# Security Monitoring SLA Verification in Clouds: the Case of Data Integrity

Amir Teshome Wonjiga, Louis Rilling, Christine Morin


# Security Monitoring SLA Verification in Clouds: the Case of Data Integrity

**Amir Teshome, Louis Rilling, Christine Morin**

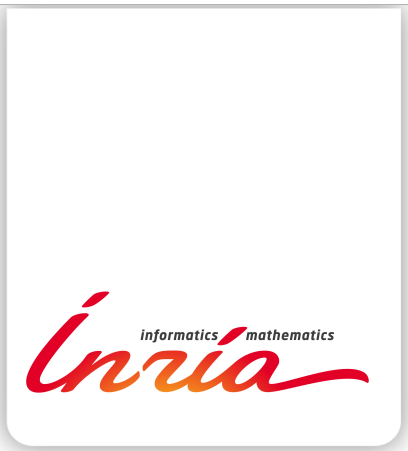
**RESEARCH REPORT**

**N° 9267**

March 2019

Project-Team Myriads

ISSN 0249-6399 ISRN INRIA/RR--9267--FR+ENG




# Security Monitoring SLA Verification in Clouds: the Case of Data Integrity

Amir Teshome*, Louis Rilling†, Christine Morin*

Project-Team Myriads

Research Report n° 9267 — March 2019 — 29 pages

**Abstract:** The cloud computing business model introduced a new paradigm in terms of ownership of a system. Before the cloud, a user acquires physical infrastructure and uses it by installing and configuring according to her/his needs. In that scenario, the full system is owned by a single entity. In the cloud, when the user outsources a service for a cloud provider the user owns some part of the system while the provider owns the remaining part. Thus, ownership in the cloud is divided between different entities. Clients hosting their information system need to trust and rely on what the providers claim. At the same time providers try to give assurance for some aspects of the provided service (e.g. availability) through service level agreements (SLAs). We aim at extending SLAs to include security monitoring terms. In a previous study [1] we proposed an SLA verification method for security monitoring SLAs describing the performance on an NIDS.
In this paper we consider an SLA guaranteeing the integrity of tenants' data stored in the cloud. The tenant outsources data storage service to a Storage as a Service cloud provider. In such a system the data is owned by the tenant while the provider owns the infrastructure. We consider an SLA offered by the provider to guarantee the integrity of tenants' data. In this paper, we propose a verification method, i.e. an integrity checking method, which is based on a distributed ledger. Specifically, our proposed method allows both providers and tenants to perform integrity checking without one party relying on the other. The method uses a *blockchain* as a distributed ledger to store evidences of data integrity. Assuming the ledger as a secure, trusted source of information, the evidence can be used to resolve conflicts between providers and tenants. In addition, we present a prototype implementation and an experimental evaluation to show the feasibility of our verification method and to measure the time overhead introduced.

**Key-words:** clouds, security monitoring, SLA, data integrity, remote integrity checking, blockchain, Hyperledger Fabric

* Univ Rennes, Inria, CNRS, IRISA
† DGA-MI

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

# SLAs pour la supervision de sécurité dans les clouds : le cas de l'intégrité des données

**Résumé :** Le modèle d'affaires du cloud computing a introduit un nouveau paradigme en termes de qui est propriétaire d'un système. Avant le cloud, un utilisateur acquiert une infrastructure physique et l'utilise en l'installant et en la configurant selon ses besoins. Dans ce scénario, le système complet appartient à une seule entité. Dans le cloud, lorsque l'utilisateur sous-traite un service chez un fournisseur de services dans le nuage, l'utilisateur possède une partie du système tandis que le fournisseur possède la partie restante. Ainsi, la propriété dans le nuage est divisée entre différentes entités. Les clients hébergeant leur système d'information doivent se fier aux fournisseurs. En même temps, les fournisseurs essaient de donner des éléments d'assurance sur certains aspects du service fourni (ex : la disponibilité) par le biais de contrats sur les niveaux de service (*Service-Level Agreement* ou SLA). Notre objectif est d'étendre les SLAs afin d'y inclure des aspects de supervision de la sécurité. Dans une étude précédente [1] nous avons proposé une méthode de vérification des SLA pour la supervision de la sécurité dans le cadre de SLAs décrivant la performance d'un système de détection d'intrusion réseau.

Dans cet article, nous considérons un SLA garantissant l'intégrité des données stockées par les utilisateurs dans le cloud. L'utilisateur sous-traite le service de stockage de données à un fournisseur de cloud Storage as a Service. Dans un tel système, les données sont la propriété de l'utilisateur tandis que le fournisseur est propriétaire de l'infrastructure. Nous considérons un SLA offert par le fournisseur pour garantir l'intégrité des données des utilisateurs. Dans le présent document, nous proposons une méthode de vérification, c'est-à-dire une méthode de contrôle de l'intégrité, qui est fondée sur un registre distribué. Plus précisément, la méthode que nous proposons permet aux fournisseurs et aux utilisateurs d'effectuer des vérifications d'intégrité sans qu'une partie ne se fie à l'autre. La méthode utilise une *blockchain* comme registre distribué pour stocker les preuves de l'intégrité des données. En supposant que le registre est une source d'information sécurisée et de confiance, la preuve peut être utilisée pour résoudre les conflits entre les fournisseurs et les utilisateurs. De plus, nous présentons un prototype de mise en œuvre et une évaluation expérimentale afin de démontrer la faisabilité de notre méthode de vérification et de mesurer les coûts introduits sur la performance.

**Mots-clés :** clouds, supervision de la sécurité, SLA, intégrité des données, vérification d'intégrité à distance, blockchain, Hyperledger Fabric

# 1 Introduction

The work presented in this paper was done during a six-month internship at Laurence Berkeley National Laboratory (LBNL). The internship focused on enhancing data integrity checking methods in a scientific environment. Specifically, we studied the advantages of using logical or physical secure components in the process of integrity checking for scientific data. In this paper, we propose to adapt and use the internship work for the case of user-centric security monitoring service-level agreements in clouds.

The cloud computing paradigm introduced a new type of business model where the service provider owns the underlying physical infrastructure and the user (tenant) owns the system running on top of the infrastructure. Tenants do not have a full control on the physical information system, thus they need to trust and rely on what the providers claim, including the security aspect of the infrastructure. At the same time providers try to give assurance for some aspects of the provided service (e.g. availability) through service level agreements (SLAs). An SLA is a negotiated contract between the service provider and a tenant. An SLA has different components including a quantitative description of the targeted quality of service, called service level objective (SLO), using Key Performance Indicators (KPI). In our work, we aim at extending SLAs to include security monitoring terms.

*Security monitoring* is the collection, analysis, and escalation of indications and warnings to detect and respond to intrusions [2]. The goal of collecting and analyzing events and generating indicators is to *detect* and prevent intrusions. In addition, when prevention eventually fails, the goal is to *respond* to incidents as quickly as possible and understand how the intruder achieved its attack and what damage it made. Different types of security monitoring devices and techniques are used for different components.

In a previous study [1], we saw that *availability* is the most common security property covered by current SLAs. Other aspects of security are the targets of few recent research works but they are not included in any of existing SLAs. Moreover, a few of the research initiatives follow a user-centric approach [3]. However these user-centric approaches do not address (i) how users express security requirements, (ii) how to offer quantifiable service-level objectives (SLOs) regarding security properties, and (iii) how users can verify an SLO in their environment. In the context of clouds, where the provided services are tailored for each tenant, SLAs should be user-centric. Especially when addressing the security of tenants' environments, each tenant has its own security requirements and very often they require different implementation mechanisms.

User-centric refers to users having the utmost attainable control, choices or flexibility on a system. Providing user-centric security services means allowing tenants to have the utmost attainable control, choices, and flexibility on types of vulnerabilities to be monitored, over actions to be taken in cases of incident and compensation if damages happen as a result of security breaches. All these can be addressed by allowing tenants to participate in the process of securing their environment.

In our previous study [1], we have seen one way of implementing user-centric security monitoring in the cloud. We used service-level agreements (SLAs) to include tenants in the processes of security monitoring. Specifically, allowing users to define which vulnerabilities to be monitored and check the validity of an SLO at any time (or according to a schedule defined in the agreement). In the process, we mentioned the need for cooperation between providers and tenants. In this section, we first recall the dependency between tenants and providers then we describe a proposed mechanism to reduce such dependency and finally we present the SLA that we consider in this paper.

## 1.1 Dependency Between Tenants and Providers

Going back to the initial problem, the need to have security monitoring SLAs for clouds is because of the nature of its operation. In the cloud business model tenants outsource their information system and providers are in charge of monitoring the physical infrastructure including its security monitoring aspect. This scenario creates a trust issue between tenants and the provider, thus the need for an agreement.

