IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 21, 2020, accepted October 10, 2020, date of publication November 2, 2020,
date of current version November 12, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3035271

Monitoring File Integrity Using Blockchain and Smart Contracts

ALEXANDRE PINHEIRO[1], (Member, IEEE), EDNA DIAS CANEDO[2], (Member, IEEE), RAFAEL TIMÓTEO DE SOUSA, JR.,[1] (Senior Member, IEEE), AND ROBSON DE OLIVEIRA ALBUQUERQUE[1]
[1]Electrical Engineering Department, National Science and Technology Institute on Cybersecurity, University of Brasília (UnB), Brasília 70910-900, Brazil
[2]Department of Computer Science, University of Brasília (UnB), Brasília 70910-900, Brazil

Corresponding author: Alexandre Pinheiro (alexandre.pinheiro@redes.unb.br)

This work was supported in part by the Brazilian Higher Education Personnel Improvement Coordination (CAPES) under Grant PROAP PPGEE/UnB, Grant 23038.007604/2014-69 FORTE, and Grant 88887.144009/2017-00 PROBRAL; in part by the Brazilian National Research Council (CNPq) under Grant 312180/2019-5 PQ-2, Grant BRICS2017-591 LargEWiN, and Grant 465741/2014-2 INCT on Cybersecurity; in part by the Brazilian Federal District Research Support Foundation (FAP-DF) under Grant 0193.001366/2016 UIoT and Grant 0193.001365/2016 SSDDC; in part by the Brazilian Ministry of the Economy under Grant 005/2016 DIPLA and Grant 083/2016 ENAP; in part by the Institutional Security Office of the Presidency of Brazil under Grant ABIN 002/2017; in part by the Administrative Council for Economic Defense under Grant CADE 08700.000047/2019-14; and in part by the General Attorney of the Union under Grant AGU 697.935/2019.

ABSTRACT The adoption of cloud computing solutions is an established reality in government agencies and in small, medium, and large companies due to procurement easiness and the variety of available services, as well as its low cost compared to the acquisition and management of own infrastructures. Among the most used services is cloud file storage, and the security of this storage has been an essential subject of recent research, particularly customer data integrity. Thus, this article proposes a solution for the monitoring of the integrity of files stored in the cloud, based on the use of smart contracts in Blockchain Networks, symmetric encryption, and computational trust. The proposed solution consists of a protocol that provides confidentiality, decentralization, audit availability, and the secure sharing of file integrity monitoring results, without overloading the services involved, as well as an unabridged reference implementation which was used to validate the proposal. The results obtained during the validation tests have shown that the solution is feasible and faultless in detecting corrupted files. These tests also confirmed that the sharing of integrity monitoring results, coupled with the application of computational trust techniques, significantly increased the efficiency of the proposed solution.

INDEX TERMS Blockchain, cloud computing, data security, smart contracts, trust.

I. INTRODUCTION

Cloud Computing (CC) is an environment that provides on-demand access, through the network, to computational resources such as storage, servers, applications, and other services, which the customer can efficiently aggregate or release [1]. Customers can rapidly create cloud services in distributed cloud data centers where they can store and process their data, and deploy and efficiently run their applications [2]. CC services follow a business model that charges the customer for the use of computing resources, which the customer quickly contracts and manages through standardized web services, without bureaucracy.

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

Scalability, availability, and virtually unlimited storage capacity can be highlighted among the advantages of CC. Given the popularization of CC services, mainly due to their reduced cost compared to traditional technologies and the continuous appearance of new data storage service providers in the cloud, many companies are choosing these services to store their information [3]. CC relieves the customer from being concerned with the complexity of managing the storage infrastructure. However, the confidentiality, availability, and integrity of the stored data depend on the quality of the service provided. Because of structural or human flaws, the data may be corrupted, leaked, or deliberately deleted.

Since CC comprises comprehensive technology under continuous evolution over the years, it still offers many study opportunities in open research areas. Much research work has been integrating different techniques and technologies to both solve problems and improve the results of CC solutions, including the following examples: i) Yuan *et al.* [4] propose an algorithm, based on machine learning techniques, to improve the distribution schedule of tasks from CC services between geographically distributed data centers powered by green energy, aiming to optimize profit and reduce task loss; ii) Bi *et al.* [2] propose a new framework, based on mathematical modeling techniques, to provide dynamic resource provisioning in virtualized cloud data centers, minimizing energy costs by keeping unnecessary virtual machines off while meeting the service-level agreed between the customer and the cloud service provider; and, iii) Wilczyński and Kołodziej [5] propose a new generic model for a secure cloud scheduler, based on Blockchain technology, where each neighbor node in the cloud cluster approves the proposed task schedules by means of a novel ''proof–of–schedule'' consensus algorithm, obtaining an internal agreement before offering the schedule to end-users.

As another example of this technology, Blockchain enables the creation of a decentralized database, in which agents and institutions can carry out verifiable transactions, without any of the parties being able to control or impose market power. One feature of Blockchain is that it enables the effective maintenance of a consensus in the chronological order of events and the state of affairs [6].

Blockchain implementations allow the creation of networks comprised of interested participants, among which there is no requirement for mutual trust. The transactions completed by these participants are recorded in small blocks forming a chain of sequential blocks (Blockchain) that is replicated to all participants in the respective network. The transactions are carried out directly between the parties without the need for a trusted third party. Once a new block is inserted into the Blockchain, the registered transactions cannot be changed or reversed [7].

A smart contract (SC) is a tamper-proof digital contract that is often enforced through automated execution [6]. Each SC is a small computer program stored and executed in the Blockchain to fulfill the terms of an agreement between parties that are not required to trust each other [7].

Information owners often need to keep information items stored for an extended period, without necessarily having to access them. And normally, due to their size, the owners often do not keep other copies of it. Given the possible damage caused by the loss of stored data, and for the users to acquire the trust necessary to store data using cloud storage services, these services must provide tools capable of allowing the customers to continually verify the integrity of their data.

The previous related study [8] proposed an architecture that allows the periodic verification of the integrity of files stored in the cloud by third parties, but without exposing the content of these files. The referred architecture presents a protocol based on challenges that generate low and predictable network bandwidth consumption. It also gives a mechanism, based on concepts of computational trust, to balance the load of checks that speed up or slow down according to the storage service behavior. Notwithstanding its efficacy, this architecture presents the following limitations: i) the requirement of complete trust in the third-party service responsible for performing the integrity check of the files hosted in the cloud; ii) the impossibility of carrying out any audit in both processes and results; iii) the trust level assessment of the services responsible for cloud file storage being verified separately by each integrity check service, and based exclusively on their observations regarding the behavior of each storage service provider; and, iv) the requirement for storage service providers to maintain a service available 24 hours a day exclusively to receive the challenges submitted by the integrity check services.

A. MAIN CONTRIBUTIONS OF THIS STUDY

This study proposes a solution based on the use of Blockchain technology for the storage of files in a Cloud Storage Service (CSS) that allows Clients to contract untrustworthy third parties to carry out permanent and auditable monitoring of the integrity of these files using challenges, without compromising the confidentiality of the stored information, through SCs. In addition, this study proposes a shared process to classify a CSS according to trust levels based on its behavior, this task being performed autonomously by SCs. The proposed solution improves previous work [8], minimizing its limitations, and increasing its efficiency. The main differences between this work and the one done previously [8] are the following:

- the use of a storage infrastructure in a Blockchain Network (BN) intended to record information on the file stored in the cloud to verify its integrity, as well as information related to the steps of the respective verification process which, thanks to immutability, inviolability, and resilience provided by the Blockchain technology, guarantee transparency, security, and the possibility of auditing all stages of the file storage and integrity check processes;
- the automation and decentralization of the analysis process of the results generated by monitoring the files stored in the cloud, with the use of SCs stored in a BN, enable the hiring of Integrity Check Services (ICS) that do not have a previous relationship of trust with the Client, and due to the SC characteristics, the integrity verification process is predictable, transparent and auditable by any of the involved parties, making collusion between the service contracted to verify the integrity and the service liable for storage unfeasible;
- the decentralization and sharing of the calculation process of the trust level assigned to each CSS through an SC stored in a BN, without interference from other roles (Client, ICS, and CSS), whenever requested by other SCs which, in turn, autonomously evaluate the responses from the CSS regarding challenges generated by the integrity verification processes of one or more

ICS, preventing ICS attacks against the reputation of the CSS;

- the ability to allow the CSS to audit the file received for storage, through the prior endorsement of the rules implemented in the SC linked to the file and registered at the BN, and to validate the information on the file stored in that SC, which will be used check file integrity, so that only those SCs previously approved by the CSS will be allowed to contribute to the shared calculation of the trust level assigned to CSS, a feature that reduces the chances of success of Client attacks against the reputation of the CSS;
- the automation (contracting and renewal) of the management contracts for the verification of the integrity of files stored in the CSS, entered into between the Client and the ICS, with the automatic and periodic submission of the information needed to execute the integrity checks (submission of challenges) exclusively for the contracted period, features which allow the replacement of the ICS during the contracted file storage period, without interfering with the security of the integrity monitoring process;
- the proposal of a new protocol that establishes the responsibility for each of the four roles stipulated in the solution architecture (Client, CSS, ICS, and BN), with a detailed definition of the actions performed by each part, and the messages exchanged between them;
- the reference implementation of the proposed protocol, covering the functionalities established for all roles, compatible with networks that adopt the Blockchain Ethereum platform, composed of the following components: i) a SC responsible for calculating, storing, and sharing the trust level assigned to each CSS; ii) a SC responsible for storing the file validation information, receiving challenges from the ICS, and receiving and validating the CSS responses; iii) a desktop application that implements the tasks established for the Client role; iv) a web service application that implements the expected duties of the CSS role; and, v) a web service application that implements the tasks established for the ICS role.

B. STUDY STRUCTURE

This article is structured as follows: Section II presents a review of related works. Section III describes the proposal for the new protocol and an analysis of the security aspects of the proposed protocol. Section IV then defines a complete reference implementation of the proposed protocol. In Section V, we show the protocol validation with the description of the tests performed and the analysis of the achieved results. Finally, section VI contains concluding remarks on central aspects of the work and possible future steps.

II. BACKGROUND

This Section presents a brief description of research related to the application of Blockchain, SC, and Computational Trust technologies, mainly in the areas of auditing, integrity, and reliability.

A. BLOCKCHAIN AND SMART CONTRACTS

Xue *et al.* [9] propose a scheme to conduct public audits based on cloud storage system identities. The proposed scheme, called IBPA, is based on nonces from a public Blockchain, such as bitcoins, as a mechanism to randomize the challenges. This scheme allows an adequate, *a posteriori* and in batch verification of the audits performed by third parties, preventing the service user from being mislead with false results. The IBPA also proposes using a public Blockchain as a means of storing records of the results of conducted audits, which makes the records, due to the characteristics of Blockchain technology, traceable, verifiable and undeniable. The proposed scheme bases its security on the mathematical problem of *Diffie-Hellman*, and its operation involves four entities: a fully trusted authority responsible for generating the private keys, a user/client of the service, a cloud storage service, and a third-party auditing service [9].

Yu *et al.* [10] propose a framework based on Blockchain technology to perform audits of large databases (Big Data) in a decentralized manner and without the need for a third party. The framework uses an optimized Blockchain instance called Data Auditing Blockchain (DAB), which collects evidence from the audit performed instead of the financial transaction records, and uses a variant of the Practical Byzantine Fault Tolerance (PBFT) algorithm as a consensus algorithm. Through the use of DAB, the scheme proposed in the framework enables the traceability of the audit history and, at any time, the validation of this history by any user. The framework implements an algorithm that permits batch validation, which reduces the consumption of computational resources required to execute the process. It also supports dynamic data based on a modified version of the dispersion tree, called *modified Merkle Hash Tree* (mMHT), which is stored and managed by the CSS [10].

Wang *et al.* [11] propose an auditable payment and delivery protocol for physical assets based on SCs. The proposal uses three types of SCs designed to obtain reliable payments between merchants, consumers and logistics companies, and all network members. The protocol requires the use of a third party, called regulator, responsible for authenticating users interested in participating in the network, and registering an SC with their data in the Blockchain. It is also up to the regulator to destroy (render invalid) the referred contract when the user chooses to leave the network. The common characteristic of SCs between trader and consumer, and of SCs between trader and logistics companies is the reserving of the financial resources necessary to settle the transaction in the form of a deposit. The deposit is only released to the recipient after delivery completion, or it is returned when requested by one of the parties. The use of Blockchain technology by the protocol provides an effective method to audit assets and to trace the data generated during transport [11].

Ahmad *et al.* [12] propose a multilayered Blockchain architecture to provide higher processing capacity and reduced delays than traditional Blockchain systems. Based on this architecture, a solution for audit trail applications called *BlockTrail* was proposed, which seeks to reduce the complexity of space and time common in the Blockchain-based audit applications. In the proposal, the authors mathematically demonstrate the advantages of multilayered architecture concerning complexity in terms of time, space, and demand. They also analyzed issues related to security, where they determine the minimum limits on the number of both active and honest Blockchain replicas, to prevent being vulnerable to malicious replicas. Finally, they presented the countermeasures that could be adopted, capable of detecting and preventing an attack on a Blockchain network [12].

B. COMPUTATIONAL TRUST

Albuquerque *et al.* [13] propose an information security architecture that connects elements that combine the treatment of information in cyberspace with measures based on computational trust, thus ensuring cybersecurity. This related research evaluated the relations between computational trust and information security, analyzing the development, applications, and market for exploits (tools used to exploit vulnerabilities in cyberspace). The cited paper also presented a study on Advanced Persistent Threats (APTs), describing the APT concept, the APT execution flow and the risks that APTs offer to the cyber environment, discussing some leading examples of APTs [13].

Mohammed and Omara [14] propose a ranking model for cloud service providers (CSP) based on trust degrees and the similarity between the service level agreement (SLA) parameters requested by consumers and the parameters offered by service providers. The proposed model applies a process divided into phases that include filtrating, trusting, similarity, and ranking. The filtrating phase uses a Fuzzy Controller System to reject an untrustworthy CSP. Next, the trusting phase applies the Particle Swarm Optimization technique to determine the CSP trust degrees. Then, the similarity phase employs Cosine Similarity Measures to compute the similarity percentage between requested and offered SLA parameters. Finally, the ranking phase uses the trust degree and similarity percentage combination to classify the CSP according to the capacity of providing the service to the customer.

C. INTEGRITY AUDITING OF FILES STORED IN CLOUD STORAGE SERVICES

El Ghazouani *et al.* [15] propose using Blockchain technology in conjunction with a system formed by multiple agents to perform deduplication and auditing of files stored in the CSS. On the CSS side, the proposed approach uses agents to identify entire files or parts thereof that belong or not to the same Client, which have already been stored, discarding new copies and thus saving disk space. For this, an agent divides the file into parts and stores the hashes of each region in a dispersion tree of the Merkle Hash Tree (MHT) type and a hash database. Regarding the audit, the authors propose the delegation of this competence to an outsourced ICS whose trust is limited, which is why the Client only receives the information necessary to generate challenges to the CSS but does not have access to the content of the audited files. The integrity verification mechanism used is an adapted version of the challenge/response protocol based on MHT, proposed by Coelho [16].

The scheme for real-time auditing of images stored in the cloud, proposed by Tang *et al.* [17], uses a reversible and adaptable watermark algorithm that makes it possible to incorporate authentication data without distortion or loss of image quality. The scheme uses a new challenge-response mechanism that guarantees privacy, and is resistant to replay attacks based on the *Diffie–Hellman* key exchange protocol. The proposed scheme provides three entities: the client, the cloud storage provider, and a third party who acts as a trustworthy arbitrator between the other entities. Due to the small capacity of the block generated by the watermark (416 bits), and the need to insert at least two signatures, the scheme adopts a short signature model called BLS [17].

Wei *et al.* [18] propose a Blockchain-based integrity protection framework to monitor files stored in distributed storage servers. The framework adopts a mechanism based on challenge-response that uses smart contracts distributed in a Blockchain network whose nodes are the cloud storage servers. These smart contracts cooperate to ensure data trust verification, and each node is responsible for generating challenges to verify whether the pieces of the file stored in the other nodes are intact. The integrity verification protocol proposed by the authors uses a unique file hash generated by a MHT and an asynchronous cryptographic mechanism to create the challenges and validate the answers.

The works described in Section II describe several Blockchain technology applications that emphasize auditing [9], [10], [17], integrity, and reliability techniques [11], [14], [15]. This work adapts a large part of these techniques that involve the use of Blockchain, but it differs from related works and previous research [8], mainly by combining Computational Trust with Blockchain and SCs to monitor, in a safe and auditable way, the integrity of files stored in the cloud and to share the experiences regarding the quality of the provided storage services.

III. PROTOCOL FOR MONITORING THE INTEGRITY OF FILES STORED IN THE CLOUD

We developed the protocol to operate in an architecture comprised of the following four roles: i) Client; ii) Cloud Storage Service (CSS), iii) Integrity Check Service (ICS), and iv) *Blockchain Network* (BN), as shown in Figure 1. The Client is the owner of the file, while the CSS is responsible for receiving, storing, and maintaining the integrity of the file during the contracted period. The ICS is responsible for periodically submitting challenges to the CSS to check the integrity of the file. Through the SCs, the BN stores

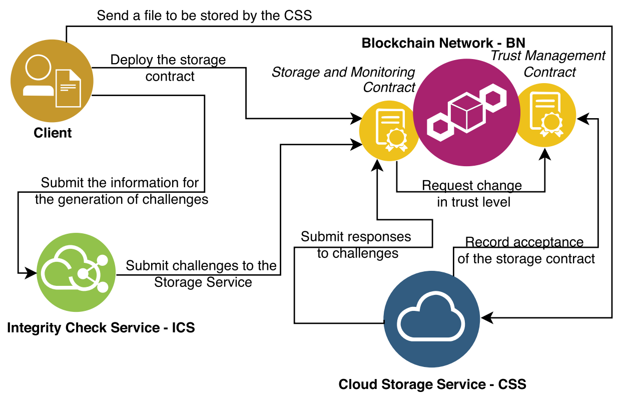

Send a file to be stored by the CSS
Blockchain Network - BN
Deploy the storage contract
Storage and Monitoring Contract
Trust Management Contract
Client
Submit the information for the generation of challenges
Request change in trust level
Submit responses to challenges
Record acceptance of the storage contract
Submit challenges to the Storage Service
Integrity Check Service - ICS
Cloud Storage Service - CSS

FIGURE 1. **Protocol overview.**

the information established for the file integrity verification generated by the Client, the challenges submitted by the ICS, the responses recorded by the CSS, and verifies the correctness of these responses. Finally, the BN calculates the trust levels and stores them for each CSS.

Aiming to facilitate understanding, we divided the proposed protocol description into three phases, each described accordingly in a proper section. In Section III-A, "PREPARATION PHASE", we describe the preparation of the BN for the operation of the proposed protocol by submitting a SC called the Storage Service Trust Management Contract (SSTMC) to the BN. The SSTMC implements the calculations to change the trust level assigned to the CSS, enables the service registry and offering by the CSS and ICS, stores the trust level calculated for each registered CSS, and records the CSS acceptance of each file storage request submitted by the Clients.

Next, in Section III-B, STORAGE PHASE, we describe the processes the Client performs to prepare the file for storage and to submit the file to the cloud. In this phase, we provide a detailed explanation of the encryption process, the generation of the information necessary for the monitoring of the file during the storage period, the submission of the encrypted file copy to the CSS, and the insertion of an instance of the SC named Cloud File Storage and Monitoring Contract (CFSMC) into the BN.

Then, in Section III-C, INTEGRITY CHECK PHASE, we describe the processes of generation of challenges by the ICS, of generation of responses by the CSS, analysis of the responses carried out by the CFSMC in the BN, and the calculation of the trust level assigned to the CSS performed by the SSTMC in the BN. Finalizing the proposal protocol, in Section III-D, PROTOCOL SECURITY ANALYSIS, we present an analysis of the security aspects of the proposed protocol.

A. PREPARATION PHASE

The protocol begins with the insertion of an instance of the SC SSTMC into the BN, whose implementation we described in Section IV-B1. For the protocol to work, one or more active SSTMC instances must be available in the BN, and it is up to the ICS and CSS to choose which will offer their services. Such a situation may occur due to the availability of improved versions of the SSTMC or to the need to implement customized versions to meet a specific criteria or requirement.

Thus, to enable this protocol for use, at least one CSS and one ICS must offer their services through an instance of SSTMC inserted in the BN. The SSTMC implements the method called "registerStakeholder", which allows any participant of the BN to self-register as an ICS or a CSS, becoming able to be selected by the Clients.

After processing the transaction of inserting each instance of the SSTMC into the BN, the generated SSTMC access address must be publicly available since the protocol does not contain its self-disclosure mechanism. The Clients, ICS, and CSS must register at least one of the SSTMC instance addresses available in the BN directly in their respective applications. The SSTMC is responsible for executing, in the BN, the computational trust processes within the scope of the proposed protocol. The standard implementation of the SSTMC adopts a modified version of the trust level classification model proposed by Pinheiro *et al.* [8], which is detailed in Table 1.

