. This serves as a comparison basis because we are not sure that the block containing the logic has been validated yet by the peer nodes. As a result, the logic cannot be accounted as a full deployment since, in some situation, it could happen that the peer nodes do not validate the corresponding deployment transaction.

The temporal overhead of full deployment of the logic is presented in Fig. 5(b). The temporal measurements include the time taken by the blockchain infrastructure to reach consensus. Consensus is reached when receiving the acknowledgement of peer nodes confirming the correctness of the mined block that stores the deployed logic. As a result, this figure shows the overhead of the full deployment of logic to the global space because it includes the confirmation from peer nodes that the logic has been properly deployed and propagated in the network. On the one hand, it is observed the high cost of consensus in public networks reaching $1000x$ with respect to the bare deployment. This measure can be reduced in private and hybrid networks; although these do not constitute a worst-case scenario. On the other hand, it is observed that the mean time of full deployment is 24.307 s. This time is below the minute range which is appropriate for time-sensitive systems



(a) Bare deployment - comparison baseline

(b) Full deployment with $v = false$

(c) Full deployment with $v = true$

Fig. 5. Deployment overhead — scenario with unverified logic (execution time in s).

with soft real-time requirements and quality of service management. This is the case of many types of cyber–physical systems present in, e.g., smart cities.

Fig. 5(c) shows the deployment cost with verification, i.e., the verification-aware deployment. In this case, after the deployment has been fully finished (which includes the completion of the consensus), the logic is guaranteed to be verified. The middleware verifier uses Polygonscan functionality. As this is the highest assurance level offered by the middleware, it exhibits a higher overhead than for the previous scenarios. The middleware uses a Polygon network with a PoS [33] consensus protocol. This consensus mechanism yields a reduced temporal cost and more efficient resource consumption to perform exhaustive transaction execution in the blockchain network. Still, the authenticity guarantee has a cost for the first verification of the logic. Nevertheless, as verification of a given logic is a one hop process, this cost pays off in a short utilization term in favour of a guarantee over attacks to the logic authenticity. Table 3 summarizes the execution time overhead of the different analysed deployment types.

Table 3
Overhead of deployment of verified logic (in seconds): Summary of Fig. 5.

Value	Bare deploy.	Full deploy.	Highest assurance
Mean	2.483	24.307	27.133
Median	2.461	23.233	27.929
Max	3.482	39.372	36.715
Min	2.261	19.236	0.656

Determination of the overhead of the access to verified logic. Fig. 6 shows the execution time of the for the case of logic that is already verified. It can be observed that the cost–benefit of using VelogCPS to access verified logic is excellent after the first deployment and verification of such a logic. In this situation, the cell requests to access a given logic are managed by middleware by consulting the *Sealed DB* component. This component has been developed with MongoDB technology. The temporal overhead is reduced to an average value of 39 ms with a median of 17 ms (see Table 4).