SLAs contain objectives (SLOs) and SLOs must be verified for their fulfillment and any violation results in a penalty for the violating party. In the best case scenario, once an agreement is signed any participant should have the option to perform a test on the satisfaction of an objective independently of the other party(ies). Moreover, in case of violation, one party should be able to prove the violation for the others. Currently detecting SLO violation is left as a responsibility for tenants. Even when tenants discover any violation, penalties do not apply automatically. Service providers perform checks on their side and the penalty is applied only if the provider discovers the violation. For example, Amazon SLA [4] describes the procedure to report a violation and states *"if the Monthly Uptime Percentage of such request is confirmed by us and is less than the Service Commitment, then we will issue the Service Credit to you ..."*.

Our SLA verification method presented in [1] describes the need for cooperation between tenants and providers in order to perform end-to-end security monitoring. Without such cooperation doing verification is difficult if not impossible. Tenants need to disclose the service(s) they are running and providers need to give an untampered output of security monitoring devices. In the context of SLAs, such dependency creates a conflict of interest and it requires trust from the other party, which goes back to the initial trust issue. Thus, we need a mechanism to reduce (remove if possible) such dependency in SLA verification.

## 1.2 Trusted Component in the SLA Verification Process

To remove such dependency on one party we need either a trusted third party or a trusted system where participants in the agreement get information and make decisions accordingly. Recently we have witnessed an increase in applications of distributed systems technology to make a secure system state change between untrusted parties without using any third party. These technologies, referred to as *blockchains*, are widely adopted for their property of *tamper-proof evidence*. Blockchains are used as the core technology in digital currency (cryptocurrency) applications and a very wide range of applications ranging from the Internet of Things (IoT) to health, identity and security are being developed.

In this paper, we provide a user-centric security monitoring SLA verification approach for clouds which relies on the blockchain technology. Users participate in the SLA life-cycle process. Specifically, they can perform verification at any given time. Besides, users directly participate in keeping and maintaining attestation data secure, data which will be used for verification. Users participation is possible as a result of a property of blockchains; by design, blockchains are distributed systems; i.e multiple nodes, ideally geographically distributed, form a network to build the blockchain. Tenants can be part of this network and participate in the SLA life-cycle process. In the method presented in this paper tenants participate in the network with the provider to keep verification data secure, this allows trusting the system in general rather than a single entity. Depending on the algorithm used in the blockchain operation process, few numbers of participants can be malicious and the system can still be trusted.

## 1.3 Considered SLA

SLAs in this paper are different from the ones presented in our previous study [1]. Previously defined SLAs were guaranteeing the performance of security monitoring devices, specifically NIDSs. In this paper, we define SLAs guaranteeing security properties of a given data stored in the cloud, specifically the *integrity* property of the data. Our goal in this paper is to show that having a trusted component in the SLA life-cycle helps to facilitate and improve the trust level of different phases in the life-cycle of SLA. Even if we are using data integrity to show the use of trusted components in the SLA life-cycle, the method presented in this paper can be adapted for other types of SLAs, including security monitoring SLAs as presented in [1]. Section 8.3 describes using secure hardware components for security monitoring SLAs.

The problem discussed in [1] for IaaS clouds mostly applies to data in the cloud. There is a lack of security SLA for data stored in a cloud. This lack of security support has been a significant difficulty for the adoption of cloud services mainly for enterprises and cautious consumers. For example in Dropbox [5], a file hosting service, user metadata is stored in the company's data centers, while the actual files reside on Amazon's S3 storage service. Such a relationship between companies requires an agreement which covers the security aspect of the data. The same way as in IaaS clouds, we can address the security issues in outsourced data storage through SLAs.

Data integrity failure is a common issue in data storage systems [6, 7, 8]. There are different solutions to protect data from corruption and to recover from corruption after its occurrence. Additionally, in some fields of research and development, it is mandatory to keep data integrity and verify this property for others. For example, almost all regulatory agencies controlling medical drugs and health care products publish data integrity guidelines. Such government entities require implementing these guidelines in testing, manufacturing, packaging, distribution, and monitoring of drugs to review the quality, safety, and efficiency of the products.

There are different reports of data integrity failure in the real world IT production environments. In 2009 Facebook temporarily lost more than 10% of photos in hard drive failures [7]. Amazon S3 suffered from a data corruption issue [8] caused by a load balancer which was corrupting single bytes in the byte stream. As a result, uploaded objects processed through that malicious load balancer did not match the MD5 hash values supplied by the user. In [6] the authors examined 138 data corruption incidents reported in the bug repositories of four Hadoop projects. The study presented conclusions on the causes and impacts of data corruption and listed limitations in detection and handling of data corruption mechanisms. The effect of data corruption is not only limited to data integrity. It may lead up to service unavailability. As presented in [9] corrupted iCloud data was a cause for iOS home screen crash.

The seriousness of the issue reaffirms the need to have SLAs guaranteeing the security aspect for data stored in the cloud. Such SLAs, in addition to other properties, would be meaningless without a verification mechanism. For existing SLA metrics, monitoring is performed by tenants or third-party companies, and service providers should confirm the violation. In order to minimize the trust issue between service providers and tenants, we need an open (non-secretive) process to do verification and to store and share the result without any bias. In this paper, we show a monitoring mechanism that can be used to check the correctness of data stored in the cloud without relying on the service provider.

As we have seen before, monitoring is a continuous verification process, and we provide a verification method for the integrity property of data hosted in a remote server without relying on the service provider. As in any security solution for clouds, our goal is to perform the verification with a limited performance impact, i.e., with minimum overhead in the production process. We start by describing the SLA life-cycle process for data in clouds and its difference with SLAs described in our previous study [1].

## 2 SLA Life-cycle for Monitoring Data Integrity

The general SLA life-cycle process also applies for SLAs guaranteeing data integrity, i.e., SLA life-cycle for data integrity also has three phases namely *SLA definition and negotiation, enforcement*, and *verification*. The SLA *definition* is a pre-negotiation phase where service providers draft SLA templates according to the services they provide. In the *negotiation* phase, tenants identify their requirements and perform assessments. Once the SLA is defined and agreed, the next step is to *enforce* the terms that are defined in the SLA. Enforcement means to implement what is written in the agreement into the actual infrastructure. After enforcement, the SLA should be monitored regularly (or with the frequency stated in the agreement) to *verify* its compliance with the SLOs promised in the SLA.

The contents of each phase of the SLA life-cycle in this paper is different from what was presented in the previous study [1] for security monitoring SLAs as the targeted properties are not the same. In this paper SLAs address security properties for data hosted in a cloud, specifically the integrity property. Other properties like confidentiality can be included in such type of SLAs. Section 8.3 describes how to incorporate the confidentiality property in the proposed method. In most existing cloud storage services (e.g., Amazon S3 [10] and DriveHQ [11]) availability is addressed in their SLA. Other properties are not addressed in SLAs. For example, Amazon S3 claims to have "extremely durable" storage with data stored redundantly across multiple devices and checks for corruption while data is at rest and in the network. However, Amazon does not guarantee these properties through an SLA.

The difficulties described in [1] related to quantitatively measuring security properties are still valid for the case of data integrity and confidentiality. It is difficult to measure integrity and confidentiality quantitatively. In our use case, tenants require checking the correctness of their data, i.e. either their data is corrupted or not. Hence, we assume the definition of SLA with an objective to keep the data uncorrupted as long as possible. Other properties like backup frequency and type of access control policies can be included in the SLA definition. Additional security properties like write-serializability (i.e. consistent among updates made by authorized users) and read freshness (read operation returns the latest update at any point) can also be integrated into SLAs.

SLA enforcement for data integrity can be achieved using different mechanisms. That is different techniques can be applied to keep the data uncorrupted, for example, replication and cryptographic methods. Data corruption can occur due to software bugs, design flaws, human errors, hardware failures, natural faults (bit flips), and malicious attacks. There exist different detection and correction mechanisms for various types of corruption sources.

Monitoring is used to check the satisfaction of SLA terms and detect any violation. Monitoring SLAs guaranteeing data integrity means verifying the correctness of data stored in the cloud. The verification can be performed either by a provider or tenant. In this paper, we want to show a mechanism to perform verification without relying on the provider. We use a distributed system, blockchain, in which as long as the majority of the participants are cooperating to keep the value, stored data can not be changed.

Note that this paper shows a verification process for data integrity, but the method could be easily adapted for other properties. The goal is to show the advantage of trusted components in user-centric security SLAs to reduce dependency on providers to verify SLOs. In Section 8.3 we discuss the possibilities in the direction of using other trusted components, like a secure hardware component for SLAs presented in [1].

In this paper, we first describe the problem that we address and give a background on the blockchain and types of blockchain. Then we describe the threat model assumed for the verifica-

tion process. We describe the verification process and a prototype implementation. Finally, we present an experimental evaluation and discussion on the proposed method.