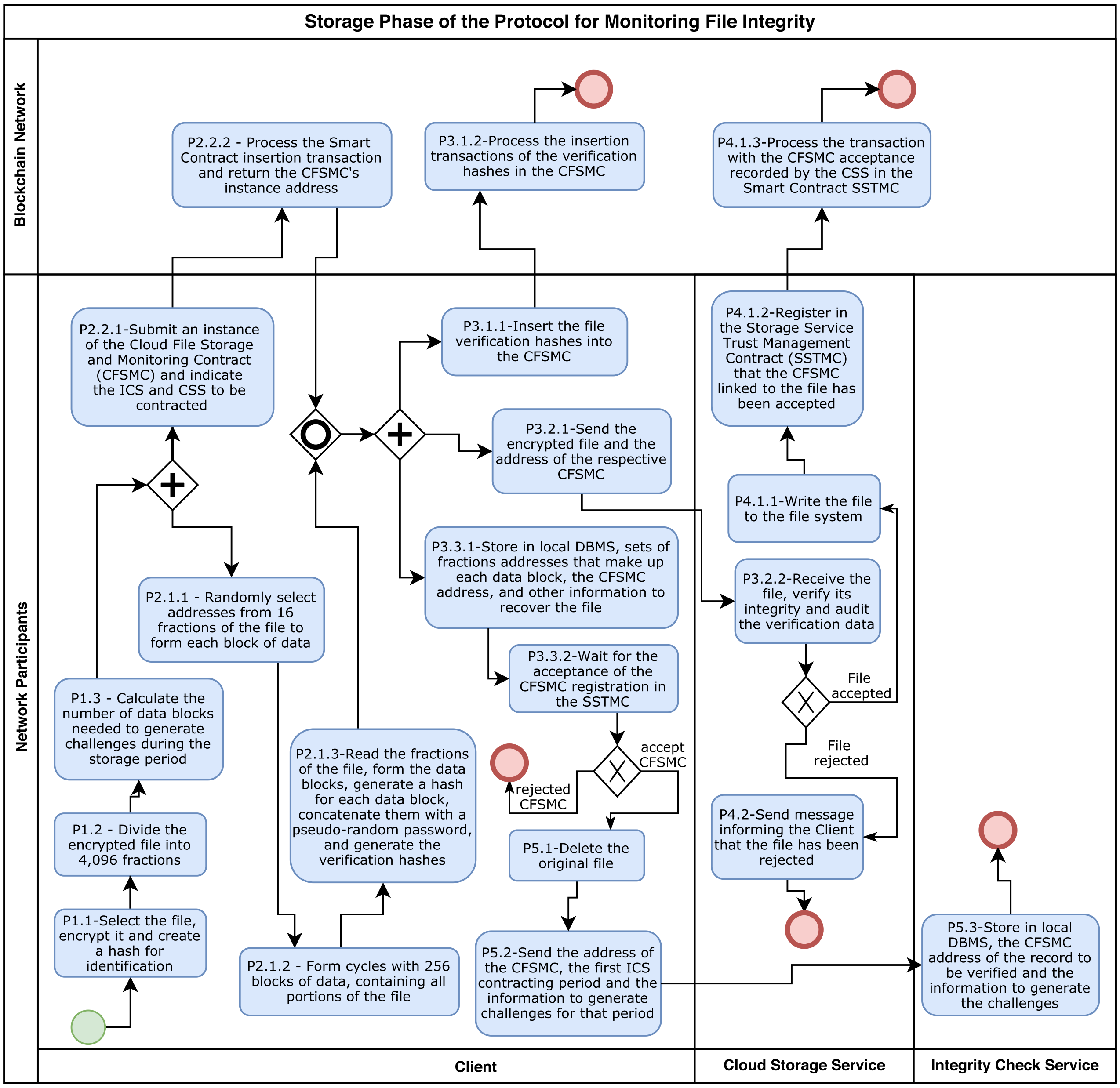
TABLE 1. **Trust Level Classification [Pinheiro *et al.* 2018 apud Marsh 1994, adapted].**

Trust Level	Trust Value Range	Checked per day		
		% from files	% of file	Data Blocks
Very high trust	$]9 \times 10^{19}, 1 \times 10^{20}[$	15%	$\sim 0.4\%$	1
High trust	$]7.5 \times 10^{19}, 9 \times 10^{19}]$	16%	$\sim 0.8\%$	2
Medium-high trust	$]5 \times 10^{19}, 7.5 \times 10^{19}]$	17%	$\sim 1.2\%$	3
Low-medium trust	$]2.5 \times 10^{19}, 5 \times 10^{19}]$	18%	$\sim 1.6\%$	4
Low trust	$[0, 2.5 \times 10^{19}]$	19%	$\sim 2.0\%$	5
Low distrust	$]-2.5 \times 10^{19}, 0[$	20%	$\sim 2.4\%$	6
Low-medium distrust	$]-5 \times 10^{19}, -2.5 \times 10^{19}]$	25%	$\sim 3.2\%$	8
Medium-high distrust	$]-7.5 \times 10^{19}, -5 \times 10^{19}]$	30%	$\sim 4.0\%$	10
High distrust	$]-9 \times 10^{19}, -7.5 \times 10^{19}]$	35%	$\sim 4.8\%$	12
Very high distrust	$]-1 \times 10^{20}, -9 \times 10^{19}]$	50%	$\sim 5.6\%$	14

Due to the possibility of using multiple instances of the SSTMC in the same BN and to how we implemented the standard version of the SSTMC, it is easily feasible to create customized versions of the SSTMC that change the behavior of the adopted computational trust model. It is only necessary to adjust the constants that represent the intervals described in Table 1, without changing the implementation of the routines responsible for calculating the trust value. In Section III-C4, we describe these calculation routines.

B. STORAGE PHASE

In this phase, the Client begins the protocol with the file preparation and submission to storage in the CSS, whose process we describe in detail in Section III-B1. Then, the CSS starts the process of auditing and accepting the received file, which we present in Section III-B2. If the storage request



Storage Phase of the Protocol for Monitoring File Integrity
Blockchain Network
Network Participants
P2.2.2 - Process the Smart Contract insertion transaction and return the CFSMC's instance address
P3.1.2-Process the insertion transactions of the verification hashes in the CFSMC
P4.1.3-Process the transaction with the CFSMC acceptance recorded by the CSS in the Smart Contract SSTMC
P2.2.1-Submit an instance of the Cloud File Storage and Monitoring Contract (CFSMC) and indicate the ICS and CSS to be contracted
P3.1.1-Insert the file verification hashes into the CFSMC
P4.1.2-Register in the Storage Service Trust Management Contract (SSTMC) that the CFSMC linked to the file has been accepted
P3.2.1-Send the encrypted file and the address of the respective CFSMC
P4.1.1-Write the file to the file system
P3.3.1-Store in local DBMS, sets of fractions addresses that make up each data block, the CFSMC address, and other information to recover the file
P3.2.2-Receive the file, verify its integrity and audit the verification data
P2.1.1 - Randomly select addresses from 16 fractions of the file to form each block of data
P3.3.2-Wait for the acceptance of the CFSMC registration in the SSTMC
File accepted
P1.3 - Calculate the number of data blocks needed to generate challenges during the storage period
P2.1.3-Read the fractions of the file, form the data blocks, generate a hash for each data block, concatenate them with a pseudo-random password, and generate the verification hashes
accept CFSMC
rejected CFSMC
File rejected
P1.2 - Divide the encrypted file into 4,096 fractions
P5.1-Delete the original file
P4.2-Send message informing the Client that the file has been rejected
P1.1-Select the file, encrypt it and create a hash for identification
P2.1.2 - Form cycles with 256 blocks of data, containing all portions of the file
P5.2-Send the address of the CFSMC, the first ICS contracting period and the information to generate challenges for that period
P5.3-Store in local DBMS, the CFSMC address of the record to be verified and the information to generate the challenges
Client
Cloud Storage Service
Integrity Check Service

FIGURE 2. Overview of the storage phase.

is accepted, the Client finalizes the protocol storage phase contracting the ICS responsible for monitoring the integrity of the file, and this process is described in Section III-B3. To conclude the description of this phase, in Section III-B4 we show the complexity analysis of the proposed algorithms. Figure 2 presents an overview of the processes performed by each of the roles in this phase.

1) PREPARING AND SENDING FILES TO BE STORED

The process is initiated with the choice by the user of the file to be stored, the cryptographic key to encrypt the file, the randomness seed, the storage period, the instance of SSTMC, the CSS, and the initial ICS. The SSTMC instance, through the “stakeholders” method, provides the list of available CSS and ICS. Then, the Client encrypts the selected file using a symmetric cryptographic algorithm and the informed cryptographic key (P1.1 of Figure 2). Next, the Client generates a hash from the encrypted file content. This hash will serve as the file identifier and integrity validator when the CSS receives the chosen file. In sequence, the Client computes the file’s fraction size dividing the referred file into 4096 portions of equal dimensions (P1.2 of Figure 2).

The protocol proposes to use challenges to validate the content of data blocks composed of 16 randomly chosen file fractions. There is a direct relation between the number of generated data blocks and the number of challenges used to verify the file integrity during its cloud storage period. As the number of generated daily challenges will vary according to the current trust level assigned to the CSS when generating these challenges, the algorithm performs the calculation of the total number of data blocks considering the worst case. In the worst case, the CSS remains evaluated at the lowest trust level, "Very high distrust'," throughout the file storage period, as shown in Table 1. Thus, the algorithm computes how many data blocks will be needed by multiplying the established file storage period in days by the number of data blocks that will be checked per day, as per Table 1.

Then, a CFSMC instance is submitted to the BN, using the following parameters for its initialization: the file identification hash, the file's fraction size, the storage deadline, the total number of data blocks that will have their verification hashes stored in the CFSMC, the credential addresses that identify the chosen CSS and ICS in the BN, and the adopted SSTMC instance address. After the BN processes the transaction generated by the CFSMC submission, the Client sends the added CFSMC instance access address and the encrypted file to the CSS. In Section IV-B2, we describe the CFSMC implementation.

At the same time, using a seed of randomness the file owner has assigned, the Client draws 16 numbers, between 0 and 4095, forming a set of numbers that represents the addresses of 16 file fractions. The algorithm repeats this drawing process 256 times, disregarding in the subsequent draw the fraction addresses that were previously drawn so that each fraction address in the file is part of a single fraction address set (FAS). The algorithm will repeat this process as a whole until it draws the number of necessary FASs to generate all required data blocks (P2.1.1 of Figure 2). The union of these 256 drawn FASs, which includes all 4096 file fraction addresses, is called a "cycle". Listing 1 presents a pseudo-code to demonstrate the process of drawing fraction addresses for each data block and generating the necessary cycles according to storage time.

For each FAS, the algorithm reads the contents of the 16 fractions from the encrypted file and concatenates these contents forming the respective data block (P2.1.3 of Figure 2). After this, the Client generates a first hash from the data block's content and a second hash, called "challenge password", from the result of the following content concatenation: i) the password informed by the Client user; ii) a random value generated from the seed of randomness; and, iii) the FAS that gave rise to the data block. Then, the Client concatenates the data block hash with the "challenge password" hash and, from the resulting content, it generates a new hash called "verification hash". For each data block generated, the algorithm inserts a record containing the FAS, the "challenge password", the "verification hash", and a sequential integer that identifies the data block in the file verification table (FVT) (P3.3.1 of Figure 2).

```
public List<Object> generateCycles(Integer
    numberOfYearsToStorage) {
  Integer numberOfCycles =
      computeNumberOfCyclesAccordingToStorageTime(
      numberOfYearsToStorage);
  List<Integer> fas;
  List<Object> cycle;
  List<Integer> addressesAlreadyDrawn;
  List<Object> cycles = new ArrayList<>();
  while (cycles.size() < numberOfCycles) {
     addressesAlreadyDrawn = new ArrayList<>();
     cycle = new ArrayList<>();
     for (int iFas=0; iFas < 256; iFas++) {
        fas = new ArrayList<>();
        for (int i=0; i < 16; i++) {
           fas.add(i, drawNotRepeated(0, 4095,
               addressesAlreadyDrawn));
           addressesAlreadyDrawn.add(fas.get(i));
        }
        cycle.add(iFas, fas);
     }
     cycles.add(cycle)
  }
  return cycles;
}
```

LISTING 1. Pseudo-code to draw fraction addresses and generate cycles.

At the same time, the Client begins the process of inserting the data block verification hashes into the submitted file's CFSMC instance stored in the BN by executing its "insertBlock" method. The hash insertions can be performed one-by-one or grouped to obtain better performance. The maximum number of data block verification hashes to be inserted in the CFSMC in a single execution of the "insertBlock" method depends on the maximum limit of bytes that a single transaction can process. This condition varies according to the characteristics of each BN.

2) AUDITING AND ACCEPTING THE FILE STORAGE REQUESTS

After receiving the entire content of the file, the CSS verifies its integrity. For this, the CSS generates a file content hash and compares it with the file identifier previously registered at the CFSMC instance stored in the BN and referred by the address received together with the file. If the file integrity verification fails, the CSS rejects its storage and informs the Client. Once the file integrity is confirmed, the CSS awaits the completions of the transactions generated in the BN when the verification hashes were inserted in the file's CFSMC instance. To do this, the CSS periodically consults the CFSMC instance by executing the CFSMC's "isReady" method, which compares the total number of data blocks reported in the referred instance insertion in the BN with the number of verification hashes already inserted.

After identifying the completion of the data block's verification hash insertions in the CFSMC instance linked to the file, the CSS draws some data blocks and, from these, generates challenge requests to the Client for auditing purposes. The CSS records these challenge requests in the file's CFSMC instance by executing the CFSMC's "requestChallenge" method. The total number of required challenges will be equal to one-tenth of the number of cycles generated by the Client. As each cycle has 256 FASs, and the necessary information to build one challenge was produced for each FAS, the audit process will use only one challenge of each set of 2560 potential challenges.

As long as the CSS does not accept the file submitted for storage or receives a message indicating its rejection, the Client monitors the file's CFSMC instance and waits for the challenge requests for audit, a procedure performed through the periodic execution of the CFSMC's "getRequestedChallenges" method. After the CSS chooses and registers the challenge requests, the Client reads these requests and obtains the id, the FAS, and the "challenge password" of all data blocks whose challenges were requested from the FVT. From this information, the Client generates and submits the challenges to file's CFSMC instance through the "submitChallenge" method. When the CFSMC instance receives the submission of a challenge from the list of challenges requested by the CSS, it automatically removes it from this list.

After requesting the audit challenges, the CSS starts monitoring the file's CFSMC instance awaiting the submission of these challenges by the Client. For this, the CSS periodically performs the CFSMC "getPendingChallenges" method. When identifying the existence of pending challenges in the file's CFSMC instance, the CSS generates a response for each registered challenge and records it in the referred CFSMC instance by executing the CFSMC "replyChallenge" method. We describe the process of generating the response to the challenge in Section III-C.

Then, the CSS monitors the file's CFSMC instance once again, awaiting the completion of the response transactions. After the BN has processed all transactions, the CSS verifies whether there are either pending challenge requests or wrong responses. This procedure is performed using both the CFSMC "getRequestedChallenges" and "getTotalFailedChallenges" methods. The audit is successful if the first one returns an empty set, indicating that the Client submitted all requested challenges, and the second returns zero, indicating that the file's CFSMC instance successfully validated all challenges responses.

Subsequently, if the audit process is successful, the CSS saves the Client's received file in its storage infrastructure and records the acceptance of the respective storage contract (the file's CFSMC instance) in the Client's chosen SSTMC instance. For this, the CSS loads the SSTMC instance from the access address received from the file's CFSMC instance, and executes the SSTMC "authorizeContract" method, using the file's CFSMC instance access address as a parameter. The CSS obtains the Client's chosen SSTMC instance access address by executing the CFSMC "getTrustManagementContract" method.

The registration of the CFSMC instance authorization by the CSS in the SSTMC instance has two objectives: i) it guarantees that the CSS accepted the file's storage SC (CFSMC instance) registered at the BN by the Client; and, ii) it authorizes the SSTMC instance to accept the trust level update requests attributed to the CSS from the referred CFSMC instance.

3) CONTRACTING THE FILE INTEGRITY MONITORING SERVICE

After the audit challenges required by the CSS are submitted, the Client monitors the chosen SSTMC instance, awaiting the registration of the authorization of the CFSMC instance linked to its file. For this, the Client periodically executes the SSTMC "isAuthorized" method. After approval, the Client registers in its database that the CSS has accepted its file, deletes the original file, and excludes its verification hashes from the FVT (P5.1 of Figure 2).

The Client then sends the stored file's CFSMC instance access address, the monitoring contract deadline, and the set of the necessary information to generate the challenges during the contracted period to the chosen ICS. This information is obtained from the FVT and is composed of the FAS, the "challenge password", and the sequential number that identifies the data block. The ICS stores all information received in its local database (P5.3 of Figure 2).

At the end of the monitoring period that the Client contracted with the ICS, the Client analyzes the records generated by the ICS integrity verification processes. If the Client identifies an inconsistency in the protocol execution, for example, if the ICS fails to inform the Client of an identified failure or submits fewer challenges than expected for the trust level assigned to the CSS, the Client then selects a new ICS. Otherwise, the Client automatically renews the expired contract. In both situations, the Client updates the file's CFSMC instance by executing its "changeIntegrityCheckService" method. After defining the ICS for the next monitoring period, the Client records the new integrity verification contract deadline in the file's CFSMC instance by executing the "setIntegrityCheckAgreementDue" method. Finalizing this process, the Client sends enough information to the ICS to generate the challenges to the CSS throughout the new contracted period. This information (FAS, "challenge password", and block identifier), originating from the FVT, is excluded from the referred table, immediately after being sent to the ICS.

4) ALGORITHM COMPLEXITY ANALYSIS

During the storage phase, we used two algorithms. The first algorithm selected the file's FAS used to assemble the data blocks applied in preparing the challenges and in verifying integrity. In addition, the algorithm groups the selected FAS into cycles, so that each cycle contains all file fractions. The algorithm input parameter is the number of cycles to be generated, which is directly related to the number of years established for storage in the cloud.

The complexity analysis of the algorithm begins by defining the number of operations performed according to its input parameter. This number is equal to $16*256*n$, where 16 is the number of fraction addresses that make up each FAS, 256 is the number of FASs that make up each cycle, and n represents the number of cycles to be generated. The n is the integer value resulting from the calculation of $14*366*m/256$, where the 14 is the number of data blocks to be checked per day considering the worst-case, i.e., the CSS classified at the "Very high distrust" level, 366 is the maximum number of days in a year, m represents the number of years established for file storage, and 256 is the number of generated data blocks per cycle. Based on the analysis presented, it is possible to infer that the complexity of this algorithm is $O(n)$, even when considering the number of years established for storing the file as an input parameter.

The second algorithm processes the generated cycles and assembles the data blocks with the concatenation of the content of 16 file fractions according to the addresses registered in each of the 256 FASs in each cycle. Next, for each generated data block, the algorithm generates a verification hash. In this algorithm, the input parameter is the number of cycles generated by the first algorithm. Considering that the content reading of the file fractions is the operation with the highest number of repetitions, the maximum number of operations performed by the algorithm is equal to $16 * 256 * n$, where 16 is the number of fraction addresses that make up each FAS, 256 is the number of FASs that make up each cycle, and n is the total number of generated cycles. As in the first algorithm, we can conclude that the complexity of the second algorithm is also $O(n)$. Based on the results shown, we can affirm that both algorithms used in this study presented a linear growth rate over the period of computation, according to the number of years established for storing the file in the cloud.

C. INTEGRITY CHECK PHASE

This phase is executed in parallel by the ICS, which generates the challenges and records them in the BN. The CSS periodically checks the pending challenges, processes them, and records their responses in the BN. Both the ICS and CSS interact with the BN through the CFSMC instance, which updates the level of trust in the CSS through an SSTMC instance. An overview of the processes performed by each of the roles in this phase is presented in Figure 3.

The ICS reads its database daily, locating the active integrity verification contracts, grouping the respective CFSMC instances by the CSS, and by the SSTMC instance used (P1 of Figure 3). Then, the ICS initializes two processes in parallel. The first process, presented in Section III-C1, selects the CFSMC instances by the CSS/SSTMC that will receive challenges on that day, calculates the number of challenges to submit for each CFSMC instance according to the CSS trust level registered at the SSTMC instance, generates the challenges, and submits them to the selected CFSMC instance. The second process, presented in Section III-C2, verifies in each CFSMC instance if there are pending challenges and if there are flaw indications in the previously submitted challenge records.

After the ICS submits challenges to the CFSMC instance, the CSS reads them and begins the process of generating and submitting challenge responses to the CFSMC instance, according to the proceedings presented in Section III-C3. Then, the CFSMC instance verifies these answers and, according to the obtained results, it requires an update in the trust level assigned to the CSS from the SSTMC instance. Section III-C4 presents the calculations used to update the trust level.

1) CHALLENGE GENERATION AND SUBMISSION PROCESS

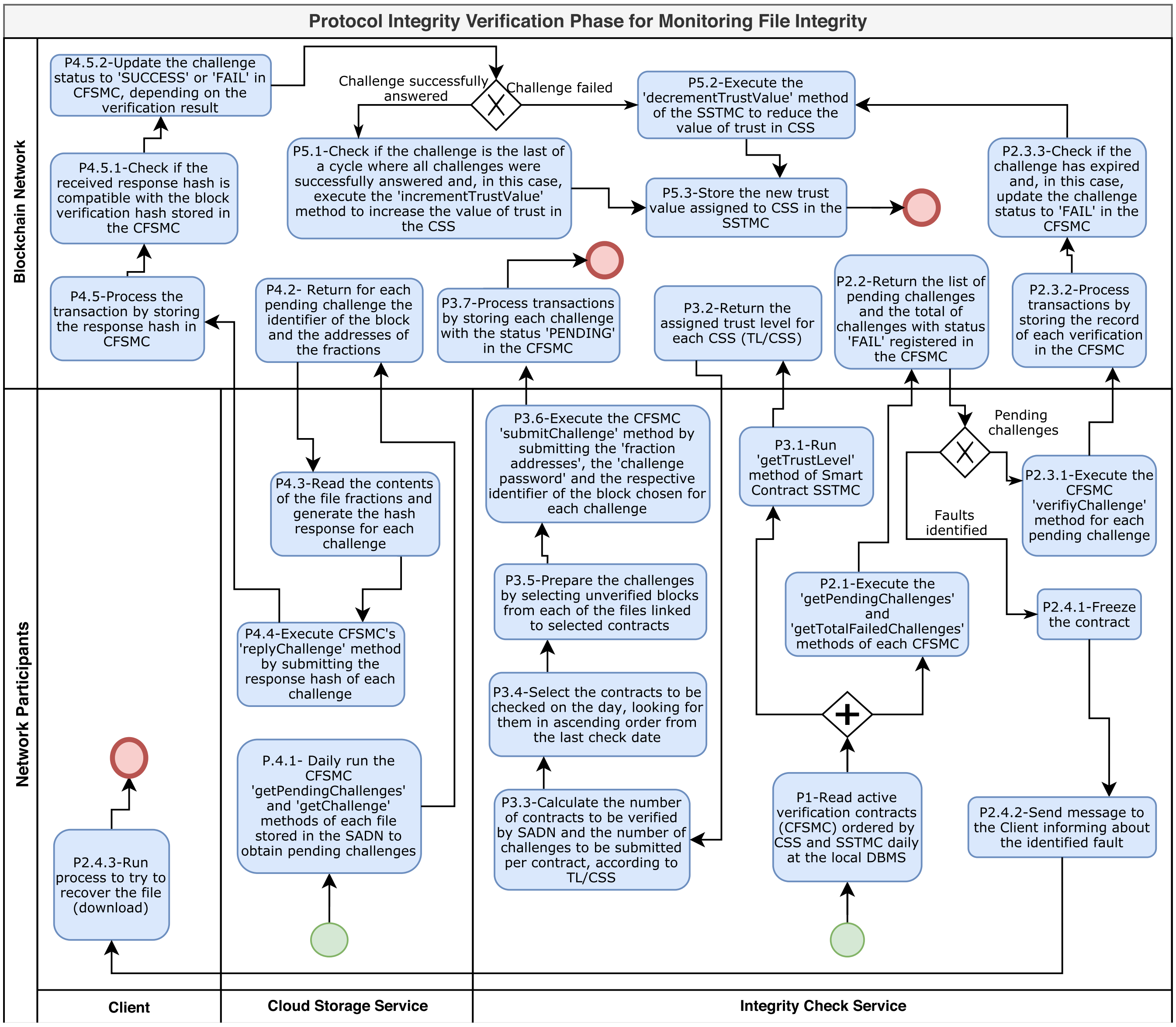
The same CSS provider can store files from Clients using CFSMCs linked to distinct SSTMC instances, and the same ICS provider can monitor CFSMC instances of files stored in different CSSs and linked to various SSTMC instances. Each SSTMC instance will assign different values to the trust level in each CSS, and theses values will define the number of stored files verified by day and the number of challenges by file. So, for calculating the total number of files, the protocol only considers the CFSMC instances linked to the same CSS and SSTMC instance (CSS/SSTMC). For each CSS/SSTMC, the ICS performs the challenge generation process with its active contracts.