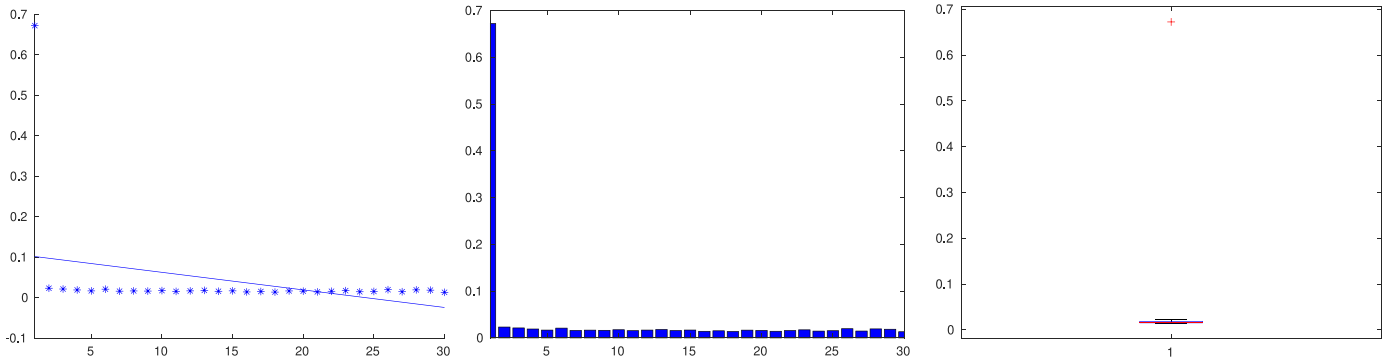
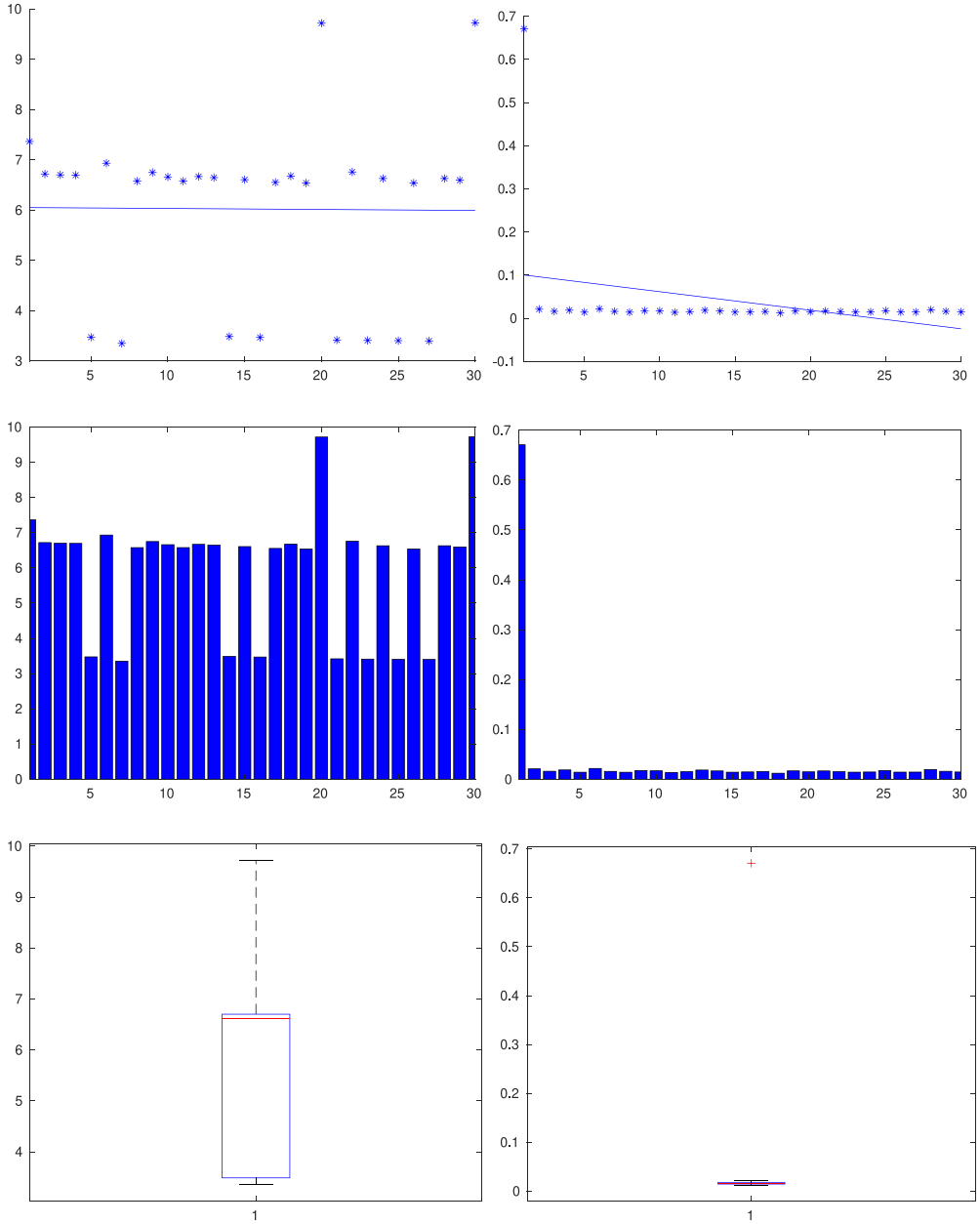
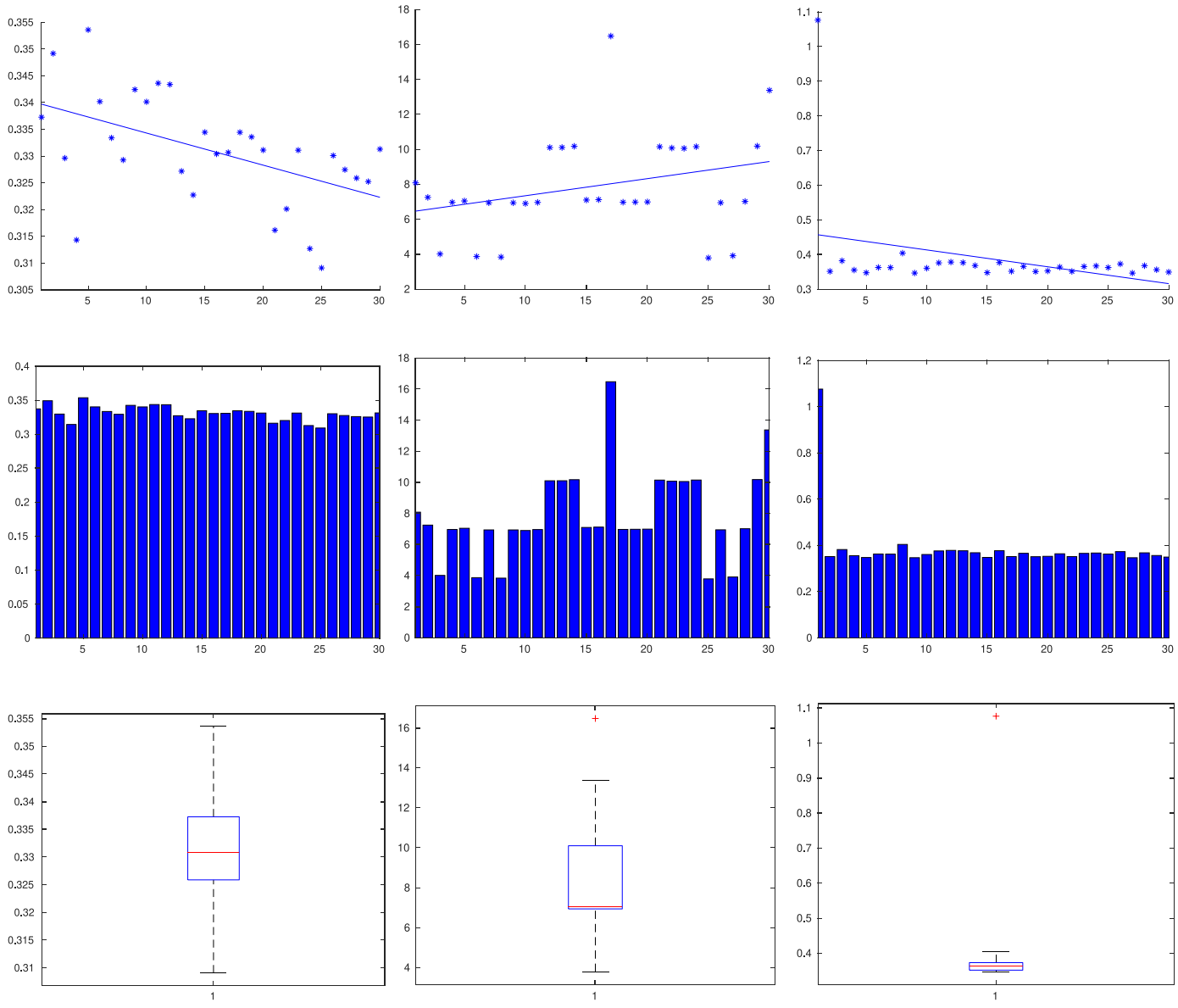