# 3 Problem Description

This section presents the problems addressed in this paper. SLAs in this paper describe the integrity property of data hosted in a cloud. We focus on the third phase of the SLA life-cycle: SLA monitoring. Hence, most of the general problems from [1] (i.e. problems that are not specific to NIDSs) are also challenges for data integrity SLA monitoring.

One of the biggest challenges in doing verification is having different owners in different tiers or layers of the cloud. Verification in such an environment means checking the status of a service from a tier which does not belong to the verifier. An SLA guaranteeing data integrity is not different, the provider infrastructure holds the data and a tenant wants to check the integrity of that data.

The verification mechanism presented in our previous study requires cooperation. In this paper, we aim to reduce the need for cooperation between tenants and providers. The considered SLO is to keep data uncorrupted as long as possible. Our problem is similar to a well-known problem called remote data integrity checking, which enables a server to prove to an auditor the integrity of a stored file. However, in our case the verification is in two ways, i.e. tenants need to check data integrity in the cloud and providers need to check the correctness of SLO violation claim which can be submitted by tenants.

In summary, in this paper, we address the problem of SLA verification for data integrity which allows both the tenant and provider to perform checks without relying (or with a minimum dependency) on the other party.

# 4 Background and Related Works

The verification mechanism proposed in this work is based on the blockchain technology. In this section, we introduce blockchain, its various types, and features. This section also presents related works on data integrity monitoring tools.

## 4.1 Blockchain

The blockchain is a distributed linked list data structure, list of connected blocks, where every block contains a cryptographic hash of its predecessor block, hence forming a chain (see Figure 1) [12]. A valid change of a block requires changing every block after that modified block until the end of a chain. Additionally, the chain is stored in a *distributed manner*, i.e. there is no central location keeping all the blocks. Every valid block in the list is available in every participant node. Adding a new block (i.e. extending the list) requires an agreement between the participants. Participants use a consensus algorithm to decide what to add next in the chain.

Blockchains are distributed, and they are relying on consensus algorithms both of which makes the system trustworthy without trusting any specific participant. It is important to note that blockchains are not 'bulletproof', as the technology is emerging and getting lots of attention, different studies are being conducted to study its property from different aspects. In general blocks in a blockchain contain at least two parts:

- *Link to a previous block*, the hash value of the previous block will be included in the current block, and it serves as a link from the previous to the current block. If any byte is changed

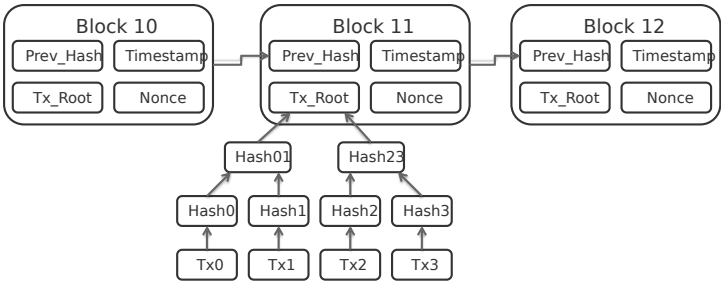



Figure 1: Simplified block chain representation

in the previous block the link would be broken, i.e. the hash value will not match with a reference in the current block hence, it requires updating all blocks after the modified block, which creates a new line of the chain.

- *Data stored in the current block* is the information stored in the block. The data can be anything depending on the application using the blockchain. Most well known digital currency implementations store information like timestamps, transactions, and values used in the consensus process. Figure 1 shows a simplified snippet of Bitcoin [12], a well known digital currency blockchain. The data section stores transactions and for an efficiency reason, they are stored in a *hash tree (Merkle tree)* data structure.

Following this, we describe types of blockchains, consensus algorithms used and existing implementations. Furthermore, we present in detail Hyperledger Fabric [13] which is the blockchain used in our work.

### 4.1.1 Types of Blockchain

Based on write permissions (permission to add blocks) blockchains are categorized into three categories [14]:

- *Public blockchain* is a type of blockchain where anyone from anywhere can join the network and participate in the consensus process to add blocks. Usually, participants have economic incentives to cooperate and when anyone wants to add a block, there is a simple verification process performed by other participants to check the correctness of the proposed block.
- *Private blockchain* is a type of blockchain where only one entity (e.g. the administrator) is allowed to add blocks. Some applications like database management may use such kind of data structure. Permission to read from the blockchain can be public, restricted or private as required.
- *Consortium blockchain* is a type of blockchain where only a pre-defined set of nodes are allowed to add blocks in the chain. It can be seen as a private blockchain having more than one authorized entity to add blocks. An example could be a group of financial institutions or a group of colleges and universities forming a consortium. As in the case of private blockchains, 'read' permissions can be set as required.

These categories could also be generalized into two groups, *permissioned and permissionless*. Consortium and Private blockchains are permissioned while Public blockchains are permissionless.

### 4.1.2 Consensus Algorithms

A blockchain is a distributed, peer-to-peer system and an agreement is needed to make any decision in the system. Consensus algorithms [15] refer to this process and in computer science, it is a well-known problem. A prominent class of such problem is the Byzantine Generals Problem [16]. It is a problem to reach an agreement in the network in the presence of potentially malicious participants. *Byzantine Fault Tolerance (BFT)* is the characteristic of an algorithm which defines a system that tolerates the class of failures that belong to the Byzantine Generals' Problem. Byzantine failures are considered the most general and most challenging class of failures to deal with. There are other types of failure models where non-malicious faults are tolerated, like crash fault tolerance.

A Byzantine failure implies no restrictions and makes no assumptions about the kind of behavior a node can have. If an algorithm can handle such a failure, it is assumed as robust. The goal of consensus algorithms in blockchains is to reach this level of fault tolerance. Moreover, in a blockchain, what is needed is not a Byzantine fault-tolerant SQL database. Instead, it is a Byzantine fault-tolerant distributed, append-only ledger, i.e. the duplicated data is not controlled by a single entity (multiple participants) and the data stored in the system is immutable.

In practice, there are different implementations of consensus algorithms. In the field of blockchains, we can generally classify these algorithms into two groups, namely *Byzantine-based* and *Proof-based* consensus algorithms.

- Byzantine-based consensus: is the traditional way of reaching an agreement between nodes in a network. The most well-known implementation in this category of algorithms is *Practical Byzantine Fault Tolerance (PBFT)* [17] which was published in 1999. After PBFT, several BFT algorithms were introduced to improve its robustness and performance. Few blockchain systems implement byzantine-based consensus. Hyperledger Fabric [13] is an example where PBFT is implemented, at least as a prototype, and used for consensus.

- Proof-based consensus: are generally called "proof of X" algorithms, where 'X' indicates evidence for having some required property. The node who poses this proof can be a leader and can propose a new block to be added. Then the block will be propagated and verified by other participants before adding it to the block list. This kind of algorithms can be seen as a probabilistic approach for Byzantine-based consensus. Examples of proof-based consensus algorithms include Proof of Work (PoW) [12], Proof of Stake (PoS) [18], Proof of Elapsed Time (PoET) [19].

  PoW is the first implementation of this category of algorithms, and it is used in major digital currency applications including Bitcoin [12] and Ethereum [20]. In the PoW consensus algorithm, to be a leader and propose a block, one must solve a cryptographic problem. The first one who gets the solution can propose a block, and other participants verify, add the block and start working on the next block. The problem to be solved is difficult, computationally intensive, and the difficulty level can be adjusted to control the rate of new block creation. Other "proof of X" algorithms also follow a similar logic but rather than having the computationally intensive problem they propose to use other parameters.

### 4.1.3 Blockchain Implementations

In early computer science studies the concept of state machine replication was used as a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with replicas [21]. The Blockchain technology addresses similar problems and the

first implementation, Bitcoin [12], showed the potential of its usage. After that, different varieties of implementations have been done. In general, there are three generations of blockchain implementations [13]:

- *First generation*: the early implementations of blockchains have few distinct properties. The ledger (blockchain) is used specifically for one application, mainly digital currency with a particular scripting language to perform operations. It uses a specific (fixed) consensus algorithm, operations are related with the currency used in the chain, and it is permissionless; hence anyone can join the network. An example of this first generation blockchain is Bitcoin [12].
- *Second generation*: this group of implementations improves upon the first generation. The second generation blockchains separate the logic and storage parts into different sections. The logic part, also known as a *smart contract*, is the only component allowed to do operations (read and write) in the storage component. The storage part, as the name indicates, is responsible for storing the actual data. This separation allowed using blockchains for applications other than digital currencies e.g. for identity management or certificate authority. As a result of this separation, the logic component can be programmed to perform *deterministic* operations. Domain-specific languages are used to enforce the deterministic property in the scripting language. This generation implementations uses a fixed consensus algorithm which forces taking the same threat model for all applications. Ethereum is a prominent example of this generation.
- *Third generation*: this group of implementations follows a different kind of architecture to remove the constraint of using deterministic languages to write smart contracts while sacrificing on being permissionless. It allows writing smart contracts using general purpose languages and consensus in the blockchain is modular, i.e. an application can plug and use any type of consensus algorithm depending on its need. Being permissioned means that there is a requirement to get permission from some entity which, some argue, makes a blockchain centralized.