From the selection of the CSS/SSTMC, the ICS executes the SSTMC "getTrustLevel" method to obtain the updated trust level assigned to the CSS (TL/CSS) (P3.1 and P3.2 of Figure 3). Then, the ICS computes the number of files to submit the challenges to, and the number of challenges to submit for each file (P3.3 of Figure 3), both using the total number of files linked to the CSSs/SSTMCs with active contracts, and the percentages from Table 1.

The ICS selects which CFSMC instances will receive challenges on the day, in ascending order of date/time of the last challenge submitted for each CFSMC instance (P3.4 of Figure 3). Then, for each CFSMC instance selected, from information on data blocks received from the Client, the ICS selects the data blocks to generate the challenges from unused data blocks in the last cycle. When there are no data blocks available in the current cycle, the ICS draws a new cycle, except in the first execution, when the ICS selects the cycle zero (P3.5 of Figure 3). Then, the ICS submits a challenge for each chosen data block by executing the CFSMC "submitChallenge" method, using the FAS, the "challenge password", and the data block identifier as parameters (P3.6 of Figure 3). The CFSMC instance processes these transactions and stores the received challenges, designating them as "PENDING", which means that the challenge is awaiting the CSS response (P3.7 of Figure 3). Listing 2 presents a pseudo-code to demonstrate the ICS process of generating and submitting challenges to each CSS through the CFSMC instance.

2) PREVIOUS CHALLENGE VERIFICATION PROCESS

For each active verification contract, the ICS runs the CFSMC "getPendingChallenges" method, which returns a list of the pending challenge identifiers (P2.1 of Figure 3).



Protocol Integrity Verification Phase for Monitoring File Integrity
Blockchain Network
Network Participants
Client
Cloud Storage Service
Integrity Check Service
P4.5.2-Update the challenge status to 'SUCCESS' or 'FAIL' in CFSMC, depending on the verification result
Challenge successfully answered
Challenge failed
P5.2-Execute the 'decrementTrustValue' method of the SSTMC to reduce the value of trust in CSS
P4.5.1-Check if the received response hash is compatible with the block verification hash stored in the CFSMC
P5.1-Check if the challenge is the last of a cycle where all challenges were successfully answered and, in this case, execute the 'incrementTrustValue' method to increase the value of trust in the CSS
P5.3-Store the new trust value assigned to CSS in the SSTMC
P2.3.3-Check if the challenge has expired and, in this case, update the challenge status to 'FAIL' in the CFSMC
P4.5-Process the transaction by storing the response hash in CFSMC
P4.2- Return for each pending challenge the identifier of the block and the addresses of the fractions
P3.7-Process transactions by storing each challenge with the status 'PENDING' in the CFSMC
P3.2-Return the assigned trust level for each CSS (TL/CSS)
P2.2-Return the list of pending challenges and the total of challenges with status 'FAIL' registered in the CFSMC
P2.3.2-Process transactions by storing the record of each verification in the CFSMC
P3.6-Execute the CFSMC 'submitChallenge' method by submitting the 'fraction addresses', the 'challenge password' and the respective identifier of the block chosen for each challenge
P3.1-Run 'getTrustLevel' method of Smart Contract SSTMC
Pending challenges
P2.3.1-Execute the CFSMC 'verifiyChallenge' method for each pending challenge
Faults identified
P4.3-Read the contents of the file fractions and generate the hash response for each challenge
P3.5-Prepare the challenges by selecting unverified blocks from each of the files linked to selected contracts
P2.1-Execute the 'getPendingChallenges' and 'getTotalFailedChallenges' methods of each CFSMC
P2.4.1-Freeze the contract
P4.4-Execute CFSMC's 'replyChallenge' method by submitting the response hash of each challenge
P3.4-Select the contracts to be checked on the day, looking for them in ascending order from the last check date
P.4.1- Daily run the CFSMC 'getPendingChallenges' and 'getChallenge' methods of each file stored in the SADN to obtain pending challenges
P3.3-Calculate the number of contracts to be verified by SADN and the number of challenges to be submitted per contract, according to TL/CSS
P1-Read active verification contracts (CFSMC) ordered by CSS and SSTMC daily at the local DBMS
P2.4.2-Send message to the Client informing about the identified fault
P2.4.3-Run process to try to recover the file (download)

FIGURE 3. Overview of the integrity check phase.

This situation occurs when the CSS delays or chooses to postpone the response indefinitely. Taking into account that the protocol execution cycle occurs every 24 hours, asynchronously and through a decentralized network, we considered the possibility of failures in the synchronization of the Blockchain network nodes, which could cause delays both in the propagation of the transaction that registers the challenge and in the propagation of the transaction that registers the response. Therefore, we defined that the acceptable maximum propagation delay time could not be greater than 24 hours in each direction. Furthermore, based on results presented in the previous work [8], that uses a similar response generation process and after having run stress tests to be sure of the proposed solution behavior, we initially defined that 24 hours would be enough time for the CSS to process a challenge, even when considering a potential overload of challenges.

Based on the above, we concluded that 72 hours was enough time for a well-intentioned CSS to respond to a challenge, even when experiencing temporary technical difficulties. A delay greater than this would be a strong indication that the CSS was trying to hide that the file is corrupted or lost. Even if the stored file were still healthy, the lack of a proper response would indicate that the CSS is experiencing more severe problems, which would compromise the stored file availability, being a reason to penalize the CSS.

For each pending challenge, the ICS runs the CFSMC "verifyChallenge" method. When executing this method,

```
public void checkStorageService(String
    CSSCredentialAddress, String
    SSTMContractAddress) {
   List<DataBlock> dataBlocks;
   DataBlock dataBlock;
   String transactionHash;
   CFSMContract CFSMContract = new CFSMContract();
   SSTMContract SSTMContract = new SSTMContract();
   SSTMContract.load(SSTMContractAddress);
   String CSSTrustLevel = SSTMContract.
       getTrustLevel(CSSCredentialAddress);
   int percentFiles = getPercentOfFilesToVerify(
       CSSTrustLevel);
   int blocksToVerify = getBlocksByFileToVerify(
       CSSTrustLevel);
   int totalOfStorageContracts =
       getTotalOfStorageContracts(
       CSSCredentialAddress, SSTMContractAddress);
   int contractsToVerify = totalOfStorageContracts
        * (percentFiles / 100);
   StorageContract lastStorageContract =
       getLastStorageContractVerified(
       CSSCredentialAddress, SSTMContractAddress);
   List<StorageContract> storageContracts;
   while (contractsToVerify > 0) {
      storageContracts = storageContractDAO.
          findByCSSAndSSTMC(CSSCredentialAddress,
          SSTMContractAddress, lastStorageContract
          ).interator();
      StorageContract storageContract = null;
      while (storageContracts.hasNext() &&
          contractsToVerify > 0) {
         contractsToVerify--;
         int nBlocks = blocksToVerify;
         storageContract = storageContracts.next()
             ;
         CFSMContract.load(storageContract.
             getContractAddress());
         int cycle = storageContract.
             getSelectedCycle();
         while (nBlocks > 0) {
            dataBlocks = dataBlockDAO.
                getDataBlocksUnverified(
                storageContract, cycle).iterator()
                ;
            while (dataBlocks.hasNext() && nBlocks
                 > 0) {
               nBlocks--;
               dataBlock = dataBlocks.next();
               transactionHash = CFSMContract.
                   submitChallenge(dataBlock.
                   getBlockId(), dataBlock.
                   getAddressCodes(), dataBlock.
                   getPassword());
            }
            if (nBlocks > 0) {
               cycle = dataBlockDAO.getNextCycle(
                   storageContract);
               updateStorageContract(
                   storageContract, cycle);
            }
         }
         updateLastStorageContractVerified(
             CSSCredentialAddress,
             SSTMContractAddress, storageContract)
             ;
      }
      lastStorageContract = null;
   }
}
```

LISTING 2. Pseudo-code to generate and submit challenges to each CSS.

the CFMSC instance records the request in the BN and calculates how long the challenge has been pending (P2.3.1 and P2.3.2 of Figure 3). If the period is greater than 72 hours, it changes the challenge status from "PENDING" to "FAIL" (P2.3.3 of Figure 3), indicating that the challenge has failed. Next, the CFSMC instance automatically executes the SSTMC "decrementTrustValue" method, which reduces the trust level assigned to the CSS (P5.2 in Figure 3). We described the calculation of the trust level in Section III-C4.

After processing the pending challenge verification transactions, the ICS executes the CFSMC "getTotalFailedChallenges" method, which returns the total number of challenges submitted to the CFSMC instance whose registered status is "FAIL" (P2. 2 of Figure 3). If the returned value is greater than zero, this means that one or more challenges have failed, and it is not possible to warrant that the file stored by the Client in the CSS remains unadulterated. Then, the ICS immediately sends an electronic message (*e-mail*) to the Client informing the identified flaw (P2.4.2 of Figure 3) and sets the integrity verification contract status to "frozen" (P2.4.1 of Figure 3). The ICS will maintain the referred contract in a "frozen" state until the Client expresses itself about the reactivation or the definitive cancellation of the contract.

The Client, when informed about an identified fault, must download the file and check its integrity by comparing the file identification hash with the hash of file content recovered from the CSS (P2.4.3 of Figure 3). When it is confirmed that the file content remains intact in the CSS and that the identified fault is a false positive, the Client will request the ICS to reactivate the verification contract. Otherwise, the Client will request the definitive cancellation of the respective verification contract regarding that file to the ICS.

3) CHALLENGE RESPONSE GENERATION, SUBMISSION AND VERIFICATION PROCESS

Every CSS that holds active file storage contracts for the Client periodically checks for pending challenges by executing the "getPendingChallenges" method of the CFSMC instance of each stored file. This method returns a list of the pending challenge identifiers, whose number is equal to the key that identifies the data block that originated the list. For each CFSMC instance with pending challenges, the CSS executes both the CFSMC "getFileId' and "getChunkSize" methods to obtain, respectively, the file identification hash and the file fraction size. Next, the CSS processes each pending challenge. This process starts with the CSS obtaining the FAS of the data block that will be verified by executing the CFSMC "getChallenge" method, which receives the challenge identifier as parameter (P4.1 of Figure 3).

After locating the file with its hash identifier, the CSS reads the content of each file fraction indicated by the addresses contained in the FAS. To do this, the CSS calculates each file fraction's initial position by multiplying its address by the file fraction's size. Next, the CSS generates a data block by concatenating the read fraction content and generates the response hash from this data block (P4.3 of Figure 3). Finally, the CSS responds to the challenge by executing the CFSMC' "replyChallenge" method, using both the response hash and the challenge identifier as parameters (P4.4 of Figure 3).

When processing the transaction containing the response to the challenge, the CFSMC instance stores the response hash and reads the "verification hash" from its database recorded by the Client when it sent the file for storage in the cloud, and the "challenge password" registered by the ICS when the challenge was submitted. In sequence, the CFSMC instance generates a "new hash" from the concatenation of the read "challenge password" and the received "response hash". To end this round, the CFSMC compares the generated "new hash" with the read "verification hash" (P4.5.1 of Figure 3).

A match between the compared hashes confirms the integrity of the file fractions used in the challenge and, in this case, the CFSMC instance changes the challenge status to " SUCCESS " (P4.5.2 in Figure 2 3). It also checks whether the received response refers to the last challenge in a cycle, whose other challenges also confirmed the integrity of the fractions they verified. As a challenge cycle covers all file fractions, when the CSS successfully answers all challenges in the same cycle, it is possible to state that the verified file is unbroken. Whenever it finishes a challenge cycle without finding a failure, the CFSMC asks the SSTMC instance to increase the trust level assigned to the CSS by executing the SSTMC "incrementTrustValue" method. Section III-C4 describes the performed calculation (P5.1 of Figure 3). Likewise, if the integrity of the fractions verified by the challenge is not confirmed, the CFSMC instance changes the challenge status to "FAIL" (P4.5.2 of Figure 3) and asks the SSTMC instance to reduce the trust level assigned to the CSS by executing the SSTMC "decrementTrustValue" method (P5.2 of Figure 3).

4) CALCULATION OF THE TRUST VALUE

We adapted both the SSTMC "incrementTrustValue" and "decrementTrustValue" method implementation in the trust calculation model proposed in the previous study [8] to allow its utilization in a SC. Due to the limitation of the Solidity programming language [20] used to implement SC, the trust level calculation used only integer numbers. However, for the sake of clarity, some values will be presented in scientific notation format according to their magnitude.

The trust value assigned to the CSS (TV/CSS) is an integer value, belonging to the interval between -1×10^{20} and 1×10^{20}. The CFSMC instance updates the TV/CSS by executing either the SSTMC "incrementTrustValue" or the "decrementTrustValue" method according to the file integrity checking result. The CFSMC instance must ask the SSTMC instance to increase the TV/CSS whenever it concludes a file verification cycle without finding a failure. However, whenever an integrity failure is found in a file stored by the CSS, the CFSMC instance must request the SSTMC instance to reduce the TV/CSS.

The "decrementTrustValue" method implements the equation $z = x - y$, where the z is the new and updated TV/CSS, the x represents the TV/CSS stored in the SSTMC instance, and the y represents the decrement value computed according to Equation 1:

$$\begin{aligned}&(x > 0 \rightarrow y = x)\\ &\quad \wedge (x = 0 \rightarrow y = 1.5 \times 10^{19})\\ &\quad \wedge (0 > x \geq -5 \times 10^{19} \rightarrow y = (x \times -0.15))\\ &\quad \wedge (x < -5 \times 10^{19} \rightarrow y = ((-1 \times 10^{20} - x) \times -0.025)).\end{aligned} \tag{1}$$

Likewise, the "incrementTrustValue" method implements equation $z = x + y$, where the z represents the new and updated TV/CSS, the x represents the TV/CSS stored in the SSTMC, and the y represents the increment value computed according to Equation 2:

$$\begin{aligned}&(x < 0 \rightarrow y = ((-1 \times 10^{20} - x) \times -0.025)\\ &\quad \wedge (x = 0 \rightarrow y = 1.5 \times 10^{19})\\ &\quad \wedge (0 > x \geq 5 \times 10^{19} \rightarrow y = (x \times 0.025))\\ &\quad \wedge (x > 5 \times 10^{19} \rightarrow y = ((1 \times 10^{20} - x) \times 0.005)).\end{aligned} \tag{2}$$

After performing the calculation according to the respective equations, the resulting new TV/CSS, the z value in the equations, is stored in the SSTMC instance (P5.3 of Figure 3).

The constants we defined in Equations 1 and 2 represent the percentages that will be applied on the calculation basis to determine the decrease/increase value for each case. These equations use different percentages according to the current TV/CSS, so that the needed "number of failures" or the "number of cycles correctly verified" to progress to the "Medium" level of either distrust (TV/CSS $\leq -5 \times 10^{19}$) or trust (TV/CSS $> 5 \times 10^{19}$), according to Table 1, are proportionally much smaller than those needed to reach the "High" and "Very High" levels. Furthermore, in these last levels, the decrement/increment calculation, instead of using the current TV/CSS as the calculation basis, uses the difference between the maximum or minimum scale value and the current TV/CSS. Thus, the closer the TV/CSS is to the scale limits, the smaller the calculation basis and, consequently, the smaller the impact of the resulting decrease/increase in the distrust/trust level progression.

We obtained the adopted percentages in Equations 1 and 2 from the results of the simulations presented in the previous study [8]. However, in this new version, we calibrated them through preliminary tests, with the intention of improving the calculations to accelerate the penalizing of services that presented recurrent failures. This penalty is the rapid progression of their classification through the different distrust levels. In return, we ensure that service providers who maintain good behavior, i.e., that have no flaws detected over time, are gradually reclassified from the lowest to the highest trust level. Listing 3 presents Equation 1 in a pseudo-code that

```
if ( trustValue > 0 ) {
    decrement = trustValue;
} else if (trustValue == 0) {
    decrement = 1.5 * (10 ** 19);
} else if (trustValue >= (-5 * (10 ** 19)) {
    decrement = trustValue * -0.15;
} else {
    decrement = ((-1 * (10 ** 20)) - trustValue) *
         -0.025;
}
newTrustValue = trustValue - decrement;
```

LISTING 3. Trust value computing after a failure has been identified.

```
function incrementTrustValue(address
    storageServiceAddress) external {
    require(authorized[prepareAuthorizedKey(msg.
        sender, storageServiceAddress)] != 0, "
        CFSMC not authorized!");
    (, int x) = computeIncrementValue(trust[
        storageServiceAddress]);
    trust[storageServiceAddress] = x;
}

function computeIncrementValue(int x) internal
    pure returns (int y, int z) {
    int constant MIN_VERY_HIGH_DISTRUST = int(-1)
        * (int(10) ** int(20));
    int constant MAX_LOW_MEDIUM_TRUST = int(5) * (
        int(10) ** int(19));
    int constant MAX_VERY_HIGH_TRUST = int(10) **
        int(20);

    if (x < 0) {
        y = (MIN_VERY_HIGH_DISTRUST - x) * int
            (-25)) / 1000;
    } else if (x == 0) {
        y = int(15) * (int(10) ** int(18));
    } else if (x <= MAX_LOW_MEDIUM_TRUST) {
        y = (x * int(25)) / int(1000);
    } else {
        y = ((MAX_VERY_HIGH_TRUST - x) * int(5) /
            int(1000);
    }
    z = x + y;
}
```

LISTING 4. Implementation of "incrementTrustValue" method.

demonstrates the trust value computation after a CFSMC instance to identify a failure. The "incrementTrustValue" method implementation shown in Listing 4 demonstrates use of the Equation 2.

D. PROTOCOL SECURITY ANALYSIS

1) CLIENT SECURITY

The proposed protocol promotes Client security by making it possible to monitor file integrity without exposing any part of the original file's content to any third parties involved in the protocol execution (ICS, CSS, and BN). So, both the cryptographic algorithm strength and the Client password secrecy preservation used ensure confidentiality. In addition, the CSS publicly records in the BN the acceptance of the CFSMC instance linked to the stored file, through the SSTMC instance chosen by the Client. This process gives non-repudiation assurance to the Client concerning the file received by the CSS, and the CSS concordance regarding the contracted service requirements defined in the SCs CFSMC and SSTMC.

The protocol also assures the Client that none of the parties involved (ICS, CSS, BN) need to keep enough information to forge the "hash response" in their records. Moreover, the anticipated generation of all possible answers using the brute force method would have a high computational cost. Due to the "challenge password" size ($2^{256\ bits} \approx 1.15 \times 10^{77}$ possibilities) and the number of possible arrangements for the FAS assembly ($A_{4096,16} \approx 6.09 \times 10^{57}$), it became necessary to test an average of $\approx$ 3.5×10^{134} hashes to identify a FAS and a "challenge password". For this reason, the CSS needs to access a complete copy of the file to produce the correct responses to the challenges, even if there is collusion involving the CSS and ICS, or a recurrent leak from the ICS of information used to generate challenges, since in each integrity verification contract renewal the Client submits only the necessary information to generate challenges during the contracted period to the ICS.

The BN immutability ensures that rules implemented in a SC, once inserted in the BN, cannot be changed. The proposed protocol determines that the challenges generated by the ICS, as well as the respective responses generated by the CSS, must be registered at the BN through an instance of the SC CFSMC. These characteristics ensure to the Client the possibility of performing an audit, at any time, of every action performed by both the ICS and CSS. This audit process provides the Client the guarantee of the integrity of its files, without the need of the other parties involved. Besides, the fractionated challenge generation information sent to the ICS allows the Client to replace the ICS at any time, without compromising the security of the integrity verification process. In turn, the CFSMC implementation ensures that only the authorized ICS can submit challenges and that only the Client that has inserted the CFSMC instance in the BN can replace the authorized ICS.

2) CSS SECURITY

The use of the SC SSTMC to compute the trust value in the CSS and its storage prevents the influence or interference of the other roles (Client, ICS, and CSS) in the obtained results and, consequently, in the trust level assigned to each CSS. In addition, only the CFSMC instances previously approved by the CSS can ask to SSTMC instance to execute this calculation. This approach gives veracity and reliability to the computed trust value for each CSS managed by the SSTMC instance. Smart contracts guarantee calculation security due to code immutability. Furthermore, any BN participant can read and audit their codes. This architecture allows the safe sharing of the CSS trust level between all the ICSs that monitor files stored in the same CSS which have adopted the same SSTMC instance for trust management.

The protocol promotes CSS security by allowing it to previously check the rules implemented in the trust management contract options, that is, in the available SSTMC instances, allowing the CSS to only offer its services when it agrees to the implemented rules. Another proposed protocol security mechanism that we designed to protect the CSS is the audit of the data generated by the Client to verify file integrity. The CSS performs this audit before it registers the final acceptance of the CFSMC instance linked to the file submitted by the Client for storage. In this audit process, the CSS randomly chooses certain challenges registered by the Client in the CFSMC instance and tests their compatibility with the submitted file content. This approach minimizes the possibility of a malicious Client generating invalid verification information, which, if not identified at the time of file storage contract acceptance, will compromise the reputation of the CSS. Another security mechanism provided for in the protocol is the verification carried out by the CSS, before a Client file is definitely accepted for file storage, which tests whether the smart contract linked to the file is a reliable CFSMC instance (without changes).

IV. IMPLEMENTATION OF THE PROTOCOL

We organized the protocol implementation to provide a phased validation. In Section IV-A, we present the choice of programming languages and other technologies required to implement both the SCs and the applications intended for the Client, the ICS, and the CSS roles. Section IV-B describes the implementation of the SCs, the SSTMC and the CFSMC. Then, in Section IV-C1, we show the implementation of the application responsible for the processes assigned to the Client, and their integration with the SCs. Next, in Section IV-C2, we present the implementation of the functionalities assigned to the CSS and their interaction with the Client and the SCs. Finally, in Section IV-C3, we show the implementation of functionalities assigned to the ICS and their interaction with the Client and the SCs.

A. TECHNOLOGY CHOICES

The programming languages available for the implementation of the SCs are not generic; that is, they are mostly specific to the technology adopted in the BN implementation [21]. Although the proposed protocol is generic and applicable to any BN, before starting the implementation of the proposed SCs, we chose a Blockchain technology and a compatible SC programming language.

We selected the ''Ethereum'' platform [22] guided by the following criteria: the availability of documentation, the ease of creating a network to perform tests inside a laboratory, and the number of available tools to support the use and the testing execution. The programming language chosen for the development of the SCs was Solidity [20], due to its popularity, the ample documentation, and tools available that allow the smooth interaction between the implemented SCs and the applications responsible for the actions assigned to the Client, the ICS, and the CSS.

For the development of the applications that implement the processes assigned to the Client, CSS, and ICS, we chose the JAVA EE language and its components, such as JPA, EJB, CDI, and JAX-WS [23]. Since each of the roles involved in the proposed protocol has different functions, we decided to develop three applications, each of which will be responsible for implementing the responsibilities of a single role. For the Client, we opted for a local desktop-type application, while for the CSS and the ICS, we decided on web service-type applications. For these two applications, we chose the *Glassfish Application Server* [24] as the application server, and the *PostgreSQL* [25] software as the Database Management System (DBMS) for all applications.