Fig. 6. Deployment overhead — scenario with verified logic (execution time in s).



(**a**) Unverified logic (**b**) Verified logic

Fig. 7. Verification process overhead (execution time in s).



(a) $v = false$ (b) $v = true$; unverified logic (c) $v = true$ with verified logic

Fig. 8. Overhead of logic execution (time in s).

Table 4
Overhead summary (in seconds) for deployment results of Fig. 6.

Value	Time
Mean	0.039
Median	0.017
Max	0.672
Min	0.013

Table 5
Overhead summary of verification process of Fig. 7.

Column	Unverified logic	Verified logic
Mean	5.896	0.038
Median	6.606	0.016
Max	9.718	0.671
Min	3.352	0.012

Determination of the overhead of the verification process. The performance of the verification process is shown in Fig. 7 for the relevant scenarios. It is also summarized in Table 5. Fig. 7(a) shows the cost of verification over logic that is already deployed in the network but not previously verified. In this scenario, the logic has to be verified so it is needed to make a service request to the network block verifier and a wait for its response. It is observed that an overhead of 5.896 s is obtained which is appropriate for a wide range of time-sensitive systems. Fig. 7(b) presents the cost of verification for logic that is already deployed and verified. In the first case, the verification is performed directly by the *Verifier* component that has a built-in bridge to the blockchain network verifier. The advantage of using VelogCPS in the second scenario is evidenced because it translates to a read operation or query over the *Sealed DB* component. The latter shows very fast response times in the order of 0.645% in average with respect to the previous scenarios.