It is important to note that this classification is not universally accepted; it is possible to find other classifications, like [22].

#### 4.1.4 Hyperledger Fabric (HLF)

Hyperledger Fabric or *Fabric* [13] for short, is an implementation of permissioned blockchains. It was initially developed by IBM and it is now hosted by Linux Foundation project called Hyperledger, hence the name *Hyperledger Fabric (HLF)*. HLF is permissioned by design, and it is developed focusing on business use cases where participants are expected to be known in the system. HLF introduces a different architecture than the first and second generations of blockchain implementations.

The data stored in blocks are in the form of transactions, which are executed on the current state of the system and create the next state. Before HLF, other implementations follow an *order-execute* architecture, i.e the first transactions are ordered, and a consensus is reached on that order. Then every participant node executes transactions in that order and updates the ledger accordingly. However, this architecture has drawbacks including:

- *Transactions must be written in deterministic code*: Since consensus is reached before execution, a non-deterministic code may create different results on different nodes, hence

breaking the consensus and creating more than one state (*fork*) for the blockchain. In previous generations of smart contracts, domain-specific languages (e.g. Solidity for Ethereum) are used to achieve a deterministic behavior.

- *Limited to sequential execution*: Since a consensus is reached before executing the transactions, every node executes transactions in a block sequentially. Sequential execution affects the throughput in the system limiting any parallelization.

HLF introduces a new type of architecture called *execute-order-validate*. *Fabric* tries to overcome previously described and other drawbacks from prior permissionless blockchains (e.g. hard-coded consensus and confidentiality) by introducing the highly modular *execute-order-validate* architecture.

In this model transactions are first executed (to be specific, transaction execution is simulated), all nodes are not required to do this step, and this will not affect the state of the ledger because it is a simulation on a local copy of the ledger. This feature gives the opportunity for confidentiality and parallel execution. Then transactions are ordered using a consensus algorithm, and any algorithm can be used for this process. This feature gives the opportunity for a modular blockchain design which can use different consensus algorithms depending on the application threat model. Finally, a predefined policy is used to validate transactions. Such a policy is defined per application level which allows having a flexible trust model. Furthermore, since ordering is performed after execution, the executions are not expected to be deterministic, the validation phase guarantees freshness i.e conflicting transactions will be dropped.

*Fabric* provides the confidentiality feature by dividing the network into different *channels*. An organization will get permission to perform some action (according to its role) only if it is a member of that channel. This is an essential feature in Hyperledger Fabric, and it is useful in cases where participants might be competitors or just when a subset of participants are not part of a project. The organization deploying an application in a channel can decide who can participate in the process of that application.

As *HLF* is a permissioned blockchain, there is *Membership Service Provider (MSP)*, which is responsible for maintaining thidentities of participants in the network. MSP is an abstraction where the back-end could be different membership standards and architectures. The interaction between entities in the network is cryptographically signed with identities provided by MSP.

As a design principle, HLF follows a highly modular design pattern. It provides different components of the blockchain network as Linux containers (containerized) service. Each organization contains different services including peers, orderer, MSP and so on.

In the next section, we present related works focusing on data integrity checking tools and practices.

## 4.2 Related Works

This section presents the related works in two parts, first related works on remote data integrity checking and second, the usage of blockchains for data integrity.

### 4.2.1 Remote data integrity checking

The concept of remote data integrity checking has been around in relation to distributed systems [23]. While the adoption of cloud increases, remote integrity checking becomes more critical because more and more sensitive data is being stored in the cloud. There have been different works on checking properties of remotely stored data, including integrity. One way to check data

integrity is by completely downloading the data and compare it with the original version but this method contradicts the basic idea of the cloud, and it is impractical. Usually, the size of data stored in the cloud is huge, and users may not keep a copy of the entire data. As a result, almost all remote integrity checking works rely on cryptography methods without the need to completely download the data.

Due to its necessity, remote data integrity checking has attracted extensive research interest [24, 25, 26, 27]. Generally, we can categorize these works into two main groups, integrity checking *with and without a third party auditor (TPA)*. TPA is an entity who has expertise and capabilities for performing an integrity check and convincing both the client and the provider. It is considered to be a trusted party. TPA performs the primary role of data integrity check; it performs activities like generating a hash value for blocks received from the cloud server and compares signatures to verify whether the data stored in the cloud has been tampered with or not.

For example, TPA is used in [26]. The paper presents a remote data possession checking protocol with the support of public verifiability (i.e. anyone can perform the verification). It defines four functions namely *KeyGen, SigGen, GenProof, VerifyProof*: *KeyGen* run by users to generate a key, *SigGen* used by users to generate verification metadata, *GenProof* run by service providers to generate a proof of data storage correctness and *VerifyProof* used by TPA to verify the proof from service providers. The protocol executes in two phases, namely the *setup* and *audit* phases. In the setup phase, users generate a key using the *KeyGen* function and generate metadata for data files to be sent to the cloud. Then the user sends the data to the cloud and publishes the metadata to TPA. After that, a user can delete the local copies of the uploaded data.

The second phase is an audit phase. Upon request by a user, TPA starts the verification process. TPA formulates and sends a challenge to the provider and waits for a response. The service provider upon receiving the challenge, runs *GenProof* to generate a proof showing the data is correctly stored and sends back the proof to TPA. The TPA runs *VerifyProof* using the returned value and checks the result with the original metadata. This way TPA is used to check data integrity in the cloud.

Other works like [27, 24] follow a similar pattern, but they use different cryptographic methods to achieve properties other than integrity, e.g data dynamics, privacy against verifiers, and proof on multiple providers. The protocol presented in [27] supports data dynamics, and the work presented in [24] supports privacy against third-party verifiers. In some works the TPA is optional and the user can also be an auditor. However, in other works, the TPA is required and acts as a trusted third party (TTP), It is used to resolve disputes between the provider and tenant.

There are also some works which do not use third parties in the process. The protocol presented in [25] does not rely on a TPA. It follows a similar procedure as described above using six functions to operate, except the cryptographic procedures are different. Such protocols cannot be used for SLA verification because the process is only one way, i.e. only one party is performing the integrity check. For our use case both parties need to perform the verification. CloudProof [28] presents a storage system which provides proofs of a violation; hence neither providers nor tenants can bring a false claim of violation. Our work can be used to extend such a system in order to exploit the distributed nature of blockchains and become even more independent from other entities, like certificate authorities.

In our work, we want to reduce dependency between participants and possibly remove TTPs as it requires as much trust as providers. We thus propose to rely on a secure and distributed ledger, blockchain.

### 4.2.2 Blockchains on data integrity

About the usage of blockchains, there are different applications for data integrity services in various scenarios including IoT, database systems, data provenance tools and so on. In addition, new distributed storage systems based on blockchains (e.g Storj [29]), which include the data integrity feature by design are being developed. In [30] the authors proposed a blockchain-based data integrity framework for IoT data stored in the semi-trusted cloud. The framework incorporates data generators and data consumers and enables consumers to perform integrity verification. The framework follows a typical blockchain applications strategy, i.e it does not store all the data in the blockchain, it stores only the hash of the data and the actual data is stored in a cloud storage service.

The authors in [31] presented a blockchain-based database with strong integrity guarantees. The system uses two layers of blockchains, the first layer with a lightweight distributed consensus protocol that assures low latency and high throughput. The second layer is designed with a strong consensus to guarantee better integrity by using PoW-based algorithm. There are new upcoming companies (e.g Chainpoint [32]) offering to anchor users data to existing blockchains, which helps to verify the integrity and existence of data without relying on a trusted third party.

To use the presented studies in our work, as they are proposed they do not describe the ability to verify integrity by other parties (other than the data owner). This feature is mainly a result of the considered threat model. In other words, using these works for our use case would allow only one party to perform the verification. Having a verification method that can be used by both parties is a requirement for our verification process. In that sense, we are addressing a different kind of problem than most blockchain-based applications.

In our work, we want to show the use of a trusted component, like blockchain, in the SLA verification process in order to reduce the dependency between tenants and providers and enable independent verification for both parties. In the next section we describe the assumed threat model.