We chose the platform adopted for the implementation of the CSS and ICS applications taking into account the need to provide asynchronous interaction between them and the Client. Based on this premise, the availability of resources for implementing web services, asynchronous communications, scheduling tasks, and monitoring events determined the choice of JAVA EE [23]. We selected Glassfish and PostgreSQL because they are both open-source and fully meet the needs required by the applications.

B. IMPLEMENTATION OF SMART CONTRACTS

When implementing SCs, we took into account that every transaction carried out in the BN has a cost, which serves to remunerate the network nodes that mine the blocks to insert into the BN, blocks where the transactions submitted to the BN by other nodes will be embedded. Some BNs, such as Ethereum [22], use a unit called ''gas'' to compute the transaction cost, and the amount is directly related to the number of bytes that will be processed/stored in the BN and the complexity of the performed calculations. At the end of the execution of a transaction, the BN converts the amount of ''gas'' consumed in the transaction execution to the adopted cryptocurrency and charges it to the node that submitted the transaction. At the transaction submission time, its node of origin offers the amount of cryptocurrency to pay for each unit of ''gas''. The node must have a sufficient balance in cryptocurrency to cover the referred transaction, or the BN will decline it.

The BN also limits the maximum amount of ''gas'' that a transaction can consume, and this amount varies according to the settings of each BN. At the time of submission, the node of origin also defines the maximum amount of ''gas'' it is willing to pay for the transaction. If the submitted transaction exceeds the ''gas'' consumption stipulated in any of the maximum limits mentioned, the BN will refuse it. Because of the above issues, we carried out a series of tests to determine the best strategy to be adopted in the SC implementation, so that the transactions result in the lowest possible consumption of ''gas'', grouping them when possible to reduce the number of individual transactions executed, but without surpassing the limits of the BN.

Section IV-B1 presents the description of the features implemented by the SC SSTMC, responsible for managing the trust level attributed to the CSS. Section IV-B2 introduces the SC CFSMC, responsible for managing storage and for monitoring files in the cloud. Figure 4 shows a class diagram of the implemented SCs.

The language Solidity [20] has a limitation since it does not allow iterating over the records of a mapping, i.e., over the elements of an array that maps keys to values using "mapping" type, but allows access to the stored value with the use of the respective key. Due to this limitation, to allow the storage and processing of information composed of key and value, we created the auxiliary classes (RequestMap, StakeHolerMap, BlockMap, and ChallengeMap) that implement the "iterateStart", "iterateNext", and "iterateValid", and "iterateGet" methods described in the abstract class "IterateMap" shown in Figure 4. These methods make it possible to read the values sequentially in the stored order. The attribute "keyIndex" passed as a parameter indicates the element's position, according to the order of its insertion in the matrix. The "contains", "insert", "remove" and "get" methods (Figure 4) receive the real "key" as a parameter. Each of these methods has the function of respectively confirming the existence of the key, inserting an element with the value informed in the attribute "data" linked to the key, deleting the element with the key, and reading the content stored in the attribute "data".

1) SMART CONTRACT SSTMC

The Storage Service Trust Management Contract (SSTMC) is a SC defined by the proposed protocol which has the following functions: to calculate, store and share the trust level assigned to the CSS; to record the acceptance by the CSS of the storage contracts (CFSMC instances) submitted by the Clients; and to publish the CSS and ICS providers willing to provide their services under the rules implemented in a SSTMC instance. The SSTMC implements the methods described in the abstract class "SSTMCAbstractContract" as can be seen in Figure 4.

In order to calculate the trust value, the SSTMC class implements the "incrementTrustValue" and "decrementTrustValue" (Figure 4) methods. Both methods receive the credential address that identifies in the BN the CSS as a parameter, whose assigned trust value must be updated. Only CFSMC instances previously authorized by the CSS can execute these methods. According to the process described in Section III-C, a CFSMC instance will respectively invoke one of these methods whenever it identifies the need to increase or decrease the trust value assigned to the CSS linked to it. We presented the calculations implemented in these methods in Section III-C4. These methods store the obtained result as an element of the 'trust' attribute (Figure 4), a mapping whose key is the CSS credential address at the BN. Listing 4 presents the SSTMC "incrementTrustValue" method implementation.

Aiming to share the trust attributed to a CSS, we implemented the "getTrustLevel" and "getTrustValue" methods (Figure 4), both of which receive the CSS credential address as a parameter. The ICS performs the first method (Section III-C) daily, which returns the trust level assigned to the CSS. The ICS obtains the trust level by converting the trust value stored in the attribute 'trust' into one of the trust levels, according to the limits defined in Table 1. The second method, which returns the trust value stored in the 'trust' attribute, is used by the Client application to classify all available CSS trust values in decreasing order (Section III-B).

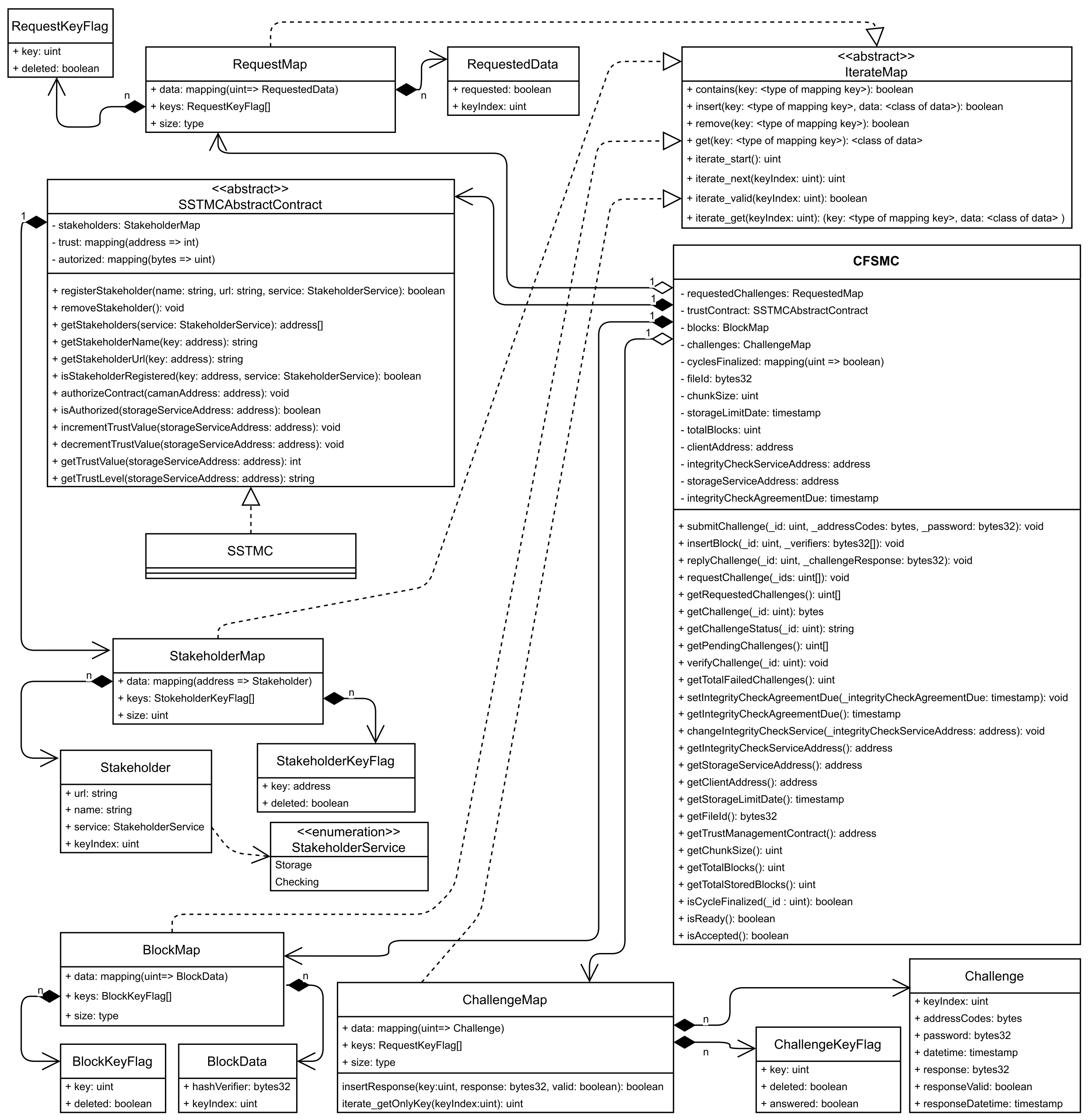
After the CSS validates a received file for storage, it authorizes the SSTMC instance, chosen by the Client, to accept requests from the CFSMC instance linked to that file using the "authorizeContract" method. The authorization also confirms the CSS acceptance of the Client storage request. The "authorizeContract" method receives the CFSMC access address at the BN as parameter. Only CSSs that have been previously self-registered in the SSTMC instance can execute this method. When performing this method, the SSTMC instance stores a new element in the 'authorized' attribute, a mapping whose key is the received CFSMC access address concatenated with the credential address of the CSS that called the process, and the value is the number one. Once the authorization is registered, it cannot be undone by any of the roles, as the SSTMC does not implement this functionality for security reasons.

The procedure for disclosing the CSSs and ICSs interested in providing their services begins with self-registration using the "registerStakeholder" method. When executing this method, the service provider informs as parameters a name for its identification, the address of the "*web service*" through which the Client will interact with its services, as well as the type of service it wishes to provide, either "Storage" (CSS) or "Checking" (ICS), as defined in the enumeration "StakeholderService". The information received is stored in a new element in the "stakeholders" attribute, a mapping managed through an object of the "StakeholderMap" class, whose key is the service provider's credential address at the BN.

The "getStakeholders" method publishes the registered service providers returning a list with the credentials address of the service providers according to the service type informed as a parameter ("Storage" or " Checking"). To enable access to the name and the web service URL of the service provider, the SSTMC implements respectively the "getStakeholderName" and "getStakeholderUrl" methods, both of which receive the provider's credential address at the BN as a parameter. The "isStakeholderRegistered" method (Figure 4) accepts a provider credential address and a type of service ("Storage" or "Checking") as parameters, and its execution checks whether the provider to which the credential address belongs is still registered at the SSTMC instance to provide the specific service. If a provider no longer wishes to advertise its services for one particular instance of the SSTMC, the execution of the "removeStakeholder" method excludes it from the list of registered providers.

2) SMART CONTRACT CFSMC

The *Cloud File Storage and Monitoring Contract* (CFSMC) is a SC defined by the proposed protocol which has the



RequestKeyFlag
+ key: uint
+ deleted: boolean
RequestMap
+ data: mapping(uint=> RequestedData)
+ keys: RequestKeyFlag[]
+ size: type
RequestedData
+ requested: boolean
+ keyIndex: uint
<<abstract>>
IterateMap
+ contains(key: <type of mapping key>): boolean
+ insert(key: <type of mapping key>, data: <class of data>): boolean
+ remove(key: <type of mapping key>): boolean
+ get(key: <type of mapping key>): <class of data>
+ iterate_start(): uint
+ iterate_next(keyIndex: uint): uint
+ iterate_valid(keyIndex: uint): boolean
+ iterate_get(keyIndex: uint): (key: <type of mapping key>, data: <class of data>)
<<abstract>>
SSTMCAbstractContract
- stakeholders: StakeholderMap
- trust: mapping(address => int)
- autorized: mapping(bytes => uint)
+ registerStakeholder(name: string, url: string, service: StakeholderService): boolean
+ removeStakeholder(): void
+ getStakeholders(service: StakeholderService): address[]
+ getStakeholderName(key: address): string
+ getStakeholderUrl(key: address): string
+ isStakeholderRegistered(key: address, service: StakeholderService): boolean
+ authorizeContract(camanAddress: address): void
+ isAuthorized(storageServiceAddress: address): boolean
+ incrementTrustValue(storageServiceAddress: address): void
+ decrementTrustValue(storageServiceAddress: address): void
+ getTrustValue(storageServiceAddress: address): int
+ getTrustLevel(storageServiceAddress: address): string
SSTMC
CFSMC
- requestedChallenges: RequestedMap
- trustContract: SSTMCAbstractContract
- blocks: BlockMap
- challenges: ChallengeMap
- cyclesFinalized: mapping(uint => boolean)
- fileId: bytes32
- chunkSize: uint
- storageLimitDate: timestamp
- totalBlocks: uint
- clientAddress: address
- integrityCheckServiceAddress: address
- storageServiceAddress: address
- integrityCheckAgreementDue: timestamp
+ submitChallenge(_id: uint, _addressCodes: bytes, _password: bytes32): void
+ insertBlock(_id: uint, _verifiers: bytes32[]): void
+ replyChallenge(_id: uint, _challengeResponse: bytes32): void
+ requestChallenge(_ids: uint[]): void
+ getRequestedChallenges(): uint[]
+ getChallenge(_id: uint): bytes
+ getChallengeStatus(_id: uint): string
+ getPendingChallenges(): uint[]
+ verifyChallenge(_id: uint): void
+ getTotalFailedChallenges(): uint
+ setIntegrityCheckAgreementDue(_integrityCheckAgreementDue: timestamp): void
+ getIntegrityCheckAgreementDue(): timestamp
+ changeIntegrityCheckService(_integrityCheckServiceAddress: address): void
+ getIntegrityCheckServiceAddress(): address
+ getStorageServiceAddress(): address
+ getClientAddress(): address
+ getStorageLimitDate(): timestamp
+ getFileId(): bytes32
+ getTrustManagementContract(): address
+ getChunkSize(): uint
+ getTotalBlocks(): uint
+ getTotalStoredBlocks(): uint
+ isCycleFinalized(_id : uint): boolean
+ isReady(): boolean
+ isAccepted(): boolean
StakeholderMap
+ data: mapping(address => Stakeholder)
+ keys: StokeholderKeyFlag[]
+ size: uint
Stakeholder
+ url: string
+ name: string
+ service: StakeholderService
+ keyIndex: uint
StakeholderKeyFlag
+ key: address
+ deleted: boolean
<<enumeration>>
StakeholderService
Storage
Checking
BlockMap
+ data: mapping(uint=> BlockData)
+ keys: BlockKeyFlag[]
+ size: type
BlockKeyFlag
+ key: uint
+ deleted: boolean
BlockData
+ hashVerifier: bytes32
+ keyIndex: uint
ChallengeMap
+ data: mapping(uint=> Challenge)
+ keys: RequestKeyFlag[]
+ size: type
insertResponse(key:uint, response: bytes32, valid: boolean): boolean
iterate_getOnlyKey(keyIndex:uint): uint
ChallengeKeyFlag
+ key: uint
+ deleted: boolean
+ answered: boolean
Challenge
+ keyIndex: uint
+ addressCodes: bytes
+ password: bytes32
+ datetime: timestamp
+ response: bytes32
+ responseValid: boolean
+ responseDatetime: timestamp
n
1

FIGURE 4. Class diagram of the implemented smart contracts.

following duties: to store the information generated by the Client to validate the integrity of the file; record the information about the contract for monitoring the integrity of the file stored in the cloud; register the challenges submitted to the CSS by the ICS; receive, register and validate the responses to the challenges provided by the CSS; request the SSTMC instance to update the trust level assigned to the CSS according to the results obtained by the monitoring.

The process of storing information on the file starts with the insertion of the CFSMC instance containing the file information stored in the "fileId", "chunkSize", "storageLimitDate", "totalBlocks", "storageServiceAddress", "integrityCheckServiceAddress", and "trustContractAddress" attributes into the BN (Figure 4). These attributes respectively represent the file hash identifier, the size of each fraction of the data blocks, the total number of data blocks generated by the Client, the CSS credential address at the BN, the credential address at the BN of the ICS chosen for the first period of the integrity verification contract, and the access address at the BN of the SSTMC instance chosen by the Client to manage the trust assigned to the CSS.

Aiming to complete the storage of the set of information necessary to validate the file's integrity, we implemented the "insertBlock" (Figure 4) method whose responsibility is to receive the "verification hashes" from the Client and save them in the BN. The Client successively executes this method according to the number of verification data blocks, which are calculated by the Client, starting from the first to the last data block, using as parameters the data block identifier of the first "verification hash" that will be inserted (parameter "_id"), and an array containing 64 "verification hashes" (parameter "_verifiers"). The CFSMC inserts each received hash as a new element into the "blocks" attribute, a mapping managed through an object of the "BlockMap" class, whose value is an element of the "_verifiers" array, and whose key is the position of this element summed with the "_id" parameter. The number of hashes inserted into the CFSMC instance per method execution is limited to the maximum consumption of "gas" allowed by the BN per transaction.

The CFSMC instance blocks the execution of the "insertBlock" method when the number of "verification hashes" stored is equal to the value registered at the "totalBlocks" attribute. In this situation, the "isReady" method (Figure 4) returns the value "true", indicating that the Client has completed the CFSMC preparation, and is ready to be audited. The "getTotalBlocks" and "getTotalStoredBlocks" methods (Figure 4) respectively return the total number of blocks registered at the "totalBlocks" attribute and the total number of "verification hashes" already received and stored in the "blocks" instance. Listing 5 presents the CFSMC "insertBlock" method implementation.

```
function insertBlock(uint _id, bytes32[] calldata
    _verifiers) external {
    require(msg.sender == clientAddress, "Only the
         Client that created this CFSMC can insert
         new blocks!");
    require((_id + _verifiers.length - 1) <=
        totalBlocks, "Last Block Id must be less
        than total of blocks informed when this
        CFSMC was created!");
    require(blocks.size <= totalBlocks, "All
        blocks permitted were already informed!");
    uint id = _id;
    for (uint i=0; i<_verifiers.length; i++) {
        blocks.insert(id, _verifiers[i]);
        id++;
    }
}
```

LISTING 5. Implementation of "insertBlock" method.

In order to record the deadline of the next integrity verifying period, we implemented the "setIntegrityCheckAgreementDue" method (Figure 4), which receives the deadline timestamp as a parameter and stores it into the "integrityCheckAgreementDue" attribute. The "changeIntegrityCheckServiceAddress" method (Figure 4) allows the Client to replace the contracted ICS by declaring the new ICS credential address as a parameter. Only the Client that has inserted the CFSMC instance in the BN may perform these methods. The contracted ICS credential address and integrity verifying contract deadline can be consulted respectively by executing the "getIntegrityCheckServiceAddress" and "getIntegrityCheckAgreementDue" methods (Figure 4).

Only the registered ICS or Client who has inserted the CFSMC instance in the BN can submit challenges. For this, we implemented the "submitChallenge" method, which receives the following parameters: i) the identifier of the data block used by the challenge ("_id"); ii) the FAS whose contents make up the data block to be checked ("_addressCodes"); and, iii) the "challenge password" ("_password"). The parameter "_addressCodes" is assembled from the concatenation of each fraction address in hexadecimal format (from "000" to "FFF"). The received challenge is stored as a new element in the "challenges" attribute, a mapping managed through a "ChallengeMap" class object, whose key is the parameter "_id" and value is an object of the "Challenge" class. This "Challenge" object receives the FAS in the "addressCodes" attribute, the "challenge password" in the "password" attribute, and the moment timestamp of challenge receipt in the "datetime" attribute. Listing 6 presents the CFSMC "submitChallenge" method implementation.

```
function submitChallenge(uint _id, bytes calldata
    _addressCodes, bytes32 password) external {
    require(msg.sender == clientAddress || msg.
        sender == integrityCheckServiceAddress, "
        Only the CFSMC owner Client or the selected
         ICS can submit challenges!");
    require(!challenges.contains(_id), "Challenge
        for block id was already submitted before!"
        );
    require(blocks.contains(_id), "Block id for
        this challenge was not found!");
     challenges.insert(_id, _addressCodes, password
        );
     requestedChallenges.remove(_id);
}
```

LISTING 6. Implementation of "submitChallenge" method.

The "getChallenge" and "getChallengeStatus" methods (Figure 4) provide access to information on each challenge registered at the CFSMC, and both receive the data block identifier used in the challenge as a parameter. The "getChallenge" method returns the FAS, and the "getChallengeStatus" method returns the challenge status "PENDING" if the answer is not registered; "SUCCESS" if the recorded response is judged valid by the CFSMC instance; and "FAIL" if the answer is considered invalid or if the challenge's validity period has expired (Section III-C2). The CFSMC also implements the "getPendingChallenge" and "getTotalFailedChallenges" (Figure 4) methods, which respectively return a list with the data block identifiers of the pending challenges registered at the CFSMC, and the number of challenges with invalid or expired answers.

The more important duties of the SC CFSMC are to receive, record, and validate the responses received from the CSS. We implemented these duties through the "replyChallenge" method (Figure 4), which receives the data block identifier ("_id") and the "response hash" ("_challengeResponse") as parameters. When executed, after it validates the response hash, according to what is described in Section III-C3, it updates the element of the "challenges" attribute whose key is equal to the data block identifier. This attribute is a mapping managed by an object of the "ChallengeMap" class (Figure 4), whose value is an object of the "Challenge" class. This object receives the response hash in its"response" attribute, the result of the validation in its"responseValid" attribute, and the moment timestamp of response in its "responseDatetime" attribute. Only the CSS registered at the CFSMC instance can execute this method.

According to the protocol (Section III-C3), to update the trust level assigned to the CSS after both the validation and registration of the response, the CFSMC instance calculates the cycle to which the verified data block belongs using the formula: $x = TRUNC(y/256)$, where y is the data block identifier, and x is the cycle number (current cycle). Next, the CFSMC instance determines whether it has already finalized the referred cycle. For this, we implemented the "isCycleFinalize" method (Figure 4), which receives the current cycle as a parameter and, using it as a key, returns the value registered at the mapping stored in the "cyclesFinalized" attribute. If the returned value is "true", the current cycle has already been finalized, and the CFSMC instance may terminate this validation process. Otherwise, the process proceeds according to the result of validating the CSS response.

If the challenge's response has been considered invalid, the CFSMC executes the SSTMC "decrementTrustValue" method (Figure 4). Otherwise, the CFSMC executes the SSTMC "incrementTrustValue" method (Figure 4) only if all responses for the 256 challenges that belong to the same cycle have been received and validated successfully. Both these methods receive the CSS credential address registered at the "storageServiceAddress" attribute as a parameter. Whenever either of these methods is executed, the CFSMC registers the conclusion of the verification of the current cycle, so that there is no duplicate updating of the trust level within the same cycle. For this, it inserts a new element with value "true" in the 'cyclesFinalized" attribute, a mapping whose key is the current cycle number. Listing 7 presents the CFSMC "replyChallenge" method implementation.

The CFSMC also implements other support features, such as the "verifyChallenge" method (Figure 4), which induces the CFSMC instance to check whether the validity of a pending challenge has expired. In this case, the CFSMC instance records the failure and requests the SSTMC instance to update the trust level assigned to the CSS. This method, which can only be executed by the ICS and the Client, receives the data block identifier as a parameter. When executed, the CFSMC instance confirms the pending issue, executing the "getChallengeStatus" method (Figure 4), using the received data block identifier as a parameter. If the returned result is different from "PENDING", the process is terminated. Otherwise, to test whether the challenge has expired, the CFSMC instance executes the "get" method of the "ChallengeMap" class object stored in the "challenges" attribute, using the data block identifier as a parameter, which returns an object of the "Challenge" class with the challenge information.

```
function replyChallenge(uint _id, bytes32
    _challengeResponse) external {
    require(msg.sender == storageServiceAddress, "
        Only the Storage Service selected can
        reply challenges!");
    require(challenges.contains(_id), "Challenge
        id was not found!");
    require(challenges.data[_id].responseDatetime
        == 0, "This Challenge id was already
        replied!");
    challenges.insertResponse(_id,
        _challengeResponse, checkResponse(blocks.
        data[_id].hashVerifier, _challengeResponse
        , challenges.data[_id].password));
    uint cycle = computeCycle(_id);
    if (!cyclesFinalized[cycle]) {
        if (challenges.data[_id].responseValid) {
            if (checkIncrementTrust(cycle)) {
                SSTMContract.incrementTrustValue(
                    storageServiceAddress);
                cyclesFinalized[cycle] = true;
            }
        } else {
            SSTMContract.decrementTrustValue(
                storageServiceAddress);
            cyclesFinalized[cycle] = true;
        }
    }
}
```

LISTING 7. Implementation of "replyChallenge" method.

Then, the CFSMC instance compares the timestamp stored in the "datetime" attribute of the "Challenge" object with the timestamp of the challenge verification transaction itself. If the time difference exceeds the 72 hours granted in the protocol (Section III-C2), the CFSMC instance sets a "false" value in the "responseValid" attribute and the timestamp of this transaction in the "responseDatetime" attribute. The CFSMC instance updates the challenge in the "challenges" attribute by executing the "Insert" method of the "ChallengeMap" object, using the data block identifier as a parameter and the updated "Challenge" object (Figure 4). The CFSMC instance then executes the "decrementTrustValue" method of the SSTMC, using the CSS credential address registered at the attribute "storageServiceAddress" as a parameter. Finally, it calculates the number of the cycle to which the pending challenge data block identifier belongs and records its conclusion in the "cyclesFinalized" attribute. Listing 8 presents the CFSMC "verifyChallenge" method implementation.

Another support feature is implemented by the "requestChallenge" method, which allows the CSS to choose a list of data block identifiers and to request auditing challenges to the Client. Only the CSS can carry out this method, and only while the acceptance of the respective CFSMC

```
function verifyChallenge(uint _id, bytes32
    _noResponse) external {
    require(msg.sender == clientAddress || msg.
        sender == integrityCheckServiceAddress, "
        Only the CFSMC owner Client or the
        Integrity Check Service can verify
        submitted challenges!");
    require(challenges.contains(_id), "Challenge
        id was not found!");
    require(challenges.data[_id].responseDatetime
        == 0, "This challenge id was already
        replied!");
    challenges.insertResponse(_id, _noResponse,
        false);
    uint cycle = computeCycle(_id);
    if (!cyclesFinalized[cycle]) {
        SSTMContract.decrementTrustValue(
            storageServiceAddress);
        cyclesFinalized[cycle] = true;
    }
}
```

LISTING 8. **Implementation of "verifyChallenge" method**

instance is not registered at the SSTMC instance. When the CSS executes the "requestChallenge" method, it passes an array of integers containing the data block identifiers of the requested challenges as a parameter. When processing the transaction, the CFSMC stores each challenge requested as a new element in the "requestedChallenges" attribute, which is a mapping managed through an object of the "RequestedMap" class, whose key is the data block identifier and whose value is an object of the "RequestedData" class with its 'requested" attribute set to "true". The execution of the "getRequestedChallenges" method (Figure 4) obtains the registered challenge requests, which returns a vector of integers with the list of the data block identifiers of the challenges required by the CSS. Whenever a new challenge is submitted by the Client to the CFSMC instance using the "submitChallenge" method (Figure 4), the CFSMC instance excludes the challenge request with the same data block identifier.

Finalizing the description of the implementation of the SC CFSMC, we present the "getClientAddress", "getStorageServiceAddress", "getFileId", "getChunkSize", and "getStorageLimitDate" (Figure 4) methods. These methods have the following common characteristics: they do not generate transactions in the BN, as their execution does not alter any information in the CFSMC instance, and; they don't receive parameters. The reason for this is that they only return the contents stored respectively in the attributes "clientAddress", "storageServiceAddress", "fileId", "chunkSize", "storageLimitDate". The methods "getTrustManagementContract" and "isAccepted" (Figure 4) have the same characteristics mentioned above. However, the "getTrustManagementContract" method returns only the access address at the BN to the object of the SSTMC instance stored in the "trustManagementContract" attribute. Finally, the "isAccepted" method returns the result of the execution of the "isAuthorized" method of the SSTMC instance stored in the "trustManagementContract" attribute, which receives the CSS credential address stored in the "storageServiceAddress" attribute as a parameter.

C. IMPLEMENTATION OF THE CLIENT, CSS, AND ICS APPLICATIONS

We implemented the Client, CSS, and ICS applications using JAVA. To implement communication between the applications and the BN, we adopted the Web3J [26], a library for application integration with the Ethereum BN [22]. In addition, to facilitate interaction with the BN, we used the Web3J library to generate the "*Java Wrappers*" [26] of the SCs SSTMC and CFSMC, both implemented using the Solidity [20] language. These "*wrappers*" allow interaction with the SCs through Java objects, eliminating the complexity of communication between languages.

For the implementation of the Client, CSS, and ICS applications, we adopted the use of threads to enable the parallel execution of the various functionalities, such as file uploads, file downloads, and the monitoring of results. We also implemented routines to identify non-executed transactions and to perform their automatic re-submission. To monitor the BN, we adopted scheduling mechanisms (java.util.concurrent.ScheduledExecutorService and EJB @Schedules), which periodically trigger the methods responsible for interacting with the SCs to collect information, identify issues, process them, and record the results in the BN. The implementation of the "detectStorageSevicesToCheck" method of the class "CheckFilesHandler" of the ICS application, responsible for triggering the verification process of the files stored in the different CSS and which uses the annotation EJB "@Schedules", is presented in Listing 9.

```
@Schedules({@Schedule(hour="00", minute="30", second="30")})
public void detectStorageSevicesToCheck() {
    List<StorageServiceCheckEvent> storageServiceCheckEvents =
        storageContractDAO.findStorageServiceCheckEvents();
    Iterator<StorageServiceCheckEvent> it = storageServiceCheckEvents.
        iterator();
    while (it.hasNext()) {
        StorageServiceCheckEvent sscEvent = it.next();
        storageServiceCheckEvent.fire(sscEvent);
    }
}
```

LISTING 9. **Implementation of "detectStorageSevicesToCheck" method.**

1) IMPLEMENTATION OF THE APPLICATION FOR THE *Client*

The Client application is a desktop application whose main features are the following: it allows the Client to select the file to be stored in the cloud, the trust management contract, the ICS to check its integrity, and the CSS to which the file copies will be submitted; it encrypts the file; calculates and generates the challenges, the "challenge passwords", and the "verification hashes"; it submits a CFSMC instance with the file verification information for each chosen CSS to the BN; it submits file copies to the selected CSS; it submits information to generate challenges to the ICS; and, it manages files stored in the cloud. Figure 5 shows the class diagram of the application for the Client.

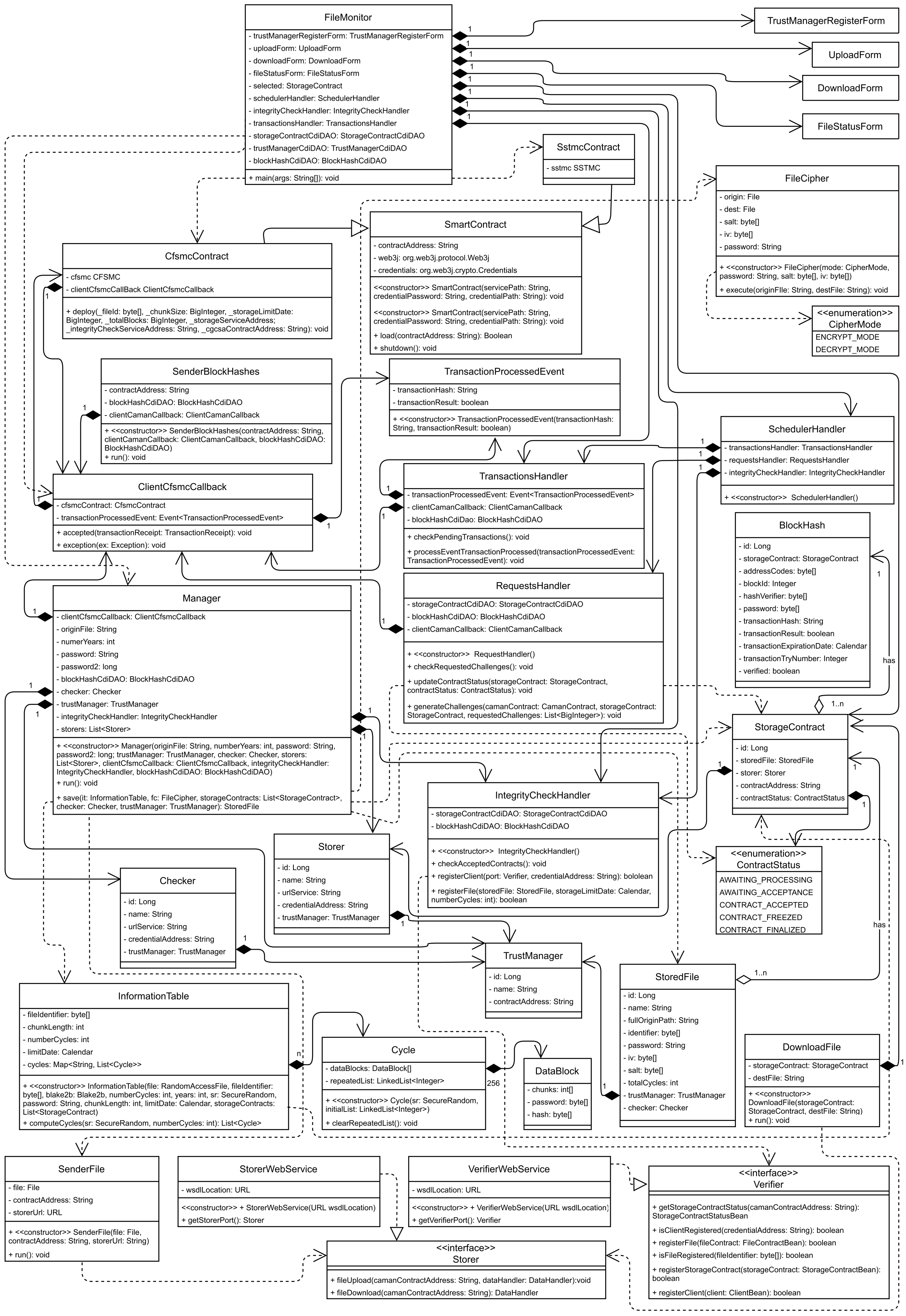
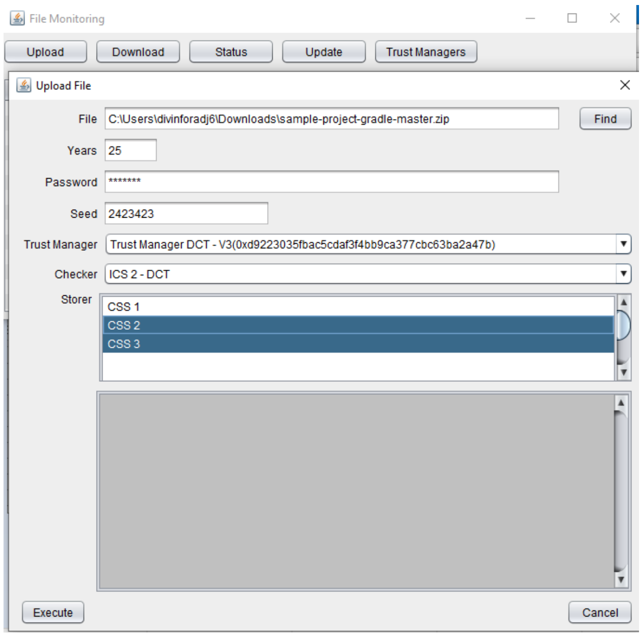

FileMonitor
- trustManagerRegisterForm: TrustManagerRegisterForm
- uploadForm: UploadForm
- downloadForm: DownloadForm
- fileStatusForm: FileStatusForm
- selected: StorageContract
- schedulerHandler: SchedulerHandler
- integrityCheckHandler: IntegrityCheckHandler
- transactionsHandler: TransactionsHandler
- storageContractCdiDAO: StorageContractCdiDAO
- trustManagerCdiDAO: TrustManagerCdiDAO
- blockHashCdiDAO: BlockHashCdiDAO
+ main(args: String[]): void
TrustManagerRegisterForm
UploadForm
DownloadForm
FileStatusForm
SstmcContract
- sstmc SSTMC
FileCipher
- origin: File
- dest: File
- salt: byte[]
- iv: byte[]
- password: String
+ <<constructor>> FileCipher(mode: CipherMode, password: String, salt: byte[], iv: byte[])
+ execute(originFIle: String, destFile: String): void
<<enumeration>> CipherMode
ENCRYPT_MODE
DECRYPT_MODE
SmartContract
- contractAddress: String
- web3j: org.web3j.protocol.Web3j
- credentials: org.web3j.crypto.Credentials
<<constructor>> SmartContract(servicePath: String, credentialPassword: String, credentialPath: String): void
<<constructor>> SmartContract(servicePath: String, credentialPassword: String, credentialPath: String): void
+ load(contractAddress: String): Boolean
+ shutdown(): void
CfsmcContract
- cfsmc CFSMC
- clientCfsmcCallBack ClientCfsmcCallback
+ deploy(_fileId: byte[], _chunkSize: BigInteger, _storageLimitDate: BigInteger, _totalBlocks: BigInteger, _storageServiceAddress; _integrityCheckServiceAddress: String, _cgcsaContractAddress: String): void
SenderBlockHashes
- contractAddress: String
- blockHashCdiDAO: BlockHashCdiDAO
- clientCamanCallback: ClientCamanCallback
+ <<constructor>> SenderBlockHashes(contractAddress: String, clientCamanCallback: ClientCamanCallback, blockHashCdiDAO: BlockHashCdiDAO)
+ run(): void
TransactionProcessedEvent
- transactionHash: String
- transactionResult: boolean
+ <<constructor>> TransactionProcessedEvent(transactionHash: String, transactionResult: boolean):
SchedulerHandler
- transactionsHandler: TransactionsHandler
- requestsHandler: RequestsHandler
- integrityCheckHandler: IntegrityCheckHandler
+ <<constructor>> SchedulerHandler()
ClientCfsmcCallback
- cfsmcContract: CfsmcContract
- transactionProcessedEvent: Event<TransactionProcessedEvent>
+ accepted(transactionReceipt: TransactionReceipt): void
+ exception(ex: Exception): void
TransactionsHandler
- transactionProcessedEvent: Event<TransactionProcessedEvent>
- clientCamanCallback: ClientCamanCallback
- blockHashCdiDao: BlockHashCdiDAO
+ checkPendingTransactions(): void
+ processEventTransactionProcessed(transactionProcessedEvent: TransactionProcessedEvent): void
BlockHash
- id: Long
- storageContract: StorageContract
- addressCodes: byte[]
- blockId: Integer
- hashVerifier: byte[]
- password: byte[]
- transactionHash: String
- transactionResult: boolean
- transactionExpirationDate: Calendar
- transactionTryNumber: Integer
- verified: boolean
has
1..n
Manager
- clientCfsmcCallback: ClientCfsmcCallback
- originFile: String
- numerYears: int
- password: String
- password2: long
- blockHashCdiDAO: BlockHashCdiDAO
- checker: Checker
- trustManager: TrustManager
- integrityCheckHandler: IntegrityCheckHandler
- storers: List<Storer>
+ <<constructor>> Manager(originFile: String, numberYears: int, password: String, password2: long; trustManager: TrustManager, checker: Checker, storers: List<Storer>, clientCfsmcCallback: ClientCfsmcCallback, integrityCheckHandler: IntegrityCheckHandler, blockHashCdiDAO: BlockHashCdiDAO)
+ run(): void
+ save(it: InformationTable, fc: FileCipher, storageContracts: List<StorageContract>, checker: Checker, trustManager: TrustManager): StoredFile
RequestsHandler
- storageContractCdiDAO: StorageContractCdiDAO
- blockHashCdiDAO: BlockHashCdiDAO
- clientCamanCallback: ClientCamanCallback
+ <<constructor>> RequestHandler()
+ checkRequestedChallenges(): void
+ updateContractStatus(storageContract: StorageContract, contractStatus: ContractStatus): void
+ generateChallenges(camanContract: CamanContract, storageContract: StorageContract, requestedChallenges: List<BigInteger>): void
StorageContract
- id: Long
- storedFile: StoredFile
- storer: Storer
- contractAddress: String
- contractStatus: ContractStatus
IntegrityCheckHandler
- storageContractCdiDAO: StorageContractCdiDAO
- blockHashCdiDAO: BlockHashCdiDAO
+ <<constructor>> IntegrityCheckHandler()
+ checkAcceptedContracts(): void
+ registerClient(port: Verifier, credentialAddress: String): bololean
+ registerFile(storedFile: StoredFile, storageLimitDate: Calendar, numberCycles: int): boolean
<<enumeration>> ContractStatus
AWAITING_PROCESSING
AWAITING_ACCEPTANCE
CONTRACT_ACCEPTED
CONTRACT_FREEZED
CONTRACT_FINALIZED
Storer
- id: Long
- name: String
- urlService: String
- credentialAddress: String
- trustManager: TrustManager
Checker
- id: Long
- name: String
- urlService: String
- credentialAddress: String
- trustManager: TrustManager
TrustManager
- id: Long
- name: String
- contractAddress: String
StoredFile
- id: Long
- name: String
- fullOriginPath: String
- identifier: byte[]
- password: String
- iv: byte[]
- salt: byte[]
- totalCycles: int
- trustManager: TrustManager
- checker: Checker
InformationTable
- fileIdentifier: byte[]
- chunkLength: int
- numberCycles: int
- limitDate: Calendar
- cycles: Map<String, List<Cycle>>
+ <<constructor>> InformationTable(file: RandomAccessFile, fileIdentifier: byte[], blake2b: Blake2b, numberCycles: int, years: int, sr: SecureRandom, password: String, chunkLength: int, limitDate: Calendar, storageContracts: List<StorageContract)
+ computeCycles(sr: SecureRandom, numberCycles: int): List<Cycle>
n
Cycle
- dataBlocks: DataBlock[]
- repeatedList: LinkedList<Integer>
+ <<constructor>> Cycle(sr: SecureRandom, initialList: LinkedList<Integer>)
+ clearRepeatedList(): void
256
DataBlock
- chunks: int[]
- password: byte[]
- hash: byte[]
DownloadFile
- storageContract: StorageContract
- destFile: String
+ <<constructor>> DownloadFile(storageContract: StorageContract, destFile: String)
+ run(): void
SenderFile
- file: File
- contractAddress: String
- storerUrl: URL
+ <<constructor>> SenderFile(file: File, contractAddress: String, storerUrl: String)
+ run(): void
StorerWebService
- wsdlLocation: URL
<<constructor>> + StorerWebService(URL wsdlLocation)
+ getStorerPort(): Storer
VerifierWebService
- wsdlLocation: URL
<<constructor>> + VerifierWebService(URL wsdlLocation)
+ getVerifierPort(): Verifier
<<interface>> Verifier
+ getStorageContractStatus(camanContractAddress: String): StorageContractStatusBean
+ isClientRegistered(credentialAddress: String): boolean
+ registerFile(fileContract: FileContractBean): boolean
+ isFileRegistered(fileIdentifier: byte[]): boolean
+ registerStorageContract(storageContract: StorageContractBean): boolean
+ registerClient(client: ClientBean): boolean
<<interface>> Storer
+ fileUpload(camanContractAddress: String, dataHandler: DataHandler):void
+ fileDownload(camanContractAddress: String): DataHandler

FIGURE 5. **Application class diagram for the client.**

For better visualization, we have purposely suppressed the methods "*get*" and "*set*" of the attributes, the methods and attributes related to the forms, the DAO classes (*Data Access Object*) that implement the data access and persistence in the DBMS, and the private methods of all classes from the diagram shown in Figure 5. The classes "SSTMC" and "CFSMC" are classes generated through the library *Web3J* that implement the "*wrappers*" respectively for all methods implemented in the SCs SSTMC and CFSMC. To avoid redundancy and to reduce the complexity of the diagram, we do not represent the implementation of these classes in Figure 5.

The "FileMonitor" class is responsible for initializing the application and for the graphical interface through which it makes its features available to the user. These features are the submission of files to the cloud through the "*Upload*" option; the recovery of the file stored in the cloud using the *"Download"* option; the consultation of the file monitoring status using the *"Status"* option; and the registration of available trust management contracts through the "*Trust Managers*" option. Figure 6 shows the Client application interface during the process of submitting a file to the cloud.


File Monitoring

Upload | Download | Status | Update | Trust Managers

Upload File

File: C:\Users\divinforadj6\Downloads\sample-project-gradle-master.zip [Find]

Years: 25

Password: *******

Seed: 2423423

Trust Manager: Trust Manager DCT - V3(0xd9223035fbac5cdaf3f4bb9ca377cbc63ba2a47b)

Checker: ICS 2 - DCT

Storer: CSS 1, [✓ CSS 2], [✓ CSS 3]

Execute | Cancel

FIGURE 6. Client application interface.

During the Client application initialization, it automatically instantiates the "SchedulerHandler" class as an independent thread. This class schedules the execution of the functionalities of the classes "RequestsHandler", "TransactionsHandler", and "IntegrityCheckHandler" in the background. These classes are responsible respectively for: i) monitoring the audit challenge requests from the CSS, generating the requested challenges and submitting them using the "submitChallenge" method of the CFSMC instance of

```
public SchedulerHandler() {
  Runnable checkPendingTransactions = new Runnable(){
    public void run() {
       transactionsHandler.checkPendingTransactions();
  }};
  Runnable checkRequestedChallenges = new Runnable() {
     public void run() {
        requestsHandler.checkRequestedChallenges();
   }};
   Runnable checkAcceptedContracts = new Runnable(){
     public void run() {
        integrityCheckHandler.checkAcceptedContracts();
   }};
   ScheduledFuture<?> cptHandle = scheduler.scheduleAtFixedRate(
        checkPendingTransactions, 2, 30, MINUTES);
   ScheduledFuture<?> crcHandle = scheduler.scheduleAtFixedRate(
        checkRequestedChallenges, 1, 15, MINUTES);
   ScheduledFuture<?> cacHandle = scheduler.scheduleAtFixedRate(
        checkAcceptedContracts, 3, 5, MINUTES);
}
```

LISTING 10. Implementation of the constructor of the class SchedulerHandler.

origin of the request; ii) monitoring the transaction processing in the BN; and, iii) executing the registration of the integrity verification contract with the ICS when the Client identifies the acceptance of the storage contract by the CSS, as well as the monitoring of the validity of the existing integrity verification contracts, with the respective renewal or contracting of a new ICS, and, in both situations, submitting the information to generate challenges for the contracted period. Listing 10 presents the implementation of the constructor method of the "SchedulerHandler" class.

Below, we present a brief description of the functions of the main classes implemented in the Client application:

- the "FileMonitor" class loads the auxiliary classes and provides the means of accessing all the Client application functionalities;
- the "UploadForm" class presents the form for the selecting of the file and the other parameters (storage time, the password for encryption, one or more CSSs, an ICS, and the SSTMC instance to manage the trust), validates these parameters, and initializes a thread with a "Manager" class instance using the chosen parameters;
- the "Manager" class performs the chosen file encryption, calculates the number of verification cycles to be generated, creates a CFSMC instance for each selected CSS and submits it to the BN using the CfsmcContract class, initializes a thread with a "SenderFile" class instance to send an encrypted file copy to each selected CSS, loads an "InformationTable" class instance that generates information to verify integrity, asks for the file registration in the selected ICS using the "registerFile" method of the "IntegrityCheckHandler" class, stores the information for checking and retrieving each file copy sent for storage in the CSS in the local database, and registers at the BN the information to check the file by instantiating a thread of the "SenderBlockHash" class for each CSS;

- the "SenderFile" class sends the encrypted file to the CSS through the "StorerWebService" class, using a new instance (thread) for each chosen CSS;
- the "StorerWebService" class performs the communication between the Client application and the CSS application, implementing the interfaces that execute the calls to the web service methods provided by the CSS;
- the "InformationTable" class creates the "verification hashes" and "challenge passwords" for each of the chosen CSS through the "Cycle" class;
- the "Cycle" class draws each of the 16 fraction addresses that form the FAS that will give rise to the challenges, and stores each FAS in the "chunks" attribute of an object of the "DataBlock" class, repeating the process until it forms a cycle containing 256 FAS and all 4096 file fraction addresses without repetition;
- the "SenderBlockHash" class registers the information to verify integrity (verification hash and challenge password) of the file in its CFSMC instance stored in the BN, using a new class instance (thread) for each chosen CSS;
- the "IntegrityCheckHandler" class performs the hiring or renewal of the integrity check service with the selected ICS for each file copy, as well as the monitoring of these services, sending the necessary information to generate the challenges through the "VerifierWebService" class;
- the "VerifierWebService" class performs the communication between the Client application and the ICS application, implementing the interfaces that execute the calls of the web service methods provided by the ICS;
- the "RequestsHandler" class performs the monitoring of the audit challenge requests generated by the CSS and recorded in the CFSMC instance of each file submitted for storage with the respective generation of challenges for each registered request, which are processes performed through the "CfsmcContract" class;
- the "TransactionsHandler" class monitors the transactions recently submitted to the BN waiting for their processing through the "CfsmcContract" class and, for each successfully processed transaction, receives its hash and stores it in the local database;
- the "DownloadForm" class presents the form for the selecting of the destination of the file copy that will be retrieved from the CSS, and initializes a thread with an instance of the "DownloadFile" class;
- the "DownloadFile" class carries out the download of a file stored in a CSS through the "StorerWebService" class.

2) APPLICATION IMPLEMENTATION FOR CLOUD STORAGE SERVICES

We developed the CSS application as a web service, whose main functionalities are the following: receive the file submitted by the Client for storage; audit the information stored in the BN through a CFSMC instance; confirm its compatibility with the received file; answer the integrity verification challenges registered at the BN through the respective CFSMC instance; and, allow exclusive access to the file content for the Client who submitted it. Figure 7 presents the class diagram of the application developed for the CSS.

In the diagram shown in Figure 7, we purposely suppress the "*get*" and "*set*" methods of the attributes and the private methods of the class. The class diagram of the application developed for the CSS presents the "CfsmcContract" and "SstmcContract" classes through which the application interacts with the SC instances stored in the BN using the "*wrappers*" implemented in the SSTMC and CFSMC classes described in Section IV-C1.

The CSS application implements only two features that permanently remain available to be executed as web services from the Client application. The "fileUpload" and "fileDownload" methods implement these features. The method "fileUpload" allows the Client application to submit a file for storage in the CSS, informing the access address of the CFSMC instance registered at the BN for this file as a parameter. The method "fileDownload" allows the Client application to request a copy of the file stored in the CSS, informing only the CFSMC instance address linked to it.

The classes "StorerHandler", "CheckerHandler" and "AnswerHandler" are instantiated as independent threads at the CSS application start, and are responsible for performing the interactions between the CSS and the BN. The "StorerHandler" class is responsible for auditing the verification information stored in the CFSMC instances linked to files received for storage. The "CheckerHandler" class implements the monitoring of the CFSMC instances related to the stored files, recovering the stored challenges and triggering the response generation processes. Finally, the class "AnswerHandler" is responsible for monitoring the completion of the response generation processes to the challenges received, and registers these responses in the CFSMC instance of origin of the challenge.

Aiming to obtain better performance, we have parallelized the execution of several processes. For this, all requests for functions that require interaction with the BN are carried out through event registration. The responsible classes monitor and asynchronously execute each event. Any problem or delay in a process execution does not interfere with the operation of the other functionalities of the application.

The methods "detectAwaitingAuditStart" and "detectAwaitingAuditChallenge", both of the class "StorerHandler", are self-executed, every minute, through scheduling using the annotation "@Schedule". The "detectAwaitingAuditStart" method searches for records of files received from the Client in the local database, for which the audit process of the respective CFSMC instance has not yet started, starting with the "RequestChallengeEvent" event record, which is

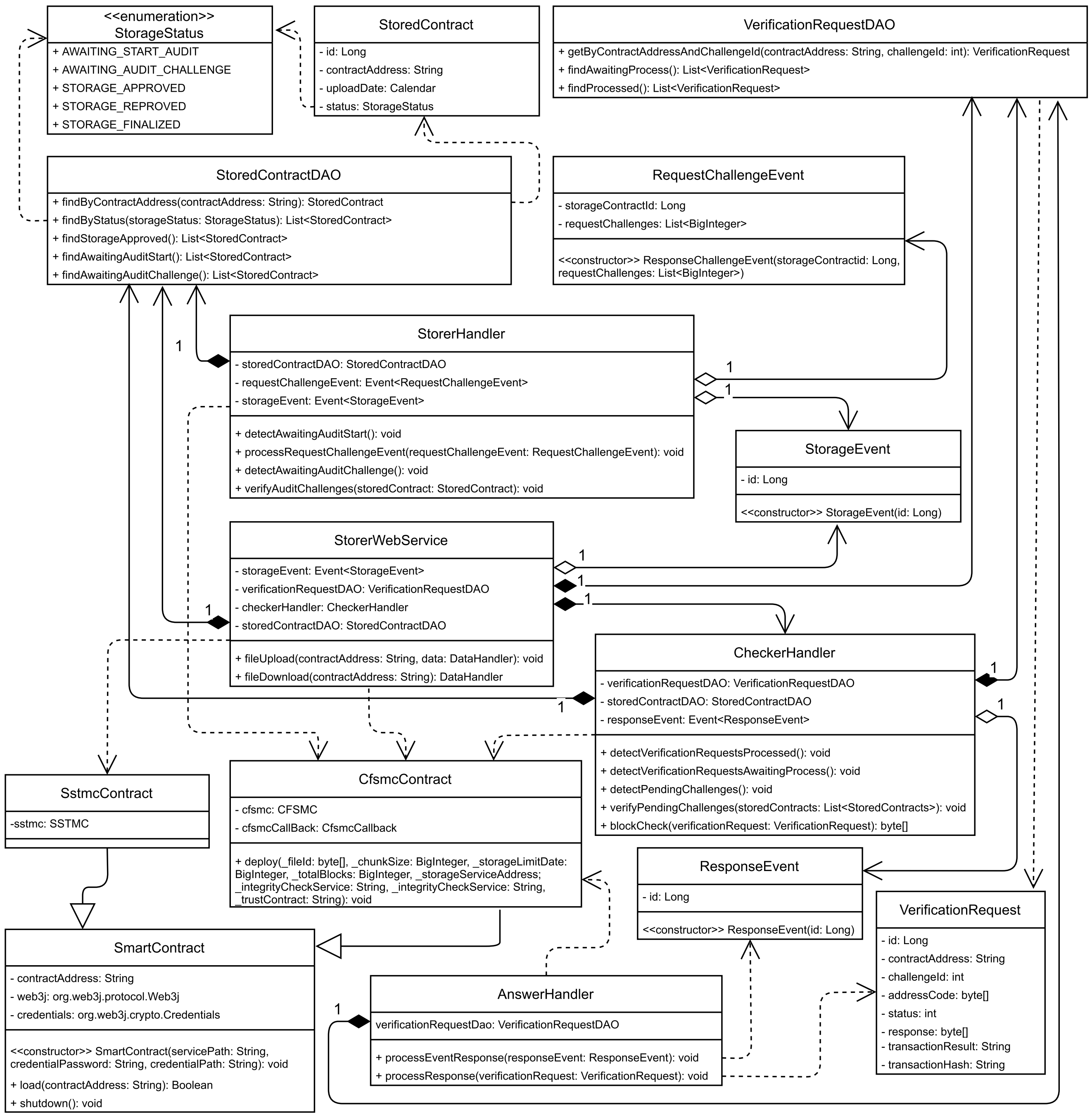

<<enumeration>>
StorageStatus
+ AWAITING_START_AUDIT
+ AWAITING_AUDIT_CHALLENGE
+ STORAGE_APPROVED
+ STORAGE_REPROVED
+ STORAGE_FINALIZED
StoredContract
- id: Long
- contractAddress: String
- uploadDate: Calendar
- status: StorageStatus
VerificationRequestDAO
+ getByContractAddressAndChallengeId(contractAddress: String, challengeId: int): VerificationRequest
+ findAwaitingProcess(): List<VerificationRequest>
+ findProcessed(): List<VerificationRequest>
StoredContractDAO
+ findByContractAddress(contractAddress: String): StoredContract
+ findByStatus(storageStatus: StorageStatus): List<StoredContract>
+ findStorageApproved(): List<StoredContract>
+ findAwaitingAuditStart(): List<StoredContract>
+ findAwaitingAuditChallenge(): List<StoredContract>
RequestChallengeEvent
- storageContractId: Long
- requestChallenges: List<BigInteger>
<<constructor>> ResponseChallengeEvent(storageContractid: Long, requestChallenges: List<BigInteger>)
StorerHandler
- storedContractDAO: StoredContractDAO
- requestChallengeEvent: Event<RequestChallengeEvent>
- storageEvent: Event<StorageEvent>
+ detectAwaitingAuditStart(): void
+ processRequestChallengeEvent(requestChallengeEvent: RequestChallengeEvent): void
+ detectAwaitingAuditChallenge(): void
+ verifyAuditChallenges(storedContract: StoredContract): void
StorageEvent
- id: Long
<<constructor>> StorageEvent(id: Long)
StorerWebService
- storageEvent: Event<StorageEvent>
- verificationRequestDAO: VerificationRequestDAO
- checkerHandler: CheckerHandler
- storedContractDAO: StoredContractDAO
+ fileUpload(contractAddress: String, data: DataHandler): void
+ fileDownload(contractAddress: String): DataHandler
CheckerHandler
- verificationRequestDAO: VerificationRequestDAO
- storedContractDAO: StoredContractDAO
- responseEvent: Event<ResponseEvent>
+ detectVerificationRequestsProcessed(): void
+ detectVerificationRequestsAwaitingProcess(): void
+ detectPendingChallenges(): void
+ verifyPendingChallenges(storedContracts: List<StoredContracts>): void
+ blockCheck(verificationRequest: VerificationRequest): byte[]
SstmcContract
-sstmc: SSTMC
CfsmcContract
- cfsmc: CFSMC
- cfsmcCallBack: CfsmcCallback
+ deploy(_fileId: byte[], _chunkSize: BigInteger, _storageLimitDate: BigInteger, _totalBlocks: BigInteger, _storageServiceAddress; _integrityCheckService: String, _integrityCheckService: String, _trustContract: String): void
ResponseEvent
- id: Long
<<constructor>> ResponseEvent(id: Long)
VerificationRequest
- id: Long
- contractAddress: String
- challengeId: int
- addressCode: byte[]
- status: int
- response: byte[]
- transactionResult: String
- transactionHash: String
SmartContract
- contractAddress: String
- web3j: org.web3j.protocol.Web3j
- credentials: org.web3j.crypto.Credentials
<<constructor>> SmartContract(servicePath: String, credentialPassword: String, credentialPath: String): void
+ load(contractAddress: String): Boolean
+ shutdown(): void
AnswerHandler
verificationRequestDao: VerificationRequestDAO
+ processEventResponse(responseEvent: ResponseEvent): void
+ processResponse(verificationRequest: VerificationRequest): void
1
1
1
1
1
1
1
1
1
1
1

FIGURE 7. Application Class Diagram for the CSS.

monitored by the "processRequestChallengeEvent" method of the same class.

The "processRequestChallengeEvent" method is asynchronously executed each time a "RequestChallengeEvent" event is registered. During its execution, challenges are randomly drawn to audit the CFSMC instance linked to the file received from the Client. Then, the CSS requests these challenges to the Client by submitting them to the CFSMC instance using the "requestChallenge" method. Listing 11 shows the source code with the "processRequestChallengeEvent" method implementation.

The "detectAwaitingAuditChallenge" method processes the received file records that await the processing of the requested challenges for auditing purposes, checking if the Clients have submitted the challenges before the request expires. Also, it verifies whether the responses generated by

```
@Asynchronous
public void processRequestChallengeEvent(@Observes
      RequestChallengeEvent requestChallengeEvent) {
  StoredContract storedContract = storedContractDao.getById(
      requestChallengeEvent.getStorageContractId());
  if (storedContract != null
     && storedContract.getStatus().equals(StorageStatus.
      AWAITING_START_AUDIT){
     CfsmcContract cfsmcContract = new CfsmcContract();
     cfsmcContract.load(storedContract.getContractAddress());
     cfsmcContract.requestChallenge(requestChallengeEvent.
      getRequestChallenges());
     updateStoredContractStatus(storedContract, StorageStatus.
      AWAITING_AUDIT_CHALLENGE);
  }
}
```

LISTING 11. Implementation of "processRequestChallengeEvent" method.

the CSS have been considered valid by the CFSMC instance. If all responses have been successful validated, the CSS registers the file storage acceptance in the local database and makes it public by executing the SSTMC's "authorize contract" method.

As with the "StorerHandler" class, the application server automatically executes the methods "detectPendingChallenges", "detectVerificationRequestsAwaitingProcess", and "detectVerificationRequestsProcessed" as independent threads using the annotation "@Schedule" for this scheduling. These methods are responsible respectively for detecting the pending challenges in the CFSMC instance linked to the files which were stored through the CFSMC "getPendingChallenges" method, registering them in the local database; for detecting pending challenges in the local database, processing them and generating the respective responses, and; for detecting challenges already processed in the local database and registering a "ResponseEvent" event.

The events of the type "ResponseEvent" are monitored and processed using the "processEventResponse" method of the "AnswerHandler" class, which is automatically executed by the application server whenever an instance of that event is registered. The "processEventResponse" method is responsible for submitting the challenge responses employing the "replyChallenge" method of the CFSMC instance of origin of the challenge.

3) APPLICATION IMPLEMENTATION FOR INTEGRITY CHECK SERVICES

We developed the application for the ICS as a web service, whose main duties are the following: to periodically receive from the Client a set of file information for generating challenges; to daily generate challenges for the CSS according to the trust level assigned by the BN; to regularly monitor pending challenges triggering the BN to penalize the CSS when the established maximum deadline for each submitted challenge to receive an answer is not met. Figure 8 presents the class diagram of the application developed for the ICS.

To enable the Client application to contract an ICS, the ICS application implements the "registerClient", "registerFile" and "registerStorageContract" methods as web services. For this, a class named "VerifierWebService" gathers all these methods. In addition, this class implements the "isClientRegistered", "isFileRegistered", and "getStorageContractStatus" information query methods. The Client application uses the "isClientRegistered" method to check for the existence of the record in the ICS. Before contracting a new service, it uses the "isFileRegistered" method to check whether the file stored in a CSS was previously registered at the ICS. Finally, the "getStorageContractStatus" method allows the Client application to consult the ICS for both the carried out check results and the status of the integrity verification contract linked to a CFSMC instance registered at the BN.

Through the "registerClient" method, the ICS application allows the Client to self-register in the ICS, becoming able to contract it. As the protocol allows submitting copies of the same file for storage in different CSSs and the contract acceptance process by the CSS occurs asynchronously, before the Client application sends the file content to one or more CSSs, it registers the file information in the chosen ICS using the "regiterFile" method. As soon as the CSS accepts the submitted file and its storage contract, the Client application registers that contract in the chosen ICS through the "registerStorageContract" method, using the file information set for the generation of challenges during the pre-defined ICS contracting period as a parameter. The Client application either uses this method to renew the existing integrity verification contract or contracts a new ICS, both of which submit a new set of file information for generating challenges to the hired ICS.

The "CheckFilesHandler", "CheckRequestsHandler", and "RequestService" classes implement the process to generate and monitor challenges, the main responsibilities of the ICS application. When the application server starts the ICS application, it automatically instantiates these classes as independent threads. Through the annotation "@Scheduler", the ICS application schedules the daily execution of the "detectStorageServicesToCheck" method of the "CheckFilesHandler" class, and the "checkRequests" method of the "CheckRequestsHandler" class. Simultaneously, the ICS application starts the monitoring of the "StorageServiceCheckEvent" event by the "checkStorageService" method of the "RequestService" class.

The first task of the "detectStorageServicesToCheck" method is selecting, in the local database, all CSS that store one or more files linked to active integrity verification contracts. For each chosen CSS, the referred method registers an event of the type "StorageServiceCheckEvent". This event is individually captured and asynchronously executed by the "checkStorageService" method of the "RequestService" class.

For each execution of the "checkStorageService" method, it instantiates an object of the "SstmcContract" class, which

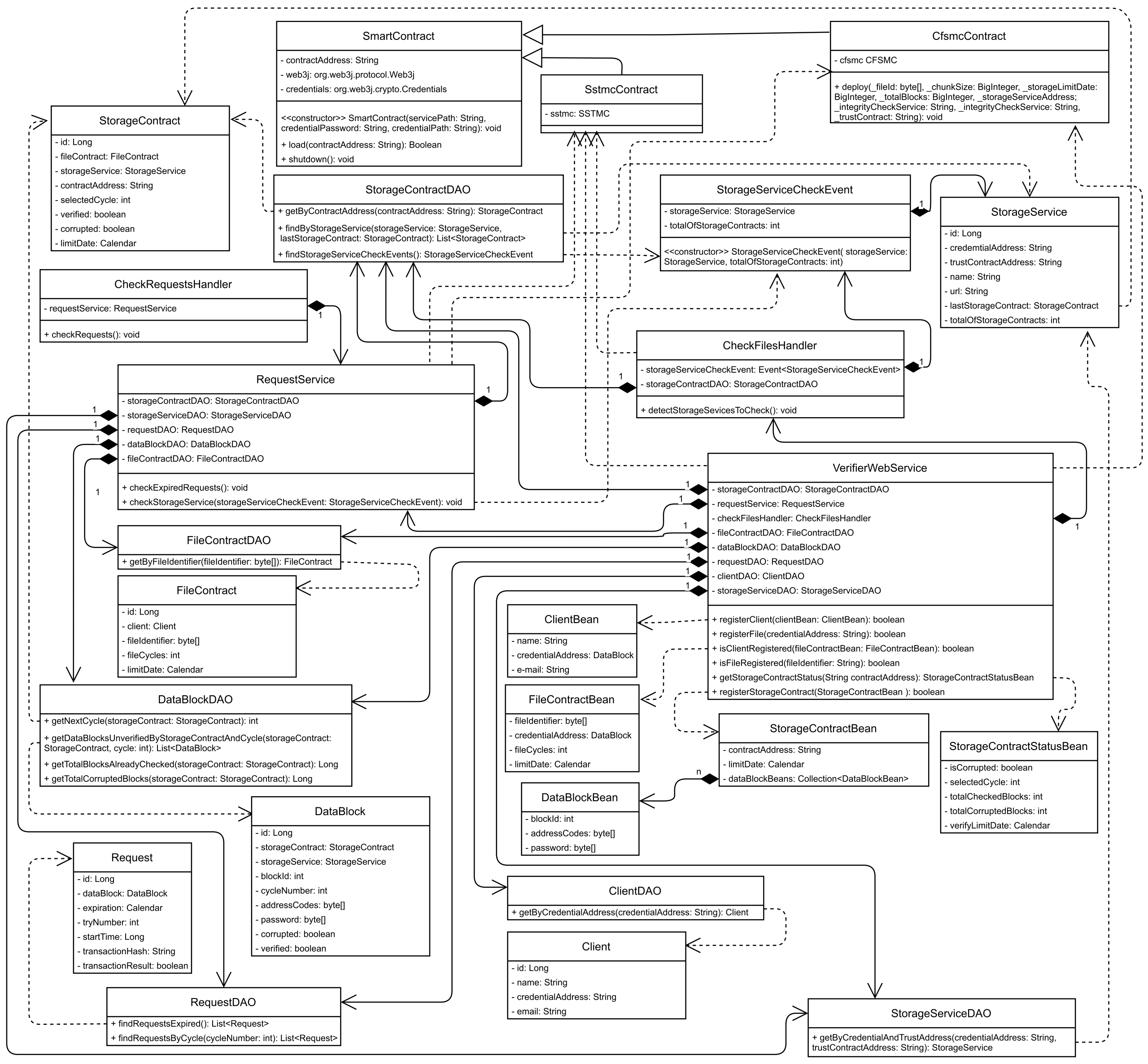

SmartContract
- contractAddress: String
- web3j: org.web3j.protocol.Web3j
- credentials: org.web3j.crypto.Credentials
<<constructor>> SmartContract(servicePath: String, credentialPassword: String, credentialPath: String): void
+ load(contractAddress: String): Boolean
+ shutdown(): void
CfsmcContract
- cfsmc CFSMC
+ deploy(_fileId: byte[], _chunkSize: BigInteger, _storageLimitDate: BigInteger, _totalBlocks: BigInteger, _storageServiceAddress; _integrityCheckService: String, _integrityCheckService: String, _trustContract: String): void
SstmcContract
- sstmc: SSTMC
StorageContract
- id: Long
- fileContract: FileContract
- storageService: StorageService
- contractAddress: String
- selectedCycle: int
- verified: boolean
- corrupted: boolean
- limitDate: Calendar
StorageContractDAO
+ getByContractAddress(contractAddress: String): StorageContract
+ findByStorageService(storageService: StorageService, lastStorageContract: StorageContract): List<StorageContract>
+ findStorageServiceCheckEvents(): StorageServiceCheckEvent
StorageServiceCheckEvent
- storageService: StorageService
- totalOfStorageContracts: int
<<constructor>> StorageServiceCheckEvent(storageService: StorageService, totalOfStorageContracts: int)
StorageService
- id: Long
- credemtialAddress: String
- trustContractAddress: String
- name: String
- url: String
- lastStorageContract: StorageContract
- totalOfStorageContracts: int
CheckRequestsHandler
- requestService: RequestService
+ checkRequests(): void
CheckFilesHandler
- storageServiceCheckEvent: Event<StorageServiceCheckEvent>
- storageContractDAO: StorageContractDAO
+ detectStorageSevicesToCheck(): void
RequestService
- storageContractDAO: StorageContractDAO
- storageServiceDAO: StorageServiceDAO
- requestDAO: RequestDAO
- dataBlockDAO: DataBlockDAO
- fileContractDAO: FileContractDAO
+ checkExpiredRequests(): void
+ checkStorageService(storageServiceCheckEvent: StorageServiceCheckEvent): void
VerifierWebService
- storageContractDAO: StorageContractDAO
- requestService: RequestService
- checkFilesHandler: CheckFilesHandler
- fileContractDAO: FileContractDAO
- dataBlockDAO: DataBlockDAO
- requestDAO: RequestDAO
- clientDAO: ClientDAO
- storageServiceDAO: StorageServiceDAO
+ registerClient(clientBean: ClientBean): boolean
+ registerFile(credentialAddress: String): boolean
+ isClientRegistered(fileContractBean: FileContractBean): boolean
+ isFileRegistered(fileIdentifier: String): boolean
+ getStorageContractStatus(String contractAddress): StorageContractStatusBean
+ registerStorageContract(StorageContractBean): boolean
FileContractDAO
+ getByFileIdentifier(fileIdentifier: byte[]): FileContract
FileContract
- id: Long
- client: Client
- fileIdentifier: byte[]
- fileCycles: int
- limitDate: Calendar
ClientBean
- name: String
- credentialAddress: DataBlock
- e-mail: String
FileContractBean
- fileIdentifier: byte[]
- credentialAddress: DataBlock
- fileCycles: int
- limitDate: Calendar
StorageContractBean
- contractAddress: String
- limitDate: Calendar
- dataBlockBeans: Collection<DataBlockBean>
StorageContractStatusBean
- isCorrupted: boolean
- selectedCycle: int
- totalCheckedBlocks: int
- totalCorruptedBlocks: int
- verifyLimitDate: Calendar
DataBlockBean
- blockId: int
- addressCodes: byte[]
- password: byte[]
DataBlockDAO
+ getNextCycle(storageContract: StorageContract): int
+ getDataBlocksUnverifiedByStorageContractAndCycle(storageContract: StorageContract, cycle: int): List<DataBlock>
+ getTotalBlocksAlreadyChecked(storageContract: StorageContract): Long
+ getTotalCorruptedBlocks(storageContract: StorageContract): Long
DataBlock
- id: Long
- storageContract: StorageContract
- storageService: StorageService
- blockId: int
- cycleNumber: int
- addressCodes: byte[]
- password: byte[]
- corrupted: boolean
- verified: boolean
Request
- id: Long
- dataBlock: DataBlock
- expiration: Calendar
- tryNumber: int
- startTime: Long
- transactionHash: String
- transactionResult: boolean
RequestDAO
+ findRequestsExpired(): List<Request>
+ findRequestsByCycle(cycleNumber: int): List<Request>
ClientDAO
+ getByCredentialAddress(credentialAddress: String): Client
Client
- id: Long
- name: String
- credentialAddress: String
- email: String
StorageServiceDAO
+ getByCredentialAndTrustAddress(credentialAddress: String, trustContractAddress: String): StorageService
1
n

FIGURE 8. Application class diagram for the ICS.

uses the SSTMC wrapper to interact with the SSTMC instance stored in the BN loaded from the access address registered at the integrity verification contract as the SC responsible for managing the trust attributed to the CSS. Using the SSTMC "getTrustLevel" method, the ICS Application obtains the updated trust level assigned to the CSS and, according to Table 1, calculates the number of files to verify on that day and the number of challenges to submit per file. Based on the results, the ICS application selects the files to check from active contracts, gets their CFSMC instance access addresses, generates the challenges, and submits them using the CFSMC "submitChallenge" method through the "CfsmcContract" class and the CFSMC wrapper.

The "checkRequests" method, on the other hand, is responsible for performing the verification of pending challenges and recording the verification failures generated by the lack of response by the CSS. The "checkRequests" method synchronously triggers the "checkExpiredRequests" method of the "RequestService" class at the scheduled time during its daily self-running. The "RequestService" class selects pending challenge submissions in the local database, for which the maximum deadlines for the CSS to record the corresponding responses have expired (24 hours). For each expired challenge found, the "checkExpiredRequests" method performs a new submission of that challenge and increases the number of attempts. After the third attempt, if there is still no answer, it triggers the CFSMC "verifyChallenge" method that is responsible for registering the lacking response in the BN, and asks the SSTMC instance to reduce the trust level assigned to the CSS.

V. PROTOCOL VALIDATION

Aiming to validate the proposed protocol, we performed two testing sessions using a controlled computational environment. Before this, we developed the software applications to carry out the functionalities established by the proposed protocol. The used computational environment is a laboratory comprised of six virtual machines (VMs), where one plays the role of Client, two the role of ICS, and three the role of the CSS. We configured each VM with a "Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz" processor, 64 bits, one *core*, 100 GB HD, Linux Operating System, *kernel* version 4.15.0-55-generic, distribution "Ubuntu 18.04.4 LTS", and 4 GB of RAM, except for the three VMs playing the role of CSS which were configured with 12 GB of RAM each.

In the testing environment, we configured a private BN *Ethereum* comprised of 6 nodes, with one node in each of the VMs. We used the Docker tools "docker-ce" and "docker-composer", and also the "puppeth" tool which belongs to the package *Ethereum* for the installation and configuration of the BN nodes in the VMs. In this BN configuration, we created an account for each instance of the Client, ICS, and CSS roles. Also, we created an independent account for each node of the BN. When we created the Ethereum "Genesis" used to start the BN, we defined the following parameters:

- The consensus algorithm: defined as the algorithm that all network nodes use to reach an agreement on the transactions that will comprise each block; considering the available infrastructure limitation to carry out the tests, we chose the "proof-of-authority" algorithm, which is based on the node reputation and suitable for use in private networks, where only previously authorized nodes are allowed to participate in the validation process of new blocks, since they present both better performance and greater energy efficiency than the "proof-of-work" algorithm.
- The interval between blocks: defined as the time that each network node will wait before trying to start the generation of a new block to be inserted in the Blockchain, which directly affects transaction processing performance; after performing preliminary tests, the results showed that time options less than 5 *seconds* did not present a perceptible improvement in the processing performance of the transactions tested, which is why we chose 5 *seconds* as the standard time interval.
- The initial fund: defined as the amount of cryptocurrency created and distributed to one or more accounts in the BN, with which financial transactions are carried out within the network; despite not influencing performance, as Ethereum BNs require payment in cryptocurrency by the application or the SC that submits a transaction to the node that processes the transaction and inserts it into a new block in the Blockchain, we chose to distribute 1 *Ether* (1×10^{18} *Wei*) for each of the accounts created in the network, because this value is sufficient to carry out the tests without the risk of running out of funds.
- The gas floor ("–miner.gastarget"): defined as the ideal number of transactions that should be processed and inserted in a single block, according to the amount of gas consumed in the execution of these transactions; the value assigned to this parameter can influence the performance according to the size and frequency of the transactions; however, there are no previous results that can support the best choice; therefore, we chose to use the default value of $7,500,000$ *gas* for all nodes, suggested by "Puppeth", the tool used to configure new nodes.
- The gas ceil ("–miner-gaslimit"): defined as the maximum amount of gas that can be consumed by the transactions that will be processed and included in a single block; as in the "gas floor" parameter and for the same reason, we chose to use the default value of $10,000,000$ *gas* suggested by the "Puppeth" tool for all nodes.
- The gas price ("–miner.gasprice"): defined as the minimum cryptocurrency value accepted as remuneration for each unit of gas consumed in a transaction processing for each node; this value can influence network performance only if each of the received transactions offers as payment a different gas price value when the nodes prioritize those with highest prices, a situation that will not occur in our tests; for this choice process we considered the premise that all nodes will offer the same gas price; since the chosen value does not directly influence the result of the predicted tests, we chose the price value of 100 *Wei* for each gas unit.

For the Client, CSS, and ICS applications, the software Postgresql, version 10.12, was used as the DBMS. For the ICS and CSS applications, the software Glassfish Server Open Source Edition, version 5.0.1, was used as the application server. Additionally, these applications, whenever submitting a new transaction to the BN, must inform the value in cryptocurrency offered as payment for each unit of gas consumed by the transaction to the node that processes it. As this gas price must be greater or equal to the minimum gas price defined by the nodes, and will not generate a failure risk due to the lack of funds, we chose the value of 150 *Wei* as the gas price for all applications. Figure 9 presents an overview of transaction processing in the BN, whose origin was the submission of a file to a CSS. We generated this view using the *Ethstats* tool [27].

A. FIRST TEST SESSION

For the first test session, we submitted six files of different sizes to each of the three available CSSs: 52MB, 196MB,

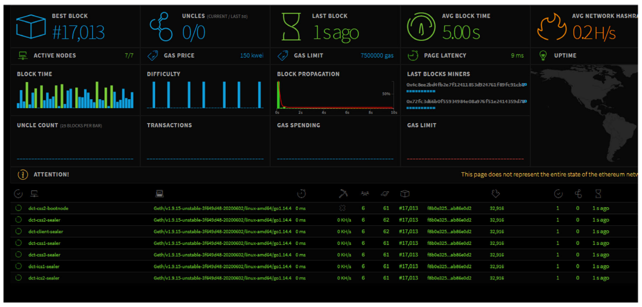
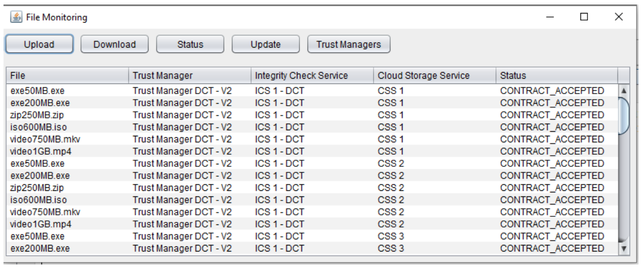


FIGURE 9. View of the BN transaction processing.



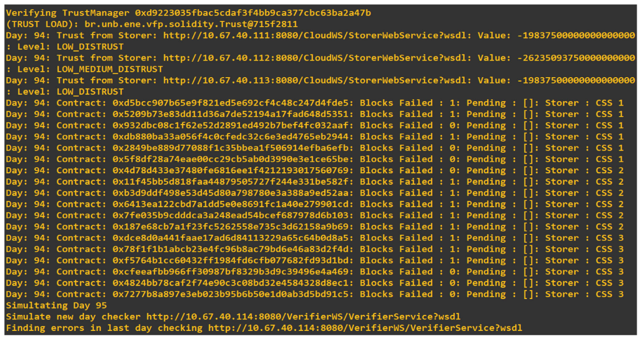
File	Trust Manager	Integrity Check Service	Cloud Storage Service	Status
exe50MB.exe	Trust Manager DCT - V2	ICS 1 - DCT	CSS 1	CONTRACT_ACCEPTED
exe200MB.exe	Trust Manager DCT - V2	ICS 1 - DCT	CSS 1	CONTRACT_ACCEPTED
zip250MB.zip	Trust Manager DCT - V2	ICS 1 - DCT	CSS 1	CONTRACT_ACCEPTED
iso600MB.iso	Trust Manager DCT - V2	ICS 1 - DCT	CSS 1	CONTRACT_ACCEPTED
video750MB.mkv	Trust Manager DCT - V2	ICS 1 - DCT	CSS 1	CONTRACT_ACCEPTED
video1GB.mp4	Trust Manager DCT - V2	ICS 1 - DCT	CSS 1	CONTRACT_ACCEPTED
exe50MB.exe	Trust Manager DCT - V2	ICS 1 - DCT	CSS 2	CONTRACT_ACCEPTED
exe200MB.exe	Trust Manager DCT - V2	ICS 1 - DCT	CSS 2	CONTRACT_ACCEPTED
zip250MB.zip	Trust Manager DCT - V2	ICS 1 - DCT	CSS 2	CONTRACT_ACCEPTED
iso600MB.iso	Trust Manager DCT - V2	ICS 1 - DCT	CSS 2	CONTRACT_ACCEPTED
video750MB.mkv	Trust Manager DCT - V2	ICS 1 - DCT	CSS 2	CONTRACT_ACCEPTED
video1GB.mp4	Trust Manager DCT - V2	ICS 1 - DCT	CSS 2	CONTRACT_ACCEPTED
exe50MB.exe	Trust Manager DCT - V2	ICS 1 - DCT	CSS 3	CONTRACT_ACCEPTED
exe200MB.exe	Trust Manager DCT - V2	ICS 1 - DCT	CSS 3	CONTRACT_ACCEPTED

FIGURE 10. Files submitted to the three CSSs in the first testing session.

243MB, 593MB, 750MB, and 1 GB. In addition, we assigned the responsibility of verifying the integrity of all these files to a single ICS. We defined one year as the estimated time of storage of these files in the cloud, and we linked the submission of all the files to the same SSTMC instance. We exclusively inserted this SSTMC instance into the BN for this test session. All CSS and ICS instances were configured in this testing session to self-register in the referred SSTMC. Figure 10 shows the Client application with the registration of the files submitted in this test session.

The main objective of the testing session was to validate the protocol's ability to monitor and identify, in a set of files stored in the cloud, those files whose original content had undergone some change. Furthermore, also to determine the average time required to identify them and to demonstrate the variation in the trust assigned to the CSS while the protocol spotted each of the corrupted files. Aiming at this goal, we randomly selected an address, according to each file size, for each of the six files stored in each of the three CSSs. Using the chosen address as a starting point, we changed 1 byte of content in each file.

We implemented an application called "Simulator" to make the protocol validation process more agile, whose responsibility is to simulate the passing of days, forcing the ICS instance to execute its daily challenge generation protocol at a specified interval of time. To this application, we added a routine to consult the SSTMC at the end of each simulated day, recording both the trust value and the level of each CSS after processing the challenges. This routine also consulted the CFSMC instance of each file stored in each CSS, recording the existence of pending challenges and



```
Verifying TrustManager 0xd9223035fbac5cdaf3f4bb9ca377cbc63ba2a47b
(TRUST LOAD): br.unb.ene.vfp.solidity.Trust@715f2811
Day: 94: Trust from Storer: http://10.67.40.111:8080/CloudWS/StorerWebService?wsdl: Value: -19837500000000000000
: Level: LOW_DISTRUST
Day: 94: Trust from Storer: http://10.67.40.112:8080/CloudWS/StorerWebService?wsdl: Value: -26235093750000000000
: Level: LOW_MEDIUM_DISTRUST
Day: 94: Trust from Storer: http://10.67.40.113:8080/CloudWS/StorerWebService?wsdl: Value: -19837500000000000000
: Level: LOW_DISTRUST
Day: 94: Contract: 0xd5bcc907b65e9f821ed5e692cf4c48c247d4fde5: Blocks Failed : 1: Pending : []: Storer : CSS 1
Day: 94: Contract: 0x5209b73e83dd11d36a7de52194a17fad648d5351: Blocks Failed : 1: Pending : []: Storer : CSS 1
Day: 94: Contract: 0x932dbc08c1f62e52d2891ed492b7bef4fc032aaf: Blocks Failed : 0: Pending : []: Storer : CSS 1
Day: 94: Contract: 0xdb880ba33a056f4c0cfedc32c6e3ed4765eb2944: Blocks Failed : 1: Pending : []: Storer : CSS 1
Day: 94: Contract: 0x2849be889d77088f1c35bbea1f506914efba6efb: Blocks Failed : 0: Pending : []: Storer : CSS 1
Day: 94: Contract: 0x5f8df28a74eae00cc29cb5ab0d3990e3e1ce65be: Blocks Failed : 0: Pending : []: Storer : CSS 1
Day: 94: Contract: 0x4d78d433e37480fe6816ee1f4212193017560769: Blocks Failed : 0: Pending : []: Storer : CSS 2
Day: 94: Contract: 0x11f45bb5d818faa44879505727f244e331be582f: Blocks Failed : 1: Pending : []: Storer : CSS 2
Day: 94: Contract: 0xb3d9ddf498e53d45d80a798780e3a388a9ed52aa: Blocks Failed : 1: Pending : []: Storer : CSS 2
Day: 94: Contract: 0x6413ea122cbd7a1dd5e0e8691fc1a40e279901cd: Blocks Failed : 1: Pending : []: Storer : CSS 2
Day: 94: Contract: 0x7fe035b9cdddca3a248ead54bcef687978d6b103: Blocks Failed : 1: Pending : []: Storer : CSS 2
Day: 94: Contract: 0x187e68cb7a1f23fc5262558e735c3d62158a9b69: Blocks Failed : 1: Pending : []: Storer : CSS 2
Day: 94: Contract: 0xdce8d0a441faae17ad6d84113229a65c64b0d8a5: Blocks Failed : 1: Pending : []: Storer : CSS 3
Day: 94: Contract: 0x78f1f1b1abcb23e4fc96b8ac79bd6e46a83d2f4d: Blocks Failed : 1: Pending : []: Storer : CSS 3
Day: 94: Contract: 0xf5764b1cc60432ff1984fd6cfb077682fd93d1bd: Blocks Failed : 1: Pending : []: Storer : CSS 3
Day: 94: Contract: 0xcfeeafbb966ff30987bf8329b3d9c39496e4a469: Blocks Failed : 0: Pending : []: Storer : CSS 3
Day: 94: Contract: 0x4824bb78caf2f74e90c3c08bd32e4584328d8ec1: Blocks Failed : 0: Pending : []: Storer : CSS 3
Day: 94: Contract: 0x7277b8a897e3eb023b95b6b50e1d0ab3d5bd91c5: Blocks Failed : 0: Pending : []: Storer : CSS 3
Simultating Day 95
Simulate new day checker http://10.67.40.114:8080/VerifierWS/VerifierService?wsdl
Finding errors in last day checking http://10.67.40.114:8080/VerifierWS/VerifierService?wsdl
```

FIGURE 11. The result captured by the "Simulator" application.

TABLE 2. Results of the first testing session.

File Corrupted	Failure Identification Day		
	CSS 1	CSS 2	CSS 3
1st	31st	34th	4th
2nd	41st	41st	22nd
3rd	81st	62nd	28th
4th	101st	75th	100th
5th	106th	92nd	116th
6th	112nd	99th	121st

TABLE 3. Summary of the results of the first testing session.

CSS	Files checked	Days to identify a corrupted file		
		Minimum	Maximum	Average Time
1	6	31	112	78.6
2	6	34	99	67.1
3	6	4	121	65.1
General	18	4	121	70.3

identified failures. We used 2 minutes as the interval during which the Simulator remains waiting for challenges to be processed, before capturing all results and simulating a new day. Figure 11 shows the result captured by the Simulator after processing the challenges of the 94th day of the first test session.

After the completion of the first testing session, we classified and organized the obtained results to present the number of days that the protocol spent to identify the flaws inserted in each of the 18 files distributed in the three CSSs used. Table 3 shows the complete results for each CSS. In Table 3, we present a summary of the results of the first testing session, and in Figure 12, we expose a comparative graph of the time taken to identify the corrupted files in each CSS.

The initial trust level of the CSS we used in the tests was the default value for a CSS recently enrolled in the SSTMC instance, the "Low Trust" level, with the trust value equal to zero. Table 4 shows the variation of the trust value for each of the CSSs during the first testing session.

B. SECOND TEST SESSION

For the second testing session, we used the same six files described in session V-A. In addition, we employed three

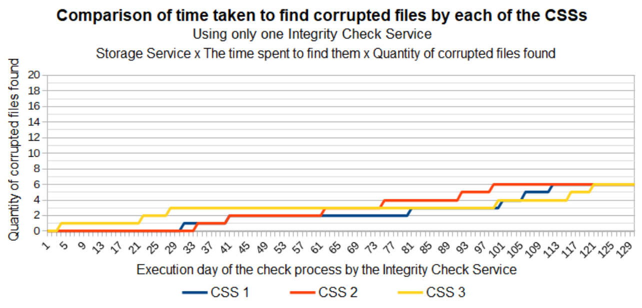


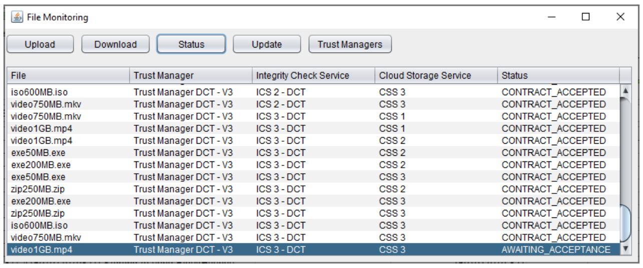
FIGURE 12. Comparison of the periods taken to identify corrupted files in each of the CSSs using a single ICS.

TABLE 4. Variation of the trust level in each of the CSSs in the first test session.

CSS	Day	Trust Value	Trust Level
1	1st	0	Low trust
	31st	-1.5×10^{19}	Low distrust
	106th	$-2.623509375 \times 10^{19}$	Low-medium distrust
2	1st	0	Low trust
	34th	-1.5×10^{19}	Low distrust
	92nd	$-2.623509375 \times 10^{19}$	Low-medium distrust
3	1st	0	Low trust
	4th	-1.5×10^{19}	Low distrust
	116th	$-2.623509375 \times 10^{19}$	Low-medium distrust

TABLE 5. Features of the second test session.

Feature	Amount
Cloud file storage time (years)	1
Verification cycles calculated per file	20
CSS used	3
ICS used	3
Files submitted per size	9
Files submitted for each CSS	18
Files registered for verification at each ICS	18
Total files monitored in CSS	54
Transactions generated in the BN per file submitted	80



File	Trust Manager	Integrity Check Service	Cloud Storage Service	Status
iso600MB.iso	Trust Manager DCT - V3	ICS 2 - DCT	CSS 3	CONTRACT_ACCEPTED
video750MB.mkv	Trust Manager DCT - V3	ICS 2 - DCT	CSS 3	CONTRACT_ACCEPTED
video750MB.mkv	Trust Manager DCT - V3	ICS 3 - DCT	CSS 1	CONTRACT_ACCEPTED
video1GB.mp4	Trust Manager DCT - V3	ICS 3 - DCT	CSS 1	CONTRACT_ACCEPTED
video1GB.mp4	Trust Manager DCT - V3	ICS 3 - DCT	CSS 2	CONTRACT_ACCEPTED
exe50MB.exe	Trust Manager DCT - V3	ICS 3 - DCT	CSS 2	CONTRACT_ACCEPTED
exe200MB.exe	Trust Manager DCT - V3	ICS 3 - DCT	CSS 2	CONTRACT_ACCEPTED
exe50MB.exe	Trust Manager DCT - V3	ICS 3 - DCT	CSS 3	CONTRACT_ACCEPTED
zip250MB.zip	Trust Manager DCT - V3	ICS 3 - DCT	CSS 2	CONTRACT_ACCEPTED
exe200MB.exe	Trust Manager DCT - V3	ICS 3 - DCT	CSS 3	CONTRACT_ACCEPTED
zip250MB.zip	Trust Manager DCT - V3	ICS 3 - DCT	CSS 3	CONTRACT_ACCEPTED
iso600MB.iso	Trust Manager DCT - V3	ICS 3 - DCT	CSS 3	CONTRACT_ACCEPTED
video750MB.mkv	Trust Manager DCT - V3	ICS 3 - DCT	CSS 3	CONTRACT_ACCEPTED
video1GB.mp4	Trust Manager DCT - V3	ICS 3 - DCT	CSS 3	AWAITING_ACCEPTANCE

FIGURE 13. Files submitted to the three CSSs in the second testing session.

different ICSs to perform the integrity check. Considering that the testing environment, described in Section V, had only two VMs for the role of ICS, the VM for the Client role was also used to host an instance of the application for the ICS role, sharing both services.

The main objective of this second testing session was to validate the behavior of the proposed protocol when information was shared about the CSS that stores files between the ICS responsible for monitoring these files, with the assignment of a shareable trust level to the CSS using the same SC. For these tests, we inserted a new SSTMC instance in the BN, and all ICS and CSS instances were configured to self-register in this new SSTMC instance to offer their services.

Using the Client application instance, together with the new SSTMC instance as its trust manager, we submitted three copies of each one of the six tested files to each CSS, and for each submission, we linked the file copy to another ICS. When the three copies of each file were stored in the CSS, despite having the same source file, they had different content because the Client application used a different set of keys to encrypt each copy. Table 5 presents the characteristics of the tests performed in this session.

Although the process performed was a test, we followed all steps established by the proposed protocol for each file submission, of which we highlight the validation by the CSS of the integrity verification information stored in the CFSMC instance linked to the file received for storage. Figure 13 shows the Client application screen while waiting for the CSS approval of the storage contract of the most recent file submitted to the last CSS used.

During the phase file copies were submitted from the Client for storage in a CSS, where we stored 54 files in three CSSs, we recorded each time period consumed during file preparation, challenge generation, and transaction processing generated in the BN. Table 6 shows the analysis of these records in each of the main processes of the proposed protocol and the average time taken to submit the files to the CSS.

For the preparation of the 54 files stored in the three CSSs monitored and used in this testing session, we reused the 18 addresses chosen in the first testing session and randomly selected another 36 according to file size. From these chosen addresses, we changed one byte of content in each of the 54 files. We used the "Simulator" application to force the three ICSs to anticipate the respective daily verification processes, executing that at two-minute intervals. In addition, the "Simulator" interacted with the Client application to perform the automatic renewal of the integrity verification contract with the ICS, based on the simulated period, instead of using the expiration date of the initial agreement registered at the CFSMC instance of each file.

This testing took three months of execution time of the proposed protocol to monitor 54 files, where each one of three ICSs simultaneously generated challenges to check 18 files equally distributed in three CSSs. Table 7 presents the complete results per CSS. In Table 8, we present the summary of these results, and Figure 14 shows a comparative graph of the periods needed to identify corrupted files in each CSS.

TABLE 6. Results of the file submission process to the cloud.

Processes	Average execution time (seconds)					
	52 MB	196 MB	243 MB	593 MB	750 MB	1 GB
File encryption	2.938	9.175	11.867	27.608	34.198	50.824
Hash generation	0.402	1.036	1.311	3.612	4.913	7.476
Insertion of the CFSMC in the BN	15.187	15.182	15.179	15.206	15.198	15.201
Generation of the challenges	22.357	36.582	41.271	78.505	99.401	134.845
Insertion of the challenges in the DBMS	6.683	6.437	6.532	6.052	8.194	6.191
Processing of the transactions in the BN	403.876	401.035	426.848	400.987	434.586	414.197

TABLE 7. Results of the second testing session.

Corrupted File	Failure Identification Day		
	CSS 1	CSS 2	CSS 3
1st	19th	6th	3rd
2nd	25th	9th	4th
3rd	28th	10th	22nd
4th	38th	27th	26th
5th	50th	31st	28th
6th	55th	31st	39th
7th	67th	36th	40th
8th	69th	46th	42nd
9th	70th	55th	52nd
10th	79th	56th	64th
11st	80th	65th	66th
12nd	84th	67th	67th
13rd	85th	72nd	70th
14th	89th	74th	71st
15th	89th	81st	75th
16th	94th	86th	79th
17th	95th	93rd	89th
18th	97th	94th	93rd

TABLE 8. Summary of the results of the second testing session.

CSS	Checked Files	Days to Identify each Corrupted File		
		Minimum	Maximum	Average Time
1	18	19	97	67.4
2	18	6	94	52.2
3	18	3	93	51.7
General	54	4	97	57.07

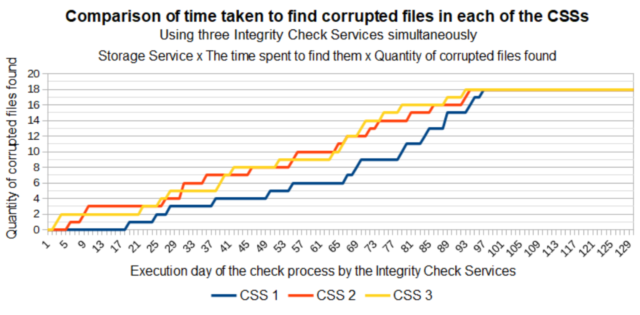


FIGURE 14. Comparison of the periods taken to identify corrupted files in each of the CSSs using three ICSs.

The initial trust value and trust level attributed to the three CSS used in this second testing session were the same used for a new CSS recently inserted in an SSTMC instance. This is the trust value equal to zero and, consequently, the trust level stated to "Low trust", according to Table 1. Table 9 shows the trust value changes over the protocol execution time.

TABLE 9. Variation in the trust value per CSS in the second test session.

CSS	Day	Trust Value	Trust Level
1	1st	0	Low trust
	19th	-1.5×10^{19}	Low distrust
	50th	$-2.623509375 \times 10^{19}$	Low-medium distrust
	79	$-5.27681443787 \times 10^{19}$	Medium-high distrust
2	1st	0	Low trust
	6th	-1.5×10^{19}	Low distrust
	31st	$-3.01703578125 \times 10^{19}$	Low-medium distrust
	56th	$-5.27681443787 \times 10^{19}$	Medium-high distrust
3	1st	0	Low trust
	3rd	-1.5×10^{19}	Low distrust
	28th	$-2.623509375 \times 10^{19}$	Low-medium distrust
	64th	$-5.27681443787 \times 10^{19}$	Medium-high distrust

C. ANALYSIS AND COMPARISON OF THE RESULTS

As shown in Table 3, the first testing session (Section V-A) confirmed the effectiveness of the proposed protocol implementation using a BN and SCs for the monitoring of files stored in the cloud. In the 121 days of protocol execution, it identified all 18 monitored files that contained a single corrupted byte. The results obtained also demonstrated the effectiveness of the SC SSTMC as trust management, as can be seen in the independent variations in the trust levels assigned to each CSS presented in Table 4.

From the results obtained in the second testing session (Section V-B), presented in Table 8, it is possible to observe that the sharing of monitoring results performed by different ICSs accelerated the corrupted file identifying process. Table 8 shows that the average identification time for six corrupted files in each CSS analyzed dropped from approximately 112 days to 42 days, representing a reduction of 62.5% in the average time consumed by the protocol to identify the flaws. Even when considering all 54 files monitored in the three CSS, the sharing of the three ICS results enabled the identification of all corrupted files in the CSS in an average time of 95 days, a reduction of approximately 15%, in comparison with an average of 112 days obtained using a single ICS.

As shown in the comparison between the variations in the trust levels attributed to each CSS obtained in the first testing session (Table 4) and the second testing session (Table 9), the increase in the number of monitored files in addition to the sharing of monitoring results reduced the average time consumed for the debasing of the trust level assigned to the CSS to the ''Low-medium distrust'' level. The time spent on this debasement decreased from about 104 days to approximately 36 days, representing a reduction of 65%. Also, the sharing of monitoring results allowed the debasing of trust level assigned to the CSS to the ''medium-high distrust'' level to drop to 66 days of protocol execution, on average.

VI. CONCLUSION AND FUTURE WORK

This research presented and validated a solution to monitor the integrity of files stored in a CSS using Blockchain and SC technologies. This solution, which is composed of a protocol and an unabridged reference implementation, leverages the results of a previous study [8], taking advantage of its strengths such as the use of trust concepts and the challenge-based integrity check mechanism using only symmetric cryptography.

Among the main contributions of this work we highlight the following: i) the use of a storage infrastructure in a BN which guaranteed transparency, security, and the possibility of performing audits; ii) the automation and decentralization of the integrity check result analysis process which prevented collusion between the ICS and the CSS; iii) the decentralization and sharing of the trust level calculation process through a SC preventing ICS attacks against the CSS reputation; and, iv) the automation of the management of the file integrity verification contracts allowing ICS replacement without interfering in the process security.

The use of Blockchain and SC technologies was fundamental, as they allowed the development of a new protocol, which introduced advances such as low processing costs and reduced network traffic, while adding security, auditing capacity, flexibility, and independence between roles. Another important contribution was the secure sharing of the results, which improved the capacity to react to the observed events. The development of a complete reference implementation, covering both the expected functionalities performed within the scope of the BN, through the SCs, and the responsibilities provided for each role (Client, CSS, and ICS), allowed us to improve the previous protocol [8]. The enhanced new version takes advantage of the characteristics of the technologies adopted, and allows the execution of tests to evaluate the effectiveness of the proposed solution.

The validation tests carried out using the reference implementation with a private BN in a controlled laboratory demonstrated the feasibility of using the Blockchain and SC technologies as a secure means for the recording and auditable processing of information exchanged between the Client, the CSS, and the ICS. Also, the tests confirmed the effectiveness of the newly proposed protocol in detecting corrupted files, as well as the efficiency generated by sharing the monitoring results performed by more than one ICS. The test with three ICSs showed, on average, a reduction of approximately 65% in the time needed to identify the same amount of corrupted files with just one ICS.

It is important to highlight the faultless functioning of the auditing processes of the information to validate challenges, stored in the CFSMC instances linked to files received for storage, as carried out by the CSS before the acceptance of the respective storage contracts. This function reduced the possibility of an attack of an evil Client on the credibility of the service provided by the CSS. Another process successfully performed during the tests was the hiring and renewal of contracts with the ICS, through the respective submission of information to generate challenges for the periods of these contracts. This feature makes it possible to replace the ICS during the storage period of the file in the cloud, without compromising the confidentiality of the challenges.

As future work, we intend to implement an extension for the case when a file owner chooses to store copies of the file in more than one CSS, and the protocol identifies an integrity failure in any of these copies, thus automatically recovering the broken file copy by replacing the corrupted content with information from its healthy copies. We also intend to take advantage of the fact that the proposed protocol uses Blockchain technology, generally used to register financial transactions, to offer a protocol that, with the use of cryptocurrency, allows the Client to autonomously remunerate the services provided by the CSS and the ICS, based on the verified results and the assigned trust levels.

Also as future work, based on the solutions proposed in [28]–[30], it would be interesting to carry out a study on the feasibility of applying artificial intelligence techniques, such as ''Savitzky-Golay'' filters, ''augmented Dickey-Fuller'' tests, and ''Haar wavelet transforms'' on SCs, in order to predict the behavior of the CSS and the ICS. Also, based on the analysis of requests to SCs (challenge and response records, and storage authorizations), we see the possibility of allowing the SCs to adapt to environmental changes. Among other advances, we have the intention of studying the proactive identification of unexpected ICS behavior, and the adapting of the trust level variation speed to consider not only the number of correct answers, but also the workload required from each CSS, thus improving the balance between the monitored services.

ACKNOWLEDGMENT

The authors would like to thank the Science and Technology Department of the Brazilian Army for providing all the laboratory infrastructure employed in this study.

REFERENCES

[1] A. Negi and A. Goyal, ''Optimizing fully homomorphic encryption algorithm using RSA and Diffie-Hellman approach in cloud computing,'' *Int. J. Comput. Sci. Eng.*, vol. 6, no. 5, pp. 215–220, May 2018, doi: 10.26438/ijcse/v6i5.215220.

[2] J. Bi, H. Yuan, and M. Zhou, ''Temporal prediction of multiapplication consolidated workloads in distributed clouds,'' *IEEE Trans. Autom. Sci. Eng.*, vol. 16, no. 4, pp. 1763–1773, Oct. 2019, doi: 10.1109/TASE.2019.2895801.

[3] R. Chakarov. *Cloud Computing Statistics 2019*. Accessed: Sep. 9, 2019. [Online]. Available: https://techjury.net/stats-about/cloud-computing

[4] H. Yuan, J. Bi, M. Zhou, Q. Liu, and A. C. Ammari, "Biobjective task scheduling for distributed green data centers," *IEEE Trans. Autom. Sci. Eng.*, early access, Jan. 7, 2020, doi: 10.1109/TASE.2019.2958979.

[5] A. Wilczyński and J. Kołodziej, "Modelling and simulation of security-aware task scheduling in cloud computing based on blockchain technology," *Simul. Model. Pract. Theory*, vol. 99, Feb. 2020, Art. no. 102038, doi: 10.1016/j.simpat.2019.102038.

[6] L. W. Cong and Z. He, "Blockchain disruption and smart contracts," *Rev. Financial Stud.*, vol. 32, no. 5, pp. 1754–1797, May 2019, doi: 10.1093/rfs/hhz007.

[7] M. Alharby and A. V. Moorsel, "Blockchain-based smart contracts: A systematic mapping study," in *Proc. Comput. Sci. Inf. Technol. (CS IT)*, Aug. 2017, pp. 125–140, doi: 10.5121/csit.2017.71011.

[8] A. Pinheiro, E. Dias Canedo, R. de Sousa Junior, R. de Oliveira Albuquerque, L. G. Villalba, and T.-H. Kim, "Security architecture and protocol for trust verifications regarding the integrity of files stored in cloud services," *Sensors*, vol. 18, no. 3, p. 753, Mar. 2018, doi: 10.3390/s18030753.

[9] J. Xue, C. Xu, J. Zhao, and J. Ma, "Identity-based public auditing for cloud storage systems against malicious auditors via blockchain," *Sci. China Inf. Sci.*, vol. 62, no. 3, p. 32104, Mar. 2019, doi: 10.1007/s11432-018-9462-0.

[10] H. Yu, Z. Yang, and R. O. Sinnott, "Decentralized big data auditing for smart city environments leveraging blockchain technology," *IEEE Access*, vol. 7, pp. 6288–6296, 2019, doi: 10.1109/ACCESS.2018.2888940.

[11] S. Wang, X. Tang, Y. Zhang, and J. Chen, "Auditable protocols for fair payment and physical asset delivery based on smart contracts," *IEEE Access*, vol. 7, pp. 109439–109453, 2019, doi: 10.1109/ACCESS.2019.2933860.

[12] A. Ahmad, M. Saad, L. Njilla, C. Kamhoua, M. Bassiouni, and A. Mohaisen, "Blocktrail: A scalable multichain solution for blockchain-based audit trails," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Shanghai, China, Jun. 2019, pp. 1–6, doi: 10.1109/ICC.2019.8761448.

[13] R. de Oliveira Albuquerque, L. J. García Villalba, A. L. S. Orozco, R. T. de Sousa Júnior, and T.-H. Kim, "Leveraging information security and computational trust for cybersecurity," *J. Supercomput.*, vol. 72, no. 10, pp. 3729–3763, Oct. 2016, doi: 10.1007/s11227-015-1543-4.

[14] A. M. Mohammed and F. A. Omara, "A trust-based ranking model for cloud service providers in cloud computing," in *Internet Things–Applications Future*, A. Z. Ghalwash, N. El Khameesy, and D. A. Magdi, A. Joshi, Eds. Singapore: Springer, 2020, pp. 325–346, doi: 10.1007/978-981-15-3075-3_22.

[15] M. E. Ghazouani, M. A. E. Kiram, and L. Er-Rajy, "Blockchain & multi-agent system: A new promising approach for cloud data integrity auditing with deduplication," *Int. J. Commun. Netw. Inf. Secur.*, vol. 11, no. 1, pp. 175–184, 2019.

[16] F. Coelho, "An (almost) constant-effort solution-verification proof-of-work protocol based on Merkle trees," in *Progress in Cryptology—AFRICACRYPT* (Lecture Notes in Computer Science), vol. 5023, S. Vaudenay, Ed. Berlin, Germany: Springer, 2008, pp. 80–93, doi: 10.1007/978-3-540-68164-9_6.

[17] X. Tang, Y. Huang, C.-C. Chang, and L. Zhou, "Efficient real-time integrity auditing with privacy-preserving arbitration for images in cloud storage system," *IEEE Access*, vol. 7, pp. 33009–33023, 2019, doi: 10.1109/ACCESS.2019.2904040.

[18] P. Wei, D. Wang, Y. Zhao, S. K. S. Tyagi, and N. Kumar, "Blockchain data-based cloud data integrity protection mechanism," *Future Gener. Comput. Syst.*, vol. 102, pp. 902–911, Jan. 2020, doi: 10.1016/j.future.2019.09.028.

[19] S. Marsh, "Formalizing trust as a computational concept," Ph.D. dissertation, Dept. Math. Comput. Sci., Univ. Stirling, Stirling, Scotland, U.K., 1994.

[20] Ethereum Foundation. *Solidity*. Accessed: May 10, 2020. [Online]. Available: https://solidity.readthedocs.io

[21] M. Valenta and P. Sandner, "Comparison of ethereum, hyperledger fabric and corda," in *Proc. Frankfurt School Blockchain Center Work. Paper*, Jun. 2017, pp. 1–8.

[22] V. Buterin. (2014). *Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform*. Accessed: Apr. 2, 2020. [Online]. Available: http://bitpaper.info/paper/5634472569470976

[23] D. R. Heffelfinger, *Java EE 8 Application Development: Develop Enterprise Applications Using the Latest Versions of CDI, JAX-RS, JSON-B, JPA, Security, and More*. Birmingham, U.K.: Packt, 2017.

[24] J. Juneau, "Java EE containers," in *Java EE 8 Recipes: A Problem-Solution Approach*. Berkeley, CA, USA: Apress, 2018, pp. 523–557, doi: 10.1007/978-1-4842-3594-2_12.

[25] The PostgreSQL Global Development Group. *Postgresql: The World's Most Advanced Open Source Relational Database*. Accessed: May 14, 2020. [Online]. Available: https://www.postgresql.org

[26] C. Svensson. (2017). *Blockchain: Using Cryptocurrency With Java*. [Online]. Available: https://www.janeiro/fevereiro

[27] C. Johnston. *Ethereum Ethstats: Learning The Ethereum Blockchain Through Its Metrics*. Accessed: Jun. 4, 2020. [Online]. Available: https://imti.co/ethereum-ethstats

[28] R. Gupta, S. Tanwar, F. Al-Turjman, P. Italiya, A. Nauman, and S. W. Kim, "Smart contract privacy protection using AI in cyber-physical systems: Tools, techniques and challenges," *IEEE Access*, vol. 8, pp. 24746–24772, 2020, doi: 10.1109/ACCESS.2020.2970576.

[29] J. Bi, H. Yuan, L. Zhang, and J. Zhang, "SGW-SCN: An integrated machine learning approach for workload forecasting in geo-distributed cloud data centers," *Inf. Sci.*, vol. 481, pp. 57–68, May 2019, doi: 10.1016/j.ins.2018.12.027.

[30] S. Gao, M. Zhou, Y. Wang, J. Cheng, H. Yachi, and J. Wang, "Dendritic neuron model with effective learning algorithms for classification, approximation, and prediction," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 601–614, Feb. 2019, doi: 10.1109/TNNLS.2018.2846646.

ALEXANDRE PINHEIRO (Member, IEEE) received the bachelor's degree in information systems from the Lutheran University of Brazil (ULBRA), Canoas-RS, Brazil, in 2005, and the master's degree in electrical engineering from the University of Brasília (UnB), Brasília-DF, Brazil, in 2016, where he is currently pursuing the Ph.D. degree with the Electrical Engineering Department. He holds specialization in cryptography and network security from the Fluminense Federal University (UFF), Niterói-RJ, Brazil, in 2012, and a specialization in information security management from UnB, in 2014, and a specialization in military sciences from the Captains Career School of Brazilian Army (EsAO), Rio de Janeiro, Brazil, in 2018. He is the Captain of the Complementary Officers Corps of the Brazilian Army and is currently serving at the Science and Technology Department, Brasília-DF, Brazil. His research interests include cryptographic protocols, cloud, blockchain, network security, and vehicular technologies, with an emphasis on cloud security.

EDNA DIAS CANEDO (Member, IEEE) received the bachelor's degree in systems analysis from the University Salgado de Oliveira, in 1999, the master's degree in computer science from the Software Engineering Group, Federal University of Campina Grande, in 2002, and the Ph.D. degree in electrical engineering from the Network Engineering Group, University of Brasilia, in 2012. She is currently a tenure track Professor with the Department of Computer Science, University of Brasilia, Brazil, where she works since April 2010. Her research interests are mostly in software engineering, including requirements engineering, database, software systems, cloud computing, as well as usability and empirical methods.

RAFAEL TIMÓTEO DE SOUSA, JR. (Senior Member, IEEE) received the bachelor's degree in electrical engineering from the Federal University of Paraíba — UFPB, Campina Grande, Brazil, in 1984, the master's degree in computing and information systems from the Ecole Supérieure d'Electricité — Supélec, Rennes, France, in 1985, and the Ph.D. degree in telecommunications and signal processing from the University of Rennes 1, Rennes, France, in 1988. He was a Visiting Researcher with the Group for Security of Information Systems and Networks (SSIR), Ecole Supérieure d'Electricité – Supélec, Rennes, France, from 2006 to 2007. He worked in the private sector from 1988 to 1996. Since 1996, he has been a Network Engineering Associate Professor with the Electrical Engineering Department, University of Brasília (UnB), Brazil, where he is currently the Coordinator of the Professional Post-Graduate Program on Electrical Engineering (PPEE) and supervises the Decision Technologies Laboratory (LATITUDE). He is the Chair of the IEEE VTS Centro-Norte Brasil Chapter (IEEE VTS Chapter of the Year 2019) and of the IEEE Centro-Norte Brasil Blockchain Group. His professional experience includes research projects with Dell Computers, HP, IBM, Cisco, and Siemens. He has coordinated research, development, and technology transfer projects with the Brazilian Ministries of Planning, Economy, and Justice, as well as with the Institutional Security Office of the Presidency of Brazil, the Administrative Council for Economic Defense, the General Attorney of the Union, and the Brazilian Union Public Defender. He has received research grants from the Brazilian research and innovation agencies, i.e., CNPq, CAPES, FINEP, RNP, and FAPDF. He has developed research in cyber, information and network security, distributed data services and machine learning for intrusion and fraud detection, as well as signal processing, energy harvesting and security at the physical layer.

ROBSON DE OLIVEIRA ALBUQUERQUE received the M.B.A. degree in computer networks from the Educational Union of Brasília, Brazil, in 2001, graduated in computer science from the Catholic University of Brasília, in 1999, the DEA degree from the University Complutense of Madrid, in 2007, the master's degree in electrical engineering from the University of Brasília, in 2003, the Ph.D. degree in information systems from the University Complutense of Madrid, Spain, in 2016, and the Ph.D. degree in electrical engineering from the University of Brasília, in 2008. He is currently a Researcher with the University of Brasília and a member of the GASS Research Group, University Complutense of Madrid. His fields of interest and research include cyber, network and information security, as well as distributed systems and computer networks.