Determination of the overhead of executing logic in the global space. The last set of experiments analyse the performance of the

Table 6
Summary for logic execution process shown in Fig. 8.

Column	No verif.	Verification (unverified logic)	Verification (verified logic)
Mean	0.331	7.880	0.387
Median	0.331	7.028	0.363
Max	0.364	16.474	1.076
Min	0.309	3.784	0.347

process of executing of a given logic in the global operation space. In this case, VelogCPS intermediates the invocation of cells to execute functions of smart contract logic that is running in the blockchain network. Fig. 8 shows the obtained results for different relevant scenarios. Firstly, for the case when the logic is already deployed shown in Fig. 8(a). Secondly, for when the logic is deployed but not verified that is shown in Fig. 8(b). In the third place, it shows the overhead of the framework in situations when the logic is deployed and verified as shown in 8(c).

In the first scenario, only the execution time of the logic in the blockchain is measured. We see that the temporal cost is quite efficient compared to the rest of values, with a mean of 331 ms, with a reduction of 23× as compared to the less favourable scenario.

Fig. 8(b) shows that the verification cost is in the order of a few seconds around a mean value of 7 s. Nevertheless, this overhead comes at the benefit of providing the highest assurance and certainty about the authenticity of the logic after its verification.

In the third scenario presented in Fig. 8(c), it can be observed that the benefit of using VelogCPS that is highest after the first verification of the logic has been performed. After this initial invocation, the overhead of the execution becomes comparable to that of executing an unverified logic. As a matter of fact, times are kept near to constant around the average value of 387 ms, with a mean value of 363 ms, that proves the stability in the middleware operation.

6. Conclusions

This paper has presented the design of a framework to support the safe interaction of connected cells in cyber–physical systems. The solution relies on a model of shared logic, enabled by blockchain technology and managed by a middleware. The intermediation of the middleware guarantees cells to use solely authentic logic in the global operation space. This is done by including the usage of on-the-fly verification supported by block verifier services.

The solution offers the possibility of exchanging performance and service cost by the security over the authenticity of the used logic. This has been shown empirically in the validation. It has been proved that the middleware is suitable to provide a safe execution scenario for connected cells, guaranteeing that the logic in such a space is verified or reporting the authenticity risk to the user cells. It has been evidenced that the highest cost is achieved for the highest assurance levels when deploying logic. Also, it has been shown that this high cost is mostly influenced by the consensus mechanisms of the blockchain network. The verification of the logic lies in the range of few seconds (in the order of 7 for the worst case scenarios). This has been shown to pay off as it is associated to the initial verification, creating right after an efficient execution scenario where authentic logic is guaranteed to be used.

CRediT authorship contribution statement

Marisol García-Valls: Writing – review & editing, Writing – original draft, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Alejandro M. Chirivella-Ciruelos:** Validation, Software, Methodology, Data curation.

Declaration of competing interest

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

Data availability

The authors do not have permission to share data.

Acknowledgements

This research was funded by project "*Design of services for resilient and real-time execution of social dispersed computing applications in cyber–physical domains*" funded by Generalitat Valenciana (Conselleria de Innovación, Universidades, Ciencia *y* Sociedad Digital), Spain, under Grant No. AICO/2021/138; and by project "*Evolution of the radio access network towards 6G for massive and low-latency services*" funded by ERDF *A way of making Europe* and Ministerio de Ciencia, Innovación y Universidades of Spain MCIN/AEI/10.13039/501100011033 under Grant no. PID2021-123168NB-I00.

References

[1] S. Ølnes, J. Ubacht, M. Janssen, Blockchain in government: Benefits and implications of distributed ledger technology for information sharing, Gov. Inf. Q. 34 (3) (2017) 355–364.
[2] L. Tseng, X. Yao, S. Otoum, M. Aloqaily, Y. Jararweh, Blockchain-based database in an IoT environment: Challenges, opportunities, and analysis, 23, 2020, pp. 2151–2165.
[3] T. Hyla, J. Pejaś, eHealth integrity model based on permissioned blockchain, Future Internet 11 (3) (2019) 76.
[4] V. Merlo, G. Pio, F. Giusto, M. Bilancia, On the exploitation of the blockchain technology in the healthcare sector: A systematic review, Expert Syst. Appl. 213 (2023) 118897.
[5] H.R. Rewatkar, D. Agarwal, A. Khandelwal, S. Upadhyay, Decentralized voting application using blockchain, in: 2021 10th IEEE International Conference on Communication Systems and Network Technologies, CSNT, 2021, pp. 735–739.
[6] M. von Rosing, S. White, F. Cummins, H. de Man, Business Process Model and notation—BPMN, in: M. von Rosing, A.-W. Scheer, H. von Scheel (Eds.), The Complete Business Process Handbook, Morgan Kaufmann, Boston, 2015, pp. 433–457.
[7] M. Gómez Gélvez, Explaining the DAO exploit for beginners in Solidity, 2016, Last accessed June 2023.
[8] Worldcoin, What's Ethereum 2.0. A complete guide, 2023.
[9] S. Palladino, The Parity Wallet Hack explained, 2017, Last accessed June 2023.
[10] M. García-Valls, A.M. Chirivella-Ciruelos, Provenance verification of smart contracts: Analysing the cost of ensuring authenticity over the logic hosted in blockchain networks, Information 15 (1) (2024).
[11] Hyperledger fabric, 2023, Last accessed May 2023.
[12] Q. Lin, H. Wang, X. Pei, J. Wang, Food safety traceability system based on blockchain and EPCISd, IEEE Access 7 (2019) 20698–20707.
[13] M. García-Valls, A.M. Chirivella-Ciruelos, CoTwin: Collaborative improvement of digital twins enabled by blockchain, Future Gener. Comput. Syst. 157 (2024) 408–421.
[14] G. Malik, K. Parasrampuria, S.P. Reddy, S. Shah, Blockchain based identity verification model, in: 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking, ViTECoN, 2019, pp. 1–6.
[15] H. Gaikwad, N. D'Souza, R. Gupta, A.K. Tripathy, A blockchain-based verification system for academic certificates, in: 2021 International Conference on System, Computation, Automation and Networking, ICSCAN, 2021, pp. 1–6.
[16] R.S. Lamkoti, D. Maji, H. Shetty, B. Gondhalekar, Certificate verification using blockchain and generation of transcript, Int. J. Eng. Res. Technol. 10 (2021).
[17] A.Y. Bykovsky, N.A. Vasiliev, Data verification in the agent, combining blockchain and quantum keys by means of multiple-valued logic, Appl. Syst. Innov. 6 (2) (2023).
[18] T.K. Agrawal, V. Kumar, R. Pal, L. Wang, Y. Chen, Blockchain-based framework for supply chain traceability: A case example of textile and clothing industry, Comput. Ind. Eng. 154 (2021) 107130.
[19] Tenderly, 2023, Last accessed on June 2023.
[20] Sourcify.eth, 2023, Last accessed on June 2023.
[21] Verifying a smart contract, 2024, Last accessed on January 2024.
[22] Etherscan, 2023, Last accessed on June 2023.
[23] P. Ma, N. He, Y. Huang, H. Wang, X. Luo, Abusing the Ethereum smart contract verification services for fun and profit, 2023, CoRR, abs/2307.00549.

[24] A.M. Chirivella-Ciruelos, M. García-Valls, Automating the verification of smart contracts in blockchain networks for improving security, in: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications, SEAA, 2023.
[25] H. network, Ethereum Development Environment for professionals, 2023, Last accessed on June 2023.
[26] M. Almakhour, L. Sliman, A.E. Samhat, A. Mellouk, Verification of smart contracts: A survey, Pervasive Mob. Comput. 67 (2020) 101227.
[27] D. Marijan, C. Lal, Blockchain verification and validation: Techniques, challenges, and research directions, Comp. Sci. Rev. 45 (2022) 100492.
[28] D. Wang, P. Wang, C. Wang, Efficient multi-factor user authentication protocol with forward secrecy for real-time data access in WSNs, ACM Trans. Cyber-Phys. Syst. 4 (3) (2020).
[29] Q. Wang, D. Wang, C. Cheng, D. He, Quantum2FA: Efficient quantum-resistant two-factor authentication scheme for mobile devices, IEEE Trans. Dependable Secure Comput. 20 (1) (2023) 193–208.
[30] Q. Mei, H. Xiong, Y.-C. Chen, C.-M. Chen, Blockchain-enabled privacy-preserving authentication mechanism for transportation CPS with cloud-edge computing, IEEE Trans. Eng. Manage. (2022) 1–12.
[31] S. Marx, Verifying contract source code, 2018, Last accessed June 2023.
[32] Matic RPC – Polygonscan Mumbai, 2023, Last accessed June 2023.
[33] H. Dou, L. Yin, Y. Lu, J. Xu, A probabilistic Proof-of-Stake protocol with fast confirmation, J. Inf. Secur. Appl. 68 (2022) 103268.