# 5 Threat Model

In this study, we assume that cloud service providers are semi-trusted. In a sense, a provider may return wrong data or wrong result of computation for a request from a tenant because an attacker altered the data or of some other error. However, providers are not actively trying to alter tenants' data. In addition, providers never lie when claiming that they have not accepted to store some data. They are working towards maximizing profit and errors on the stored data are unintentional. Providers also have economic incentive related to an SLA. Hence, they may lie in order not to violate an SLO. Tenants store their data in the cloud and do not keep a local copy of the whole data. Tenants may have some portion of the data and store hashes of the complete data on the ledger (blockchain) to be used later for verification.

Tenants may falsely claim a reward for a data integrity breach. Hence, providers need to do verification by themselves. However, tenants never lie when claiming that providers accepted to store some data. A tenant is interacting with a single provider, and the provider replies to requests about the data. Providers may store the data in one location or divide it into different chunks for security or any other reasons. We assume the blockchain network is not compromised; at least the minimum number of required participants are honest and are not controlled by the provider. In addition, we assume there is a secure communication network between the three entities, tenant, provider, and the ledger. Notably, the integrity of a message is respected, i.e.

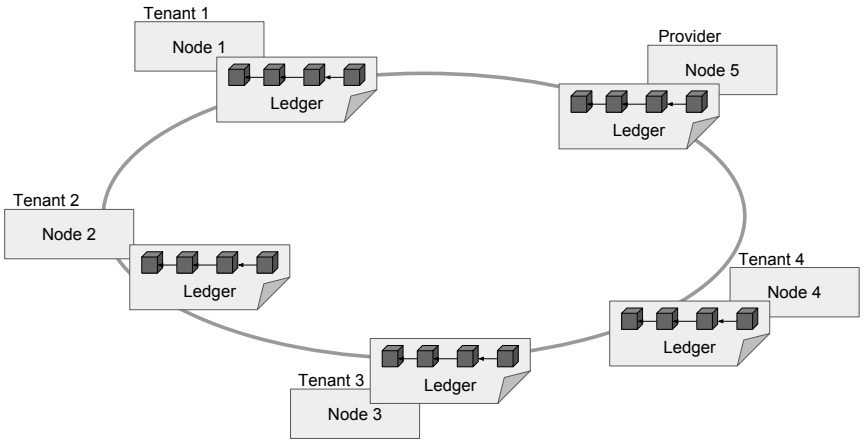



Figure 2: Consortium blockchain network formed by tenants and a provider

there is no man-in-the-middle which is actively altering the communication between the three entities.

# 6 Data Integrity Checking Process

In this section, we describe our proposed method to check integrity using a trusted, secure ledger and without relying on a third party. This method is used to perform SLA verification. Using the proposed method, a tenant can check the integrity of the data stored in the cloud, and a provider can check the correctness of SLO violation claims without relying on the other party. Our method guarantees to link the provider and the tenant with the same piece of evidence for data integrity.

As described in the previous sections, we use a blockchain to remove the trusted third party. This secure ledger is used to store a piece of evidence for attesting the correctness of outsourced data. Providers and tenants form the blockchain network. In a user-centric security monitoring context, using such a technique gives two main advantages. First, it adds trust, transparency, and security to existing integrity monitoring techniques without other parties and second users are directly participating in the process of securing their outsourced information system. Allowing users to participate in the process is the core of user-centric services and systems.

In general, given an SLA guaranteeing the integrity of users data, the verification process follows two main procedures, namely the *setup* and *verification* phases.

- *Setup phase* is where a tenant prepares *tags* for the data to be used later in the verification phase and it is performed before uploading a file to the cloud. *Tags* are proofs containing different components including the hash of the data. The tags are used by a tenant to perform verification without the need to trust either the provider or any third party entity. Section 6.2 describes *tags* in more detail. After this process, a tenant sends the data plus hash of the data to a provider. The hash value is used by service providers to check the correctness of the initial data upload. After successfully uploading the data, both the tenant and the provider publish the hash value of the data to the ledger.

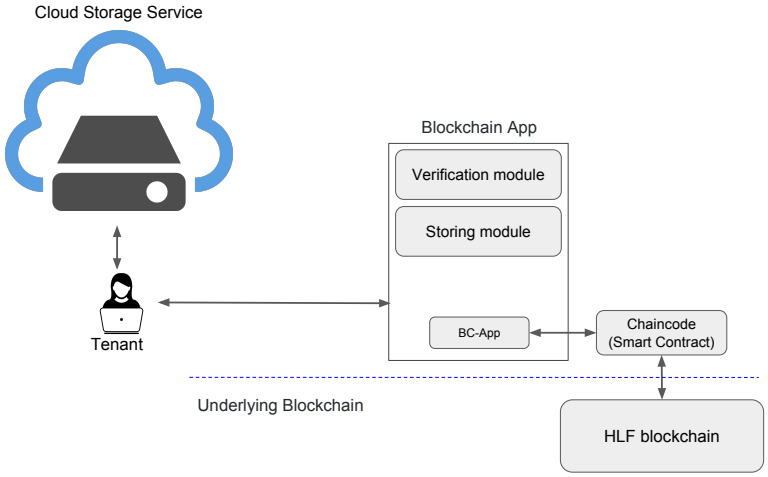



Figure 3: Data integrity verification architecture

- *Verification phase* is the procedure after the data is uploaded correctly to the cloud and the hash value published in the ledger. A tenant sends a request for providers to perform integrity checking and receives the hash value performed over the current state of stored data. Using the tags generated in the previous phase a tenant can confirm the correctness of the stored data. If the data is not correct, a tenant can claim an SLO violation using the proof stored in the ledger and proceed with the next procedure as stated in the SLA.

Section 6.2 presents a more detailed description of each phase. While the blockchain is serving as secure storage for hash values that are used to prove integrity, the providers' infrastructure stores the actual data. In the next subsections first, we present the architecture of the integrity checking platform, and second, we describe in detail the verification process.

## 6.1 Architecture

In this section, we describe the architecture and components of the proposed integrity checking platform. Our data integrity verification platform contains five main components (see Figure 3). These are the *underlying blockchain, smart contract (chaincode), blockchain app, a service provider (cloud) and a user (tenant).*

- *Underlying blockchain* is the blockchain network which stores pieces of evidence (hash values) of data that are used as an anchor for integrity checking. This component is assumed to be a trusted and secure storage system. The blockchain network is formed by the tenants and providers. It is a consortium blockchain, i.e. a private blockchain having more than one authorized entity to add blocks (see Figure 2). As described above every participant in a blockchain network holds all valid blocks and updates the list according to the consensus algorithm used. Our SLA verification method is not dependent on any specific blockchain implementation, i.e. it can be used with any type of consortium blockchain implementation. The number of required participants in the network depends on the type of blockchain implementation used and the resiliency model requirement (e.g. using HLF with a crash fault tolerant consensus algorithm, to tolerate $n$ number of crashes it requires $2n+1$ nodes). Section 7 presents the details of our implementation.

- *Smart contract (chaincode)* as described in Section 4.1.3, is the logical component of blockchains and can be programmed to perform different tasks. It is the only component which directly interacts (perform read and write operations) with the underlying blockchain. Our smart contract consists of the following functions: *initLedger()* called only once to initialize the blockchain. *addData(data)* used to add 'data' to the ledger, *queryData(data)* to check if 'data' is in the ledger and *Invoke(f, param)* used by external applications to call a function from the chaincode.

  The smart contract, which performs operations on the blockchain, avoids any duplicate entry in the blockchain, i.e when receiving a write operation from a tenant or provider, the smart contract first checks if the same data exists in the blockchain. If it found the same data it returns the ID of the existing block. Otherwise it will add the requested data. As a result, even if both tenants and providers make a separate request using the same data, the data will be added only once, and both the tenant and provider will hold the same block ID.

- *Blockchain application* or client application is a module which acts on behalf of a user, i.e. the entity who wants to call functions from the smart contract. The caller can be either tenants or the provider to store or retrieve a piece of evidence for a data block. Our application contains the *storage module* used to store evidence in the ledger and the *verification module* to retrieve evidence. These modules call the *addData()* and *queryData()* functions respectively from the smart contract.

- *Service provider (cloud)* is an entity providing the storage service. The provider offer SLAs to guarantee data integrity. As described in Section 5 providers are not malicious, and they can respond to requests (challenges) from a tenant or client application. They can also perform verification in order to check the correctness of an SLO breach claim from tenants.

- *Users (tenants)* are owners of the data stored in the cloud and they sign an agreement with the cloud provider. Users add evidence of data integrity into the ledger and they perform verification of data integrity to check if the SLO is still valid.

## 6.2 Integrity Checking Process

The process of checking integrity contains two phases, the *setup* and *verification* phases. In the setup phase, the tenant generates and stores required information for later verification. In the verification phase, the actual checking is performed by using evidences generated in the previous phase. Figure 4 shows the two phases of the integrity checking process. We also provide algorithms (Algorithm 1 and 2) describing both phases. The numbers in the figure correspond to the line numbers in the algorithms. The numbers with the suffix 'p' in the figure represent the tasks performed by the provider.

### 6.2.1 Setup phase

Figure 4 (A) shows the setup phase and Algorithm 1 shows the procedure in the setup phase (the line numbers in the algorithm represent the arrow numbers in the figure). In this section we present a step by step explanation of the process:

- The tenant (owner of the data) performs an operation on the data to produce evidence (tags) that will be used later for verification. In practice, this procedure is hashing the data using a secure hash algorithm e.g. SHA-1, SHA-2, xxHash... A hash function maps

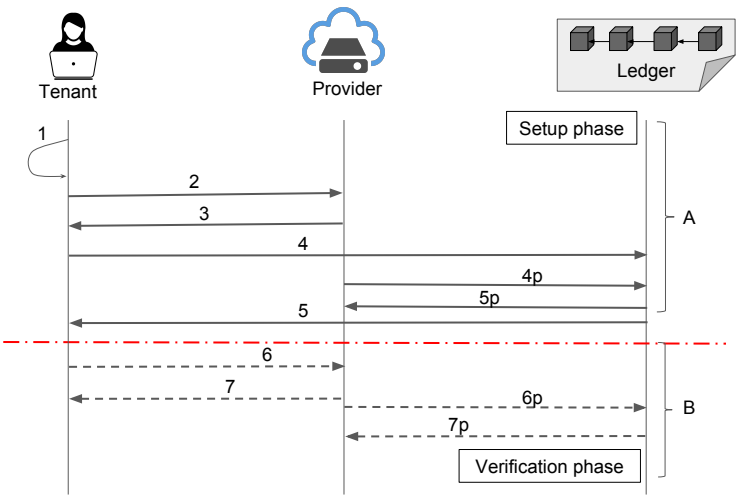



Figure 4: Data integrity verification process, (A) setup phase and (B) verification phase

input data of arbitrary size to a byte string of a fixed size. Depending on the security guarantees, there are two main categories of hash algorithms, namely *cryptographic* and *non-cryptographic* hash algorithms. The cryptographic hash functions provide strong security guarantees like collision resistant, one-way function, pre-image attack resistant etc. On the other hand, non-cryptographic hash functions provide weaker guarantees in exchange for performance improvements. They just try to avoid collisions for non malicious input. Since we consider non-malicious inputs, a non-cryptographic hash algorithm is enough for our use case. Using such a function the tenant generates the required tag values.

The generated tags include three parts, first hashing the data, *H(D)*, second generating $n$ random strings called *nonce*, $(R_1, R_2 ... R_n)$, and third hashing the data concatenated with every random string, $(H(D + R_1),\ H(D + R_2)\ ...\ H(D + R_n))$. The number of random strings are determined by the length of the SLA validity period and the frequency of verification. For example, if an SLA is valid for five years and verification is performed once per day, the tenant will generate 1825 (5 * 365) different strings and performs the hash of data plus a random string, for every value. The hashes and generated random values, i.e $(H(D+R_1),\ H(D+R_2)\ ...\ H(D+R_n))$ and $(R_1, R_2 ... R_n)$, should not be shared with other parties, specifically with the provider. These nonce values are used to force providers into performing a fresh hash computation. Even if the method requires $(n + 1)$ hash operations, it is practical in the SLA verification context. It is feasible because of two major reasons (i) the SLA has a validity period and (ii) data integrity is not a property which is checked very often like availability. Thus, the variable '$n$' is bounded.

- Tenant uploads the data with the hash value, i.e. (D, H(D)) to the cloud storage. Upon receiving the data, the service provider runs the same hash function over the data and compares the result with the provided hash value. If the value matches the provider will send a confirmation; otherwise, the provider assumes there is an error in the data transfer and notifies the tenant. In the case of such an error message, the tenant should repeat the process.

In practice this process is not novel, Amazon S3 [10] command line tool offers a similar option. The command *'s3api put-object'* takes *'--content-md5 and --metadata'* arguments and Amazon uses this information to perform integrity checking. Amazon confirms the correctness by returning an *'Etag'* and stores the hash value with the data. The hash value can be retrieved at anytime to check the integrity of stored data. This step is done only once unless the data is changed or updated. In that case, the hash of the new data should be computed.

- Once the upload is confirmed the tenant publishes the hash value, i.e $H(D)$ in the ledger. Publishing a hash value can be achieved by using the storage module from the client application. Providers also publish $H(D)$ to the ledger. However, as described in the previous section, the smart contract prevents duplicate entries in the ledger; thus the second *addData()* operation returns the block ID of the previously added data. This way both the provider and tenant have the same block ID.

**Algorithm 1:** Setup phase

**Input**: Data '$D$', Hash function '*H()*'
**Result**: $n$ Random strings $(R_1, R_2...R_n)$, Hash of '$D$' *H(D)* and Hash of '$D$' with random strings $(H(D+R_1),\ H(D+R_2)\ ...\ H(D+R_n))$

`/* these random strings are used later for verification. Both the random strings and (`$H(D+R_1)$ `...) should be kept private with tenants */`

1 Generate $n$ random strings, compute $H(D)$ and $(H(D+R_1),\ H(D+R_2)\ ...\ H(D+R_n))$ ;
2 Upload the data, i.e send $(D, H(D))$ to the service provider ;
3 The provider computes a fresh hash of '$D$', $H(D)'$ and compares it with the received one;
**if** $H(D)' == H(D)$ **then**
  return 'success' ;
**else**
  return 'error' ;
**end**
4 **if** *the previous step is successful* **then**
  `/* Both tenants and service providers execute the following function */`
  addData($H(D)$);
  `/* the smart contract adds` $H(D)$ `only once and subsequent addData(`$H(D)$`) requests return the blockID of existing` $H(D)$ `*/`
**else**
  return to step two and re-upload the data;
**end**
5 Get blockID, the blockID indicates where $H(D)$ is stored in the ledger

### 6.2.2 Verification phase

At this stage, the hash value of the data is published, and both the tenant and provider have a block ID referring to an address where the hash value is stored. It should be noted that random nonce values, $(R_1, R_2...R_n)$, and hash of the data with these values, $(H(D+R_1),\ H(D+R_2)\ ...\ H(D+R_n))$, are stored privately by tenants.

To perform verification, a tenant can challenge a provider to compute a hash value over the current state of the stored data. Figure 4 (B) shows the verification phase and Algorithm 2 shows the verification process. The steps to verify the integrity of the data are:

- A tenant selects one $R_i$ and sends it to the provider with the file name to be checked. This value of $R_i$ is then removed from the set of random numbers, i.e. it will be used only once.
- The provider computes the hash of the data concatenated with the nonce value ($R_i$) and returns the result to the tenant. The tenant compares the return value with the locally stored value and concludes about the integrity of the data stored in the cloud.

  In the event of a discrepancy between these two values, a tenant can claim for an SLO violation and use values from the ledger as evidence. Our integrity verification process guarantees that the tenant and the service provider will hold the same block ID value, i.e. they both refer to the same value in the ledger. It is important to note that holding the same pointer to a secured data storage location does not automatically resolve a conflict between tenants and providers. However, it can help in the process to resolve a disagreement between the two parties. One way of such usage can be in the process of legal action.

Values written in the ledger are immutable, i.e. it is secured by duplication and the consensus algorithm; hence, the ledger is serving as a secure and trusted anchor for both tenants and providers. Using a different nonce value for every verification, the tenant forces the provider to compute fresh hash values.

Service providers can check the integrity of stored data, by hashing the data over its current state and comparing the result with the one stored in the ledger. This checking process is especially helpful when there is a complaint from tenants, and a provider wants to check the validity of such claims.

**Algorithm 2:** Verification phase

**Input**: $n$ Random strings ($R_1, R_2 ... R_n$) and Hash of '$D$' with random strings ($H(D + R_1),\ H(D + R_2)$ ... $H(D + R_n)$)

**Result**: *true* (no data corruption) or *false* (there is data corruption)

Select one $R_i$ from the set of strings and send it to the provider. Remove $R_i$ from the set of strings in order not to use it again ;

`// note that the provider cannot cheat because` $R_i$ `is different on every request`

Compute $H(D + R_i)$ and return the *result* ;

**if** *result == Previously computed* $H(D + R_i)$ **then**

  `// no data corruption`

  return 'true' ;

**else**

  `// there is data corruption`

  return 'false';

**end**

# 7 Implementation

We have implemented a prototype of the proposed data integrity SLA verification tool. In this section, we present the implementation details by describing each component listed in Section 6.1.

We used *Hyperledger Fabric* (HLF) [13] as a back-end blockchain. For our use case, the unique features of HLF provide an advantage over other implementations. We describe three advantages

over other well-known implementations. HLF is permissioned, there is no native digital currency, and it allows writing a smart contract in any programming language.

As described in Section 4.1.4 HLF is one of the major implementations of permissioned blockchains. HLF allows us to create a consortium blockchain network with the tenants and provider as the participants. The fact that there is no native digital currency means that participants can perform operations on the ledger without paying additional payments for each operation. HLF allows using general purpose programming languages to write smart contracts, which helps to develop chaincode (smart contracts) easily and rapidly. Additional benefits of HLF include its modular consensus, and its active community both from academia and industry. It should be noted that other types of blockchain implementations that can be used to build a consortium blockchain network can be easily adapted for our use case.

All the other modules are implemented using python except the smart contract, which is written using the *Go* programing language. We used the *xxHash* algorithm to perform hash operations. *xxHash* is an alternative to the SHA hash algorithm families. It is a non-cryptographic hash algorithm with better speed than SHA families. The main criterion for the hash algorithm is to avoid collisions and since we consider non-malicious inputs a non-cryptographic hash algorithm is enough for our use case. Specifically, we use the xxhash.xxh64() method in our implementation. We perform incremental hashing based on a fixed block size rather than hashing the whole data at once; this helps to decrease the time needed to hash a given file.

The next section presents the evaluation performed to measure the performance of the proposed method.

# 8 Evaluation

In this section, we start by describing the setup used to perform experiments. We present the performance evaluation of the proposed method, specifically the time overhead as a result of the verification process and resources required to run the proposed verification process. We present an analysis and discussion on the performance and security of the proposed method.

## 8.1 Experimental Setup

To measure the performance of the proposed data integrity checking platform, we built a setup on the Grid5000 [33] testbed infrastructure. Three physical nodes are used to represent a user, a provider and a consortium blockchain built using Hyperledger Fabric (HLF). Each physical node contains two Intel Xeon E5-2630 v3 CPU, eight cores per CPU and 128 GB memory, all running Ubuntu version 14.04. We experiment with one tenant and one provider. However, the blockchain network contains ten participants. Our tenant and provider control one node each.

The entire blockchain network runs on a single physical node with containerized services, i.e. participants and related components are instantiated using Docker containers [34], and they communicate through virtual networks. In a real production environment participants reside in different physical nodes. The blockchain network contains ten participants, belonging to five different organizations (i.e. each organization has two participants in the network). As described in Section 4.1.4, HLF uses a structure of organizations and peers. For our experiment, one node in the network is owned and managed by a tenant and one node by a provider. The remaining eight nodes can be seen as other tenants and providers participating in the network.

In our experiment, HLF uses a Kafka-based ordering service, i.e. a consensus algorithm which is based on Kafka cluster and ZooKeeper ensemble [13] implementing a crash fault tolerant consensus algorithm. There are four nodes in the Kafka cluster and three nodes forming the ZooKeeper ensemble. In practice, since HLF is modular and our verification method is not

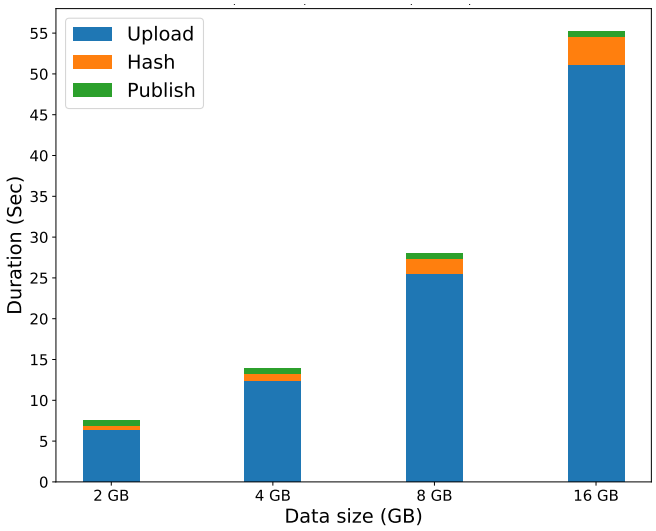



Figure 5: Time required for setup phase operations (upload, hash and publish)

dependent on the consensus process, any algorithm can be plugged and used, including Byzantine fault-tolerant algorithms. The network represents providers and tenants who agreed on SLA terms on guaranteeing the integrity of data outsourced to the cloud infrastructure.

HLF provides different components of the network as containerized services. Hence, the blockchain network is running on top of a Docker network, and each organization runs multiple Docker containers representing different components like peers, orderer and so on. A second separate physical node represents a cloud provider, specifically a storage as a service cloud provider. Users sign an SLA with the provider and submit their data after performing the setup process described in Section 6.2. In our experiment python Simple HTTP server, which implements the required functionalities like accepting tenants data upload requests, computing hash of submitted data and publishing hash value to the ledger is running on the provider node. A third separate physical node is used to represent a user. Users perform the setup and verification process on this node.

The data size is a significant factor in our verification process because the time needed for operations like hashing is directly related to the data size. For our experiment, we used different data sizes ranging from 2GB to 16GB. This range of data sizes is enough to show our experiment goals, but in practice, cloud users can upload tens or hundreds of gigabytes of data to the cloud. Initially, all the files are not cached i.e the cache is cold.

## 8.2 Performance Evaluation

The SLA verification method presented in this study requires two additional resources. First, performing the steps presented in Section 6.2 takes additional time and second, to participate in the blockchain network, it requires building at least one node in the HLF blockchain network. The results of the performance evaluation are structured in three parts: the time required for operations in the setup phase, the overhead of the verification phase, and additional resources required to participate in the process.

Using the setup described above, we measured the time overhead caused by the verification process. In an environment where there is no integrity checking the only task is to upload the

data to the cloud provider. Adding integrity checks requires two more tasks, namely hashing the data and publishing the hash value in the ledger. There are two kinds of hash operations performed by tenants (i) hashing the data alone to be sent for the provider and be published in the ledger (ii) hashing the data with random nonce values. Figure 5 shows the time required for the first kind of hashing (i.e. hashing the data alone), publishing the hash value, and uploading data. Note that the time to publish a hash of the data is constant, because the output from a given hash function is always of the same size, regardless of the input size. Time to publish means the time required to write a hash value into the ledger. As shown in the figure it is relatively small (10% and 1% for 2 and 16 GB respectively) compared to the total time.

But it should be noted that our blockchain network is being simulated on a single physical machine. In a real production setup, participants in the network are physically distributed and the time to perform write operations could be higher. Section 8.3 describes the performance of HLF on geographically distributed participants.

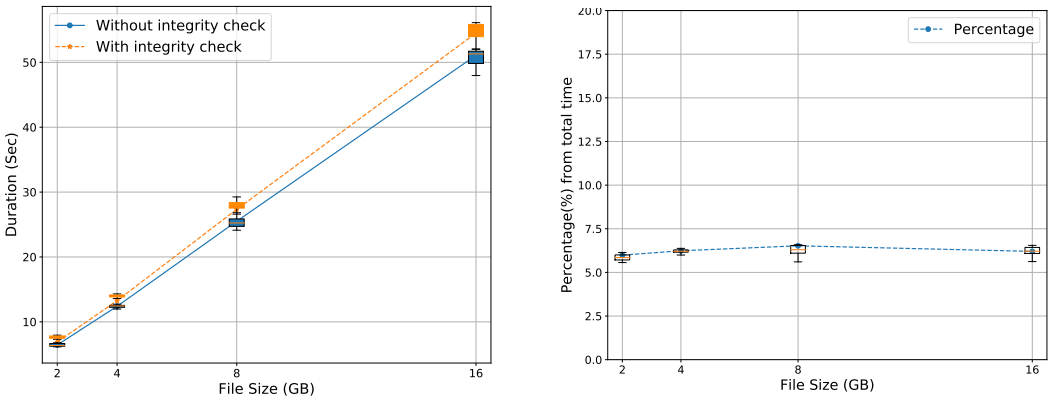
We performed the experiment for ten rounds and results are reported over an average of ten rounds. Writing the output of the hash function to the ledger takes on average 0.725 seconds, almost twice the time required for reading from the ledger. It is because writing to the ledger also performs a read operation to avoid multiple entries of the same data in the ledger.

As presented in Section 6.2.1, the setup phase requires selecting $n$ random string values and performing the hash of the data with each of those values (second kind of hash). For example, an SLA with a validity period of five years and a frequency of one verification per day, the tenant selects 1825 random values and hashes the data with each value. In comparison, our method performs a smaller number of hash operations than other cryptographic solutions presented in Section 4.2. However, it takes more computation time than other solutions. It is because our method performs a hash of the whole data with each random strings while other solutions compute a hash of a few blocks out of the whole data. Section 8.3 presents a possible optimization technique using parallelism. Moreover, in terms of resources our method requires more resources to participate in the blockchain network.

Without considering the second type of hash, Figure 5 shows that the time required for uploading data is dominant over other tasks. This task is not avoidable even without performing an integrity check. For the second type of hash, if a single hash operation takes $t$ seconds, the second type of hash operation takes $(n * t)$ seconds using a single process, where $n$ is the number of random strings. For example, to hash a 2 GB and 16 GB files take 0.4564 and 3.4266 seconds respectively. However, this task is highly parallelizable and the time could be optimized to $\frac{(n*t)}{p}$, where $p$ is the number of processes used for the hash operations and it is bounded by the number of CPU cores available on the tenant's machine.

If we assume a verification phase, it consists of asking providers to perform a hash of the data with one random string. Hence, this hashing task is the only additional time compared to the baseline i.e. without doing integrity verification. Figure 6 shows a comparison between the baseline (without integrity check) and checking integrity performed using the setup described above. The baseline operation (the solid blue line in Figure 6) measures only the time needed to download the given data while the integrity check (the dotted orange line in Figure 6) measures time to download the data plus the time for integrity checking operations as described in Section 6.2.2. Doing integrity checks introduces an addition of around of 6% of total time. The additional time is a result of hashing, and it is directly related to the data size which is also related to the baseline (upload or download) time. In the next section we present optimization measures for hashing large size data.

The other resources needed to perform the proposed SLA verification is a node to participate in the HLF blockchain network. A tenant or a provider can join the HLF network as an organization with minimum requirements for operation. These include a node which participates in the




(a) Time with and without verification

(b) Overhead percentage

Figure 6: Time overhead as a result of integrity verification

ordering service (i.e. consensus process), a node acting as a peer to maintain the state and store a copy of the ledger, and a client which acts on behalf of the tenant and submit transactions. These requirements can be achieved using light containers. HLF provides a set of containers for different services, and they can be hosted in a machine having as low as 2GB memory. Regarding disk requirements to store the blockchain data, every entry in the ledger is a key-value pair, and this should be easily manageable by using regular personal computer storage devices.

In the next section, we present a discussion and an analysis of the proposed SLA verification mechanism. We discuss advantages, drawbacks and future optimizations that can be done to have better performance.

## 8.3 Analysis and Discussion

The method proposed in this paper addresses the issue of openness (non-secretive) in the SLA verification process. The method relies on a distributed ledger which runs a consensus algorithm to keep an untampered state of the stored data. The verification process integrates security and trust by design. Tenants and providers can perform verification independently, and no one is dependent on another single entity. However other entities are required to keep the blockchain network running.

We used HLF for our prototype implementation, and we deployed HLF with the default configuration without doing any optimization. If there is a need for high throughput, e.g. a large number of clients with frequent uploads, HLF can be optimized to handle more than 3500 transactions per second as presented in [13]. Moreover, in our experiment, the blockchain network is entirely running on a single physical node using Docker containers and virtual networks. In practice, participants in a blockchain network are geographically distributed. Such distribution introduces additional latency in the process, and our method should be further evaluated on this regard.

In our verification process, the given data is hashed with $n$ random strings, i.e the process executes $n$ hash operations. This verification process takes a longer time when compared to other integrity checking protocols. The hashing process can be optimized by using parallel processes. Since the tasks of hashing the data with $n$ different random strings are independent of one another.

One of the assumptions in our threat model is a secure communication system between the cloud provider, tenant, and ledger. In the absence of such secure communication system, a man-in-the-middle attack can affect our verification process and create a conflict between tenants and providers. For example, a malicious attacker can alter the data sent from the provider to the ledger. Such an attack can cause a conflict when a tenant claims an SLO violation.

The provider neither lies nor guesses the result in advance when asked to compute the hash of data with a given random value because the tenant uses a different nonce value for each verification. Service providers can also check the validity of SLO violation claims by computing the hash of the data over its current state and comparing the result with the one stored in the ledger.

We assumed that a tenant would keep some part of the generated proof private. If a tenant loses these values, it is impossible to proceed with the verification tasks. Hence, the tenant may require a highly available, secure and private data storage mechanism.

On top of the proposed method, additional features can be added to the process. For instance, encryption can be used to add confidentiality. Blockchains can be further elevated to automate different tasks in the SLA life-cycle including payments for service and automatic compensation for SLA violation.

In this study, we showed the advantage of having secure elements in the SLA verification process for data integrity. The secure element used in our case is a distributed ledger (blockchain). This secure component is highly programmable to perform different tasks; as a result, any remote data integrity checking protocol can be implemented following our method.

In addition, the same logic can be applied to perform verification of SLAs described in our previous study. Secure components are designed for attesting and reducing the number of trusted components in different layers of an information system. For example, having a secure hardware component in the IaaS cloud could help to attest the outputs of security monitoring devices to protect them from intentional or unintentional changes by the provider. Of course, the implementation requires a more detailed study.

# 9 Conclusion and Future Work

In this paper, we started by discussing the problem of trust between providers and tenants while performing SLA verification. The previous study recommended cooperation between tenants and providers in order to achieve the desired goal. Different incentives were presented for both parties to cooperate in the verification process. However, in practice, it is difficult to have verified cooperation. This paper addresses such an issue by introducing a third component, a secure trusted and distributed ledger (blockchain) in the process.

The method allows users to directly participate in the process of SLA verification and in keeping a proof used to support an SLA violation claim. As a user-centric service, our goal is to make users part of the SLA life-cycle process and distributed ledgers formed by tenants and providers help to achieve the desired user-centric design goal. In this paper we assumed an SLA guaranteeing data integrity in the cloud. We have presented the seriousness of keeping data integrity in many applications of IT systems.

We briefly discussed related works and presented an introduction to blockchains. We described types and implementations of the blockchain technology in addition to consensus algorithms used in existing blockchains. We dived into the details of a specific implementation of the blockchain, Hyperledger Fabric (HLF). We use HLF in our prototype implementation. HLF is a permissioned blockchain and offers few advantages over other types of implementations like allowing pluggable consensus and writing chaincode (smart contracts for HLF) using any general

purpose language.

Most existing works for remote data integrity checking rely on a third-party entity to perform the audit (verification) but such solutions require just as much trust as providers for the task. Solutions without a trusted third-party fail to fulfill our desire to have a two-sided verification, i.e. they do not allow verification by both parties.

In our work, we proposed to anchor a piece of evidence for data integrity in a ledger and guarantee that the tenant and the provider will have the same block ID from the ledger, indicating the same value. It should be noted that having such a value cannot directly resolve a conflict between providers and tenants. However, it could help in the process of conflict resolution. One way can be in the process of legal action.

The verification protocol requires tenants to generate a fixed number of random nonce values and hash the data with each value before sending it to the cloud. The verification is performed by asking the provider to perform hash over the data and compare the return result with a locally stored value. Previously generated nonce values are used to force providers in performing a fresh hash computation in every verification request. Our process guarantees that the tenant and the provider will hold the same hash value which can be used to resolve the disagreement between a tenant and a provider.

We implemented a prototype for the proposed SLA verification process and run experiments to do a performance evaluation. We have measured the time required for three basic operations namely hashing, uploading data and publishing the hash value into the ledger. The time overhead to perform a verification is the result of the hashing operation which takes around 6% of the total time. In the context of SLA verification, we believe this is an acceptable overhead compared to the consequences of not having integrity check. The impact of the observed overhead should be analyzed given use cases; for example, if the use case is in time-sensitive applications, a 6% overhead may not be acceptable. Since the output of the hash function is always the same size, the time to publish it is constant.

We finalized the paper by providing a discussion and analysis of a few points that are related to the proposed method. The study showed the advantages of having secure components in the SLA life-cycle management. Using a secure, trusted and distributed storage to keep a piece of evidence for data integrity, it is possible to reduce the need for trust between providers and tenants in SLA verification. With the same approach, other secure components can be used to reduce and remove unnecessary trust between SLA participants.

In this paper, we focused on showing how to use a secure component in the SLA verification process. However we miss a method to define these SLAs. In our previous work [35] we have defined an SLA language called ECSLA. ECSLA is used to define an SLA guaranteeing the performance of an NIDS. As a future work, ECSLA can be extended and SLA definition for data integrity could be done following a similar method to NIDS SLA definition. The extension requires studying the performance metrics and users requirement description mechanism for the data integrity property.

# References

[1] A. Teshome, L. Rilling, and C. Morin, "Verification for security monitoring slas in iaas clouds: The example of a network ids," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–7.

[2] R. Bejtlich, *The Tao of network security monitoring: beyond intