information

MDPI

Article

Provenance Verification of Smart Contracts: Analysing the Cost of Ensuring Authenticity over the Logic Hosted in Blockchain Networks

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

Departamento de Comunicaciones, Universitat Politècnica de València, 46022 Valencia, Spain; alchici@dcom.upv.es

* Correspondence: mgvalls@dcom.upv.es

† These authors contributed equally to this work.

Abstract: The lack of sufficient guarantee about the authenticity of running smart contracts is a major entry barrier to blockchain networks. By *authenticity*,we refer to the smart contract ownership or provenance; this implies perfect matching between a published source-code and the corresponding running version of a given smart contract. Block verifiers are services that check the provenance authenticity of the logic contained in blockchain networks. Nevertheless, as a block verifier is an external verification service, it consumes time to use it; and the derived overhead may not comply with temporal requirements of time-sensitive domains like *cyber-physical systems*. Such systems require that the temporal cost of using external services is assessed prior to the final system deployment. To the best of our knowledge, there are no previous contributions on the determination of the temporal cost of the smart-contract provenance verification process. This paper presents the design and implementation of a middleware that assesses the temporal overhead of accessing the verification services; the middleware is hosted in the global ledger and runs the verification services over large sets of smart contracts. Our contribution is validated by providing an implementation on a real blockchain network, employing actual smart contract verifier logic, and analysing the temporal behavior of the overall system operations to comply with the time-sensitive requirements of cyber-physical systems.

Keywords: blockchain; provenance; verification; smart contract; cyber-physical systems; middleware; time-sensitive system; authenticity; security

Citation: García-Valls, M.; Chirivella-Ciruelos, A.M. Provenance Verification of Smart Contracts: Analysing the Cost of Ensuring Authenticity over the Logic Hosted in Blockchain Networks. *Information* **2024**, *15*, 24. https://doi.org/10.3390/info15010024

Academic Editor: Tieling Zhang

Received: 30 November 2023
Revised: 26 December 2023
Accepted: 30 December 2023
Published: 31 December 2023

1. Introduction

The use of blockchain technology as a way to execute distributed transactions and contracts has been growing steadily, with impressive numbers reported since its burst. The reason behind its popularity is the support of the execution of a special form of contract (namely, a smart contract) designed to avoid the presence of trusted third parties. This yields a decentralized trust mechanism that has increased interest in enabling technologies.

As shown in [1], in the first quarter of 2022, 1.45 million contracts were created only in the Ethereum blockchain network; with 329 K active addresses in the ecosystem. The Ethereum network's revenue was approximately over 1.679 billion dollars (834,874 Ether currency—ETH); over 728 K ETH were burnt in the first quarter of 2022 (amounting to 1.671 billion, aproximately).

Smart contract (SC) is the term used to designate the contract executed in a blockchain network. As referred to by [2] in his seminal work (and later elaborated in [3,4]), a *smart contract* executes the terms of a contract, as it describes a transaction protocol to execute those terms. A smart contract is a program designed to run automatically in the blockchain network without the intervention of a trusted third party, producing a result that has effects on the global ledger.

Applications that use blockchain technology face a number of security threats that may result in cyber attacks like the one of DAO [5], which yielded an extraordinary economic

loss. Another threat is related to the *authenticity* of running contracts. As smart contracts are advertised in the network, users can see them, and decide to use them; this implies undertaking a transaction based on the information claimed by the contract.

A key characteristic of blockchain networks is that it is not possible to ensure that an advertised smart contract code is the same as the code running on the blockchain unless it is checked. The reason for this is as follows. Unlike the code of advertised contracts, the code running on the network is (naturally) not source code, but bytecode; bytecode is not human-readable.

In order to adhere to the requirement of avoiding the usage of trusted third parties, blockchain technology supports that users and other developers are able to verify the source code of a smart contract. Verification is a technique to ensure identity in this respect: the code running in the blockchain corresponds uniquely to a given advertised smart contract source-code.

Verification of smart contracts has been undertaken mostly from the point of view of source code verification employing formal techniques and methods. Other approaches have also identified this issue in a somehow different way. For example, Ref. [6] focuses on the cloud as a storage center, in combination with blockchain, and uses physically uncloneable functions and fingerprint biometrics for secure data transmission. Overall, there are a number of works on the provenance of data records enabled by blockchain. These works have targeted the provisioning of a traceable structure to store data that are linked to a particular producer. Such solutions are mostly engineering developments naturally supported by the nature and design of blockchain networks. Surprisingly, the provenance verification of running contracts has seldom received attention by the research community.

We contribute an analysis of the life cycle of smart contracts in relation to their authenticity assurance or provenance verification. For this, we list the set of phases undergone by smart contracts; we identify the set of steps and interactions undertaken by the network to verify the authenticity of smart contracts; and we explore a selected set of main technology alternatives that offer verification facilities regarding smart contract provenance authenticity in blockchain networks. To the best of our knowledge, there are no previous works that focus on the determination of the temporal cost of the provenance verification process. However, this process is key in critical domains like *cyber-physical systems* (CPS), where the system needs a priori certainty about the temporal costs and timeliness of its operations and main processes. The verification of smart contracts has a cost that may not always be affordable in all application domains. It must be considered that current verification platforms tend to be hosted in remote servers, which makes them critical resources. In order to derive the temporal overhead, we analyse the performance of contract verification for different types of smart contracts and in different scenarios. The determination of the temporal cost is performed through the designed middleware that automates the extraction of the temporal cost of the verification.

The structure of this paper is the following. Section 2 provides the baseline for our contribution by explaining the verification types, paving the way to understanding the characteristics of smart contract provenance verification in blockchain technology. Section 3 describes our contribution; initially, we provide a deep dive into the life-cycle of smart contracts; later, we describe the architecture of the contributed middleware by explaining its constituent components; lastly in this section, we explain the integration of the middleware architecture to different underlying block-verifier alternatives. Section 4 describes the implementation of the middleware over an actual blockchain network; the experimental setting is explained and includes a description of different scenarios; and it employs different smart contract characterisitics to collect representative temporal results. Eventually, Section 5 presents the conclusions of this work.

2. Background

Since the presence of any defect (e.g., code bugs and vulnerabilities) may yield great economic losses, it is of paramount importance to ensure that a smart contract is correct

and functions as expected and intended. Failing to do this is undesirable, as it derives trust erosion in blockchain technology. For this, verifying each smart contract is a first step in the right direction.

There are two approaches to target the verification of smart contracts. This section lays down the different perspectives to the verification of smart contracts; it concludes the particular type of verification that this paper addresses that is related to provenance verification.

2.1. Verification

Verification [7] is the process of establishing the truth, accuracy, or validity of something. In computer science and engineering, *system verification* is the process to check the correctness of the system itself; it can be applied to its constituent parts, e.g., a component, an element, a service, a task, a requirement, a property, etc. The process of verification is essential in critical systems, where a number of different techniques have been developed to ensure that the expected properties are met and that, overall, the system behavior is the expected one. Overall, verification techniques are broadly classified into formal, informal, static, and dynamic.

Formal verification. Formal verification relies on the use of mathematical proofs to check the correctness of a system. The system is correct if it meets a given specification. Formal verification employs *formal languages* that allow the engineer to build a formal or mathematical model of a system which allows it to be later checked for correctness. This gave rise to *model checking*, as known as *property checking*, which is a method applied to check whether a system model meets a given specification. This process formulates a problem using some [8] logic with the goal of checking whether a particular model (namely, structure) satisfies a particular logical formula.

Informal verification. This techniques relies on subjective reasoning that is typically driven by humans.

Static verification. Static verification focuses on the analysis of a source code. Unlike a program or running code, the source code is a static asset. Therefore, static verification checks the structure of the code for syntactic correction and adherence to the norms and coding standards for a particular criticality level. Inspection of the code can check for coding conventions, bad practices exhibiting prohibited coding patterns, calculation of software metrics, or formal verification to detect, e.g., cycles in function invocations, failures in stop conditions, etc. Some articular examples of items that are often checked: a variable must have been defined before being used; or if a function declares to return an integer value, no other values (e.g., real or boolean values) should be returned.

Dynamic verification. This process is performed during the execution of the program, i.e., over the running software. Dynamic verification is often called *testing*. It consists of actively checking the behavior of the code to find errors caused by some activity or by the repetitive execution of an activity or a set of activities (i.e., stress test). Testing can be applied to different scopes. The smallest scope includes a function or entity/class (namely unit test); larger scopes such as groups of classes and modules; the whole system, which also comprises integration tests with more than one module; or formal tests to define acceptance (functional or non-functional—e.g., performance).

It is also interesting to note that software verification is different from software validation. Verification essentially searches for the answer to the question of whether the system is built correctly; this checks whether the system conforms to its specification. The validation strives to answer the question whether the system is right; thus, it checks the output system against its requirements, as these express the needs and desires of whoever paid for it.

2.2. Source-Code Verification

On the one hand, and since smart contracts are programs implemented through the source code, the traditional path of formal verification techniques has been extended to cover these autoexecutable programs. These techniques have been applied to verify particular properties, to detect anomalies like code errors related to reentrancy, or to detect

potential attacks that exploit vulnerabilities or code bugs. On the other hand, and since blockchain technology builds complex ecosystems that require extensive tool chains, it is also possible to target verification focusing on the architectural and performance aspects of the blockchain network. Such aspects concern the interactions among network participants like nodes, users, smart contracts, transactions, the glogal ledger, or verifiers, among others.

A prolific community on formal methods existed long before the burst of blockchain. Therefore, there is a large number of works on formal methods that have been extended in recent years regarding particular improvements to existing techniques such as Petri Nets, SMT, etc., and applied to the verification of the source code of smart contracts.

This way, we find verification works like [9], which proposes a technique to verify the smart contracts running in the Ethereum virtual machine (EVM) at a bytecode level. Also, abstract interpretation and symbolic model checking has been used in [10] to verify correctness and fairness in smart contracts. Other techniques such as Coloured Petri Nets (CPN) have been also employed. For example, Ref. [11] describes a formal verification method to check smart contracts written in Solidity to verify particular properties of the contracts; or [12] that applies CPNs to check reentrancy attacks in Solidity smart contracts. Other techniques for a static analysis through symbolic execution were explored in [13]; here, authors focus on analysing reentrancy through a static analysis tool that employs symbolic execution with Z3 SMT (Satisfiability Modulo Theory) solvers.

2.3. Smart Contract Verification through Block Verifiers

Verification of smart contracts in blockchain is highly related to ensuring the *authenticity* of the published source code. This involves a different, fresher perspective that complements the traditional one presented above. In this context, the objective is to guarantee that a given published smart contract source-code corresponds exactly to a particular bytecode that is running in the network. Often, the needed smart contract verification is provided in a central remote server through the compilation and decompilation of code (source and bytecode). These are *block verifiers*; and should not be confused with block validators. Block validators are well-known entity types in a blockchain network, as they perform the validation of the correctness of the block-mining process.

A *block verifier* is a service that determines whether the source code of an executable version of a smart contract is exactly the same as a given (other) source code. Such determination relies on checking the *provenance authenticity*. In critical domains like cyber-physical systems, it is key that any used smart contract is authentic. Thus, the provenance of such any smart contract must be well known and clear to all entities.

There are a number of platforms offering provenance verification services. Among the most popular alternatives, we find Etherscan [14], Tenderly [15], Sourcify [16], or Polygonscan [17], among others. These platforms provide a separate server for the verification task. This scheme means that the network specializes by task, resulting in a higher efficiency. Additionally, this approach enables the verification logic to work independently and provides a service to blockchain developers in order to verify the authenticity of any particular smart contract.

Block verifiers have seldom been used in the automatic deployment of logic in blockchain networks. A first approach in this direction is [18], which describes a programmatic structure to automate the verification process of smart contracts. Nevertheless, there is a broad open space to design and develop solutions that improve the verification of blocks in public networks to make them more efficient and time sensitive. The work of [19] has revealed some interesting results on very recent types of vulnerabilities that affect some smart contract verification services; this threatens to break the verification, resulting in the potential spreading of malicious smart contracts.

There are a number of works on the specific analysis of tools tailored to detect the main faults of smart contracts, like [20]. Testing applied to the smart contract source-code for fault detection has also been undertaken in a number of works like [21]. However, most works concentrate on analysing some specific tools and performing pure simulations; however,

they do not use any actual blockchain network nor any real smart contract deployments. As a result, and to the best of our knowledge, existing works do not focus on particular blockchain platforms' execution to measure the performance of the verification process in relation to the lifecycle of smart contracts.

On the side of the provenance verification, the work of [22] presents a provenance tracking system; this system leverages the static and dynamic analysis to detect and mitigate well known security issues specifically for Ethereum smart contracts.However, the vast majority of works either target the development of provenace frameworks over blockchain technology, like [23,24]; or focus on a transaction provenance, like [25]. Our work lies on a different level, as we focus on making the process of provenance verification usable for time-sensitive contexts by determining the execution cost of such a process.

3. Approach

The determination of the temporal cost of performing a verification over the authenticity of smart contracts is enabled by a middleware. The proposed middleware has been designed and architected as a modular framework with the needed pieces to perform automated verification tests and collect the temporal behavior of the interaction with any underlying block verification service.

This section describes our contribution in a self-contained manner. For this, the life cycle of a smart contract is first analyzed. Then, the different broad types of verification techniques related to smart contracts and block verifiers are described. Third, the set of most widely used block verification services is described. The middleware components are described afterwards, explaining the involved components one by one.

3.1. Life-Cycle Analysis

Blockchain networks rely on two fundamental ideas. On the one hand, their main purpose is automating the execution of an agreement without the participation of any intermediary. On the other hand, all involved participants must have immediate certainty on the outcome. To implement these ideas, blockchain technology utilizes the fundamental concept of a smart contract that is a program stored in the blockchain network itself. These programs execute when predefined conditions are satisfied. The terms of the agreement are also stored in the global distributed ledger. Since the global ledger is immutable, the applied modifications cannot be changed.

The life cycle of a smart contract determines the feasibility of the two fundamental ideas expressed above. The analysis of the life cycle of smart contracts is expressed in Figure 1, comprising the main phases from their programming to their deployment to the network. Also, the subsequent provenance verification process is shown.

The interaction of users with the distributed ledger can happen at any time, as users can request the execution of smart contracts hosted in the network. It is important to note that the *execution* is also a key part of the life cycle of these autoexecutable programs. Overall, the life cycle of a smart contract is broadly explained in what follows:

- *Contract programming*. This is the initial phase in which the contract is coded in some high-level programming language, e.g., Solidity.
- *Executable/bytecode generation*. The source code of a smart contract cannot run per se; so, an executable version of it needs to be created in order to be run by the nodes. For this reason, once the source code of a smart contract is fully programmed, it is compiled. Compilation generates a bytecode equivalent that is the executable version of the source code. The bytecode is the actual code that will eventually run in the blockchain network nodes. The bytecode is not human-readable; whereas the source code is.
- *Deployment*. The bytecode version of the smart contract must, then, be deployed to the distributed ledger. The deployment of a smart contract always results in the assignment of a particular address to it, which makes it uniquely identifiable. The address of a deployed smart contract plays a central role in the blockchain platform, as it is mandatory to refer to the corresponding smart contract that is running in

the network. Smart contract deployment involves the execution of a *transaction*. It is important to note that there are differences between a transaction involving asset exchange and a transaction to deploy a contract; the latter does not require one to specify a recipient. Smart contracts are also stored in the blockchain; then, deploying a smart contract involves storing it in the global ledger and assigning it a unique address. Deployment involves transaction fees that are usually higher than other transaction types.

- *Execution*. Smart contracts execute when the predefined conditions expressed in their code hold. The execution of a smart contract may involve changing the state of the global ledger, as new transactions may be initiated and additional blocks can be generated to store transaction records.
- *Verification*. This phase is optional; it is explained in what follows.

To deploy a smart contract in a network, it is not necessary that the smart contract is a verified one. For this, and for the sake of clarity, Figure 1 describes the path from smart contract programming to its deployment in the network (that is shown in the upper half part of the figure); and the verification process as a somehow separate path that is illustrated in the bottom half of the figure.

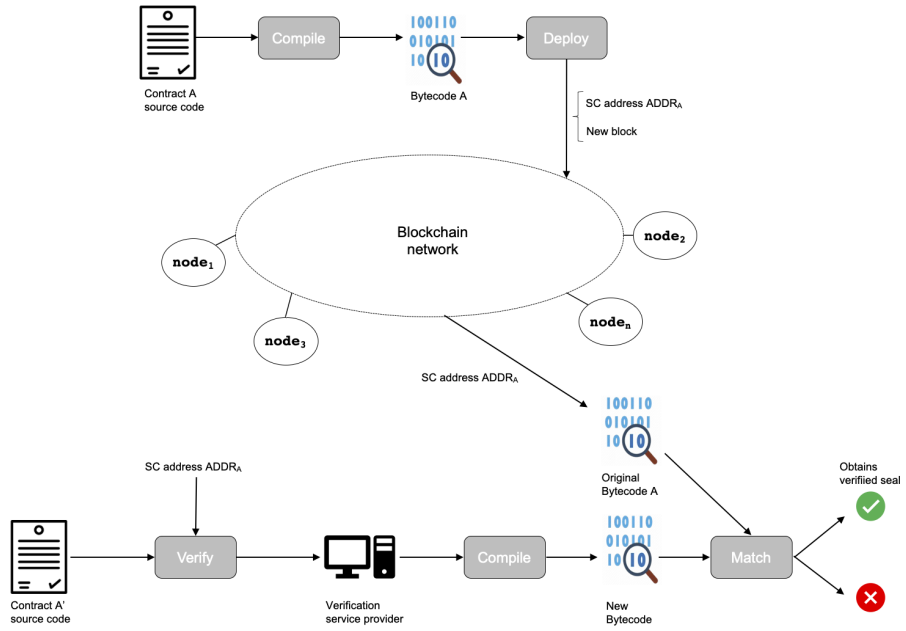

Figure 1. Life cycle of a smart contract and its verification process.

Verifying a given source code of a smart contract means that it should be true, such that the source code is exactly the same as the original bytecode that is running in the network.

Let us imagine that a developer wants to verify that source code A' is indeed the source code of a particular smart contract that is running in the network, namely A. If the result of the verification process is positive, this means that A' is the source code that originated A; so, they are the same contract (we say that A is an authentic logic or smart contract). Typically, A' is advertised in some platform's web with its corresponding data, and this includes its *address*. Using this address, the verification platform is able to retrieve the data of the running smart contract that will allow the verifier to decompile it, obtaining its original source code. This original source code will be checked against A'. If both match, then the verification is successful and the contract obtains a seal that reflects it. From that moment on, and given the nature of the blockchain technology, users can rely on the fact that this is a verified contract.

3.2. Verification Types

There are two broad types of provenance verification for smart contracts: *basic* and *perfect* (also named *full*) verification. Smart contract verification [26] is the mechanism that ensures that the bytecode stored in the blockchain corresponds to a given high-level language source code. Verifying a smart contract involves compiling the source code and checking whether it matches the stored bytecode. If both match, this means that the involved source code did produce the bytecode stored in the blockchain. The verification requires some precise information to be entered, like compilation information (i.e., the version of the compiler and that of the used optimizer), which is needed to exactly replicate the compilation process.

Most platforms perform a *simple* verification, and this yields also to a visible verification seal (usually a green checkmark) on the advertised contract. However, it is important to consider that, in the compilation process, some parts of the source code do not affect the result. Examples of such parts are variable names and comments. As a result, once two contracts with different variable names and/or comments are compiled, they may produce the same bytecode. This means that only the actual code developer can tell whether the code is exactly what was originally typed in. This leaves room for potential attackers to introduce missleading variable names and comments; and they will still be able to verify the contract to the same bytecode.

To avoid this, an improved verification mechanism, namely *full* or *perfect* verification, can be used. Full verification uses smart contract metadata to check the full integrity of the source code. The approach is simple. Metadata allows the contract to store an IPFS [27] link. This link contains a JSON file that includes interesting properties about the contract (e.g., compiler version, source names, etc.). Among those properties, there may be the hash of the original file containing the initial source code. Using such information, the verification logic can check whether such stored hash matches the hash obtained from the source code under analysis. If this is the case, it is guaranteed that the latter is identical to the source code from which the stored hash was obtained.

3.3. Component Design

The design of the middleware is described in this section. Its structure relates fully to the overall goal of performing an exhaustive evaluation of the temporal cost of block verification services. A modular design is set up for the middleware so that it can be adjusted to using particular block verifiers that support the easy replacement of neighbouring technology.

Figure 2 shows the general view of the middleware, and provides the overall scenario, including the actors that intervene in this context. The design of the middleware follows the principles of *modularity* and *separation of concerns*. A total of four components make up the middleware core, each component having a well-defined role and purpose. In the following, the components are described:

- *Generator*. Determining the temporal cost of the verification requires one to acquire a sufficient number of time samples that can be statistically representative. Each time sample corresponds to a particular smart contract verification occurrence. As block verifiers are able to distinguish whether successive smart contracts are the same, it is important to achieve an automatic generation of correct and unique smart contracts that will be identified as being unique. For this reason, the *Executor* component generates the unique smart contracts that are compiled and deployed to the blockchain. As shown in Figure 1, a smart contract can only be verified by a block verification service if it has been compiled and deployed. After being deployed, the blockchain infrastructure assigns a unique address to the running smart contract. Now, it can undergo a verification.
- *Executor*. This component manages the verification request triggers. After each smart contract is produced by the *Generator*, the *Executor* collects the verification parameters to be sent to the block verifiers. From the set of possible parameters, the bytecode

`address`, the corresponding `source code`, and the original *compiler options* are obtained. The address corresponds to the unique reference of a given smart contract and points to the physical location of the running bytecode in the global ledger.

- *Adaptor*. This component acts as a bridge between the inherent middleware logic and the particular underlying block verifier that might want to be used. As different verification techniques are possible and a number of block verifiers are available, this component makes the middleware independent from the particular underlying block verifier. To change the verification service, only this component is affected and requires minimal updates to switch from, e.g., Polygonscan to Tenderly.
- *Collector*. The temporal data are gathered by this component, which includes the invocation instants and the response data from the verification as well as its associated timing. The component manages an internal storage for the collection of the obtained data.

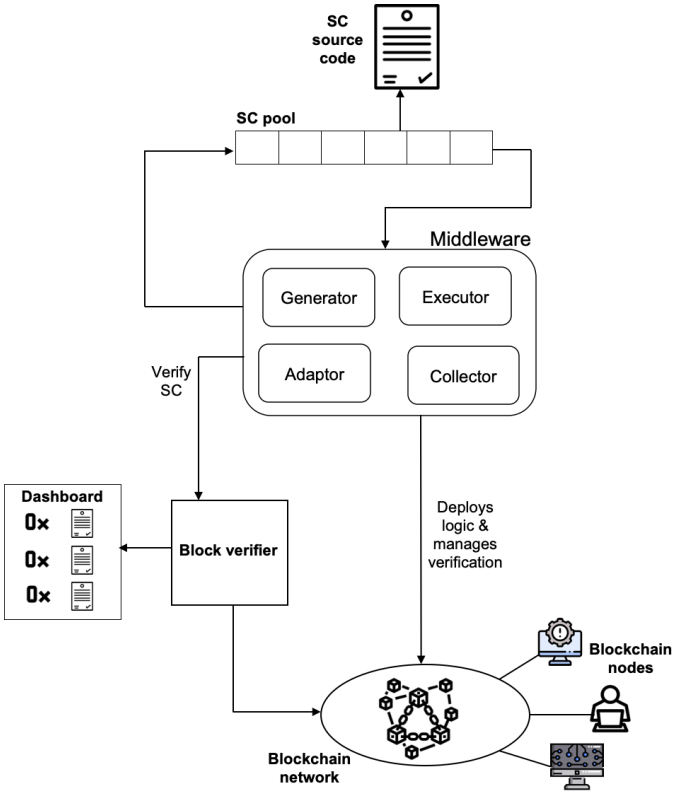

Figure 2. Middleware design.

3.4. Integration with Platform Alternatives through the Adaptor

The *Adaptor* component provides a hook to external verification actors. This is important given that in the current panorama of blockchain technologies, SC verification is often offered as a web service [28,29] that can be integrated in a system as an external actor. In this section, we describe the different alternatives that can be chosen for integration. The replacement of one verification service by another requires minor modifications that are confined to a particular part of the *Adapter* component.

Let us consider that, once a smart contract developer verifies it in a verification platform, a special seal (i.e., a verified mark) is obtained for that contract, and its information is stored in the verification platform. For this reason, anyone in the Internet can check whether a given smart contract has been verified in a particular verification platform.

Different solutions are present in the Internet that somehow compete to provide better verification logic and increase the number of verified smart contracts that they host. Among the different smart contract verification providers, this paper focuses on the mainstream

ones given the size of their user community and usability of the available tools. The most popular services come from Etherscan [14], Sourcify [16], and Tenderly [15].

Etherscan. Etherscan [14] is among the most famous companies in the domain of blockchain technologies. It consists of a block explorer for the main network of Ethereum and also for its test networks, including analytic facilities.

Etherscan is a block explorer for different blockchain networks like Ethereum, Arbitrum, or Polygon, among others. It allows users to check the information of particular blocks, which makes it an optimal location for integrating contract verification. Effectively, it includes the facilities to verify smart contracts. As a result, anyone on the Internet can access this platform to search for particular smart contracts to check whether they have been verified.

This is typically conducted in different ways: from the Etherscan web itself, from an external tool set (e.g., Hardhat [30]), or from a cloud IDE (e.g., Remix [31]).

Nevertheless, it provides a basic verification, not full verification. As a result, some variable name and comment modification attacks are still possible (as explained before). However, its popularity and simplicity attracts most of the blockchain users.

Polygonscan. Polygon [17] is a two-layer blockchain (also called *sidechain*) built on an Ethereum core. It incorporates a number of strategies for speeding up transactions and achieving lower fees. This makes it a parallel alternative to the Ethereum mainnet. Polygonscan is the block explorer that supports the interaction of users to access all transactions performed on the Polygon blockchain network.

Sourcify. Sourcify.eth [16] is also among the very top smart contract verification providers. On top of its verification process, it also offers a testing playground to test the verification process; and a block explorer to allow users to check whether a smart contract is already verified or not.

It is important to note a difference: a smart contract programmer is responsible for sharing the metadata file through IPFS. Once this is done, Sourcify engine checks that the file does exist somewhere and that it is reachable; if this check is successful, the verification process starts.

Tenderly. Tenderly [15] platform provides basic verification. An added value of Tenderly is that the visibility of the verified contracts can be configured. This means that the contract manager is able to restrict the visibility of the verificated source code to specific users. Controlling the visibility of the source code is a desirable characteristic in domains and systems that need to preserve the privacy over their source code, still providing some level of trustworthiness of that same code to the selected authorized third parties.

4. Implementation and Results

4.1. Experimental Settings

The cost of the verification of smart contracts in the lifecycle of a blockchain system is analyzed and presented in this section. A set of smart contracts programmed in Solidity are used in this analysis. The corresponding source code is designed to manage and record the actions over particular assets controlled by specified transaction terms. The source code reflects the logic to monitor the roles of potential users requesting access to resources, and how their access is granted.

As smart contract deployment is conducted in the Polygon network, the verification of smart contracts is conducted with the Polygonscan verifier, that is, the verifier component in this network. The development team of Polygonscan and Etherscan is mostly the same.

First, the smart contract source-code is compiled (see code below), generating a bytecode that will, then, be deployed on the Polygon network.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity \^0.8.0;
3
4 import ‘‘./IContrA.sol’’;
5 contract TermsC1 {
6 IContrA public CB1;
```

```
7
8  uint256 nonce = 71;
9  constructor(address CB1address) {
10 CB1 = IContrB(CB1address);
11 }
12
13 function chekterm(address enttAddress) public view returns (bool) {
14 IContrB.item memory enttyInf = CB1.getData(enttAddress);
15
16 if(keccak256(abi.encodePacked(enttyInf.prop)) == keccak256(abi.encodePacked('
       Value'))) {
17 return true;
18 }
19
20 return false;
21 }
22 }
```

The presented code shows a smart contract that codes in Solidity the policy applied to manage the access to a given resource. The function `chekterm` retrieves information from other contracts, precisely from `ContrB`. Then, data are stored in variable `enttyInf`, and are used to validate whether the specified condition is fulfilled. In this case, and for the sake of simplicity, a single condition is observed that checks whether a given entity that requests access has the proper value (`Value`) for a specified property `prop`.

The verification-related tools of current blockchain platforms provide developers with different options for system verification. For the case under study, Polygonscan, the verification process of a smart contract is separated into a number of steps, as explained in what follows.

To verify a smart contract, it must have been previously deployed to the blockchain network, and a corresponding address must have been assigned to it as its unique identifier.

Once deployed, a smart contract can be verified. In most platfoms, there are several options to verify a smart contract that is similar or equal to those of Polygon.

For many developers, a common option is to utilize the *web interface* of the platform [17] to launch the verification process of a smart contract. Using the Polygonscan block explorer, it is possible to obtain the smart contract block in the global ledger through its address. The verification process will need data like the compiler version, optimization, and (naturally) the source code, including constructor parameters, if any. It may also be necessary to specify the libraries used or other miscellaneous settings. Once all data are provided, the verification process starts. Overall, this can be quite a slow process, given that data are entered manually.

A time-saving (and more automated) option is to utilize a *especialized tool*, such as Hardhat [30] framework, which allows developers to make *direct use of the verification platform*. This tool supports the creation of a configuration file in the `etherscan.apiKey.polygonMumbai` object and in JSON. Once specified, to initiate the verification process as an online one, command `verify + SC address [+ constructor parameters]` is run, performing direct invocations to the verification framework through HTTP. To avoid misuse and/or attacks, Polygonscan provides APIkeys to registered users.

Once a smart contract has been verified, it is published with a green check in the Polygonscan platform. This way, its source code is visible to any interested user.

This is the approach followed in our experimental setting as it is the fastest option; consequently, it is likely to be used on deployments requiring a shorter verification time.

4.2. Results

A number of experiments were designed and run to obtain the performance of the verification under different conditions. The objective of the obtained measurements is to analyse the temporal cost of this process for different scenarios that are realistic and meaningful. Different smart contracts of varying sizes have been employed in the tests to obtain representative results. It should be observed that deploying the same smart contract on the network leads to caching. This effect must be avoided, as it alters the experiment

conditions and yields to unrealistic low temporal costs. To control the effect of caching, the programmed source-code is instrumented accordingly, to contemplate two different scenarios: cached and non-cached verification.

It should be noted that deploying smart contracts is a costly transaction type. Actually, deployment has an associated gas fee; thus, it is not currency-efficient. In summary, verification tries are bounded by this characteristic.

The experiments have been carried out exhaustively; and, for each scenario (cached and non-cached), ten different samples were obtained for each smart contract size. The sizes of the deployed smart contracts are shown in Table 1.

Table 1. Smart contract details.

Name	Lines of Code	Bytecode Size
UserAM	37	4583 Bytes
ContextHandler	24	2019 Bytes
Policy1	17	1916 Bytes

4.2.1. Cached Scenario

When a smart contract is sent for verification to a platform, the platform tends to cache it. This is a common behavior of servers on the Internet and an optimization for web interactions that involve resource requests over request-response protocols. The verification platform keeps copies of recently verified smart contracts. As a result, an already verified smart contract need not be uploaded again to the platform. This has a number of benefits, e.g., responsiveness and lower interaction times.

The temporal behavior in the cached scenario is ilustated in Figure 3. Here, it is shown that the temporal cost for the above smart contracts varies slightly, in median values, comparing all three smart contract sizes. The overall median values fall down to 1.57 s for all cases. This filters out the possible variations that may be done to the instant platform load. Variations show that the verification time for the medium-size smart contract are higher than for the big- and small-size smart contracts. However, this increase lies in the range of 100 ms, as it reaches 1.8 s in the 75th percentile; whereas the verification time of the big- and small-size smart contracts lies in the order of 1.6 s.

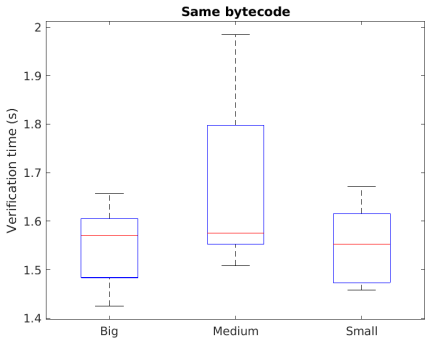

Figure 3. Performance in cached scenarios. Successive verification requests of the same SC do not result in new code uploads.

4.2.2. Non-Cached Scenario

This set of experiments focuses on situations where it is needed to constantly verify new contracts. This represents scenarios in which the server does not (or cannot) use the cache. Additionally, since our goal is to obtain *actual* costs of the verification process, we have also designed a non-cached scenario by proper (though minimal) modification of the smart contract source code.

As a result, in these experiments, each verification request is made over a different bytecode. This is instrumented in the code with a proper update of a `nonce` value. Such a minimal modification results in a bytecode that is different from the already deployed ones. Then, the platform must undertake a new verification process from scratch.

As illustrated in Figure 4, the non-cached scenario naturally yields higher verification times. It can be observed that the verification time decreases with the size of the smart contract source code. Nevertheless, a similar median value around 6.7 s is obtained for all smart contract sizes, which shows the stability of the verification platform. Additionally, the verification times of the small-size smart contract exhibits a lot less variability than the rest.

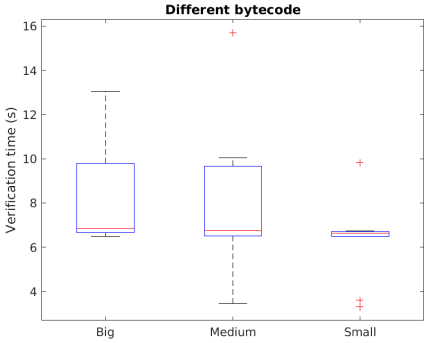

Figure 4. Performance in non-cached scenarios. Successive verification requests of same SC resulting in different bytecode that yield new source-code uploads.

5. Conclusions

There is an increasing need for smart contract verification in blockchain networks to provide a trusted environment to potential users. Particularly, the provenance verification can ensure the authenticity of the logic hosted in the network. However, this comes at a cost that cannot be neglected in some application domains like time-sensitive systems. Such systems have to perform an a priori analysis of the potential overhead of the provenance verification process. Once the temporal behavior of this process is determined, the obtained temporal results can be considered into their design equation to prepare for corrective or mitagation actions, if needed.

In this paper, we analyzed the life-cycle of smart contracts, paying the most attention to the verification phase and associated interactions undertaken by the network to verify smart contract authenticity. We presented a modular middleware that employs the separation of concerns to automate the process of determination of the temporal cost of verification for particular platforms and block verifiers. Taking the resulting values into consideration, the development process can determine whether the extracted temporal values are affordable and suitable for the target system.

We have validated our contribution by implementing the middleware on an actual blockchain network.The performed experiments employed two different underlying block verifiers that prove the modularity of the middleware design. The employed block verifiers are instrumented to operate with and without caching. Experiments showed that the interaction with the verification services yields stable times. The obtained median values are kept stable, independently from the employed smart-contract size. Delays in the most unfavorable scenarios are in the range of 6.7 s, which greatly pays off for operating with assurance about the authenticity of the employed smart contract. The automation of the provenance verification and the good temporal results were obtained in real blockchain networks, offering advice on the extrapolation of this verification type to an online execution in complex systems.

Author Contributions: Conceptualization, Methodology: M.G.-V., A.M.C.-C.; Investigation: A.M.C.-C., M.G.-V.; Graphs and experiments: A.M.C.-C.; Supervision: M.G.-V.; Journal version writing, reviewing, and editing: M.G.-V.; Funding Acquisition: M.G.-V. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the following projects: "*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. The APC was funded by AICO/2021/138 (Subvenciones para grupos de investigación consolidados). Also, this work has been supported by Grant PID2021-123168NB-I00 funded by MCIN/AEI/10.13039/501100011033 and ERDF *A way of making Europe*; and by project *Green Networks: Towards environmental-friendly 6G Networks* under grant TED2021-131387B-I00 funded by MCIN/AEI/10.13039/501100011033 and by the European Union NextGenerationEU/RTRP.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alchemy. Ethereum Statistics. 2022. Available online: https://www.alchemy.com/overviews/ethereum-statistics (accessed on 29 December 2023).
2. Szabo, N. Smart Contracts. Technical Report, Nick Szabo's Essays, Papers, and Concise Tutorials. 1997. Available online: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html (accessed on 29 December 2023).
3. Szabo, N. The Idea of Smart Contracts. Available online: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (accessed on 29 December 2023).
4. Szabo, N. Smart Contracts: Formalizing and Securing Relationships on Public Networks. 1997, Volume 2. Available online: https://firstmonday.org/ojs/index.php/fm/article/download/548/469 (accessed on 29 December 2023).
5. Gomez Gelvez, M. Explaining the DAO Exploit for Beginners in Solidity. 2016. Available online: https://medium.com/@MyPaoG/explaining-the-dao-exploit-for-beginners-in-solidity-80ee84f0d470 (accessed on 29 December 2023).
6. Jyoti, A.; Chauhan, R.K. A blockchain and smart contract-based data provenance collection and storing in cloud environment. *Wirel. Netw.* **2022**, *28*, 1541–1562. [CrossRef]
7. Dictionary, O.E. Available online: https://www.oed.com/ (accessed on 29 December 2023).
8. Blair, J.; Johnson, R.H. Informal Logic: An Overview. *Informal Log.* **2000**, *20*, 93–107. [CrossRef]
9. Qu, M.; Huang, X.; Chen, X.; Wang, Y.; Ma, X.; Liu, D. Formal Verification of Smart Contracts from the Perspective of Concurrency. In Proceedings of the International Conference on Smart Blockchain, Tokyo, Japan, 10–12 December 2018; Springer: Cham, Switzerland, 2018; pp. 32–43.
10. Sun, T.; Yu, W. A formal verification framework for security issues of blockchain smart contracts. *Electronics* **2020**, *9*, 255. [CrossRef]
11. Wang, D.; Huang, X.; Ma, X. Formal analysis of smart contract based on colored petri nets. *IEEE Intell. Syst.* **2020**, *35*, 19–30.
12. He, Y.; Dong, H.; Wu, H.; Duan, Q. Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN. *Electronics* **2023**, *12*, 2152. [CrossRef]
13. Ye, J.; Ma, L.; Lin, Y.; Xue, Y.; Sui, Y.; Peng, T. Clairvoyance: Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Melbourne, VIC, Australia, 21–25 September 2020; pp. 274–275.
14. Ethereum Foundation. Etherscan. Available online: https://etherscan.io/ (accessed on 29 December 2023).
15. Tenderly. Tenderly Node. Available online: https://tenderly.co/ (accessed on 29 December 2023).
16. Sourcify.eth. Available online: https://sourcify.dev/ (accessed on 29 December 2023).
17. Polygon Labs. Polygon. Available online: https://polygon.technology/ (accessed on 29 December 2023).
18. Chirivella-Ciruelos, A.M.; García-Valls, M. Automating the verification of smart contracts in blockchain networks for improving security. In Proceedings of the 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Durres, Albania, 6–8 September 2023.
19. Ma, P.; He, N.; Huang, Y.; Wang, H.; Luo, X. Abusing the Ethereum Smart Contract Verification Services for Fun and Profit. *arXiv* **2023**, arXiv:2307.00549.
20. Dia, B.; Ivaki, N.; Laranjeiro, N. An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools. In Proceedings of the 26th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), Perth, Australia, 1–4 December 2021; pp. 17–26.
21. Frank, J.; Aschermann, C.; Holz, T. ETHBMC: A Bounded Model Checker for Smart Contracts. In Proceedings of the 29th USENIX Security Symposium, Boston, MA, USA, 12–14 August 2020.
22. Linoy, S.; Ray, S.; Stakhanova, N. EtherProv: Provenance-Aware Detection, Analysis, and Mitigation of Ethereum Smart Contract Security Issues. In Proceedings of the 2021 IEEE International Conference on Blockchain (Blockchain), Melbourne, Australia, 6–8 December 2021; pp. 1–10.
23. Sun, L.S.; Bai, X.; Zhang, C.; Li, Y.; Zhang, Y.B.; Guo, W.Q. BSTProv: Blockchain-Based Secure and Trustworthy Data Provenance Sharing. *Electronics* **2022**, *11*, 1489. [CrossRef]

24. Yin, F.; Fu, Z. A Data Provenance Scheme Based on Blockchain for Internet of Things. In Proceedings of the 2022 2nd International Conference on Computer Science and Blockchain (CCSB), Wuhan, China, 28–30 October 2022; pp. 42–45.
25. Geng, Z.; Cao, Y.; Li, J.; Han, Y. Novel blockchain transaction provenance model with graph attention mechanism. *Expert Syst. Appl.* **2022**, *209*, 118411. [CrossRef]
26. Marx, S. Verifying Contract Source Code. 2018. Available online: https://programtheblockchain.com/posts/2018/01/16/verifying-contract-source-code/ (accessed on 29 December 2023).
27. Benet, J. IPFS—Content Addressed, Versioned, P2P File System. *arXiv* **2014**, arXiv:1407.3561.
28. @minimalsm. Verifying Smart Contracts. 2023. Available online: https://ethereum.org/en/developers/docs/smart-contracts/verifying/ (accessed on 29 December 2023).
29. Lukic, M. A Guide to Smart Contract Verification. 2022. Available online: https://blog.tenderly.co/guide-to-smart-contract-verification-methods/ (accessed on 29 December 2023).
30. Hardhat Network. Ethereum Development Environment for Professionals. Available online: https://hardhat.org/hardhat-network/docs/reference (accessed on 29 December 2023).
31. Ethereum Foundation. Ethereum. REMIX IDE. Available online: https://remix.ethereum.org (accessed on 29 December 2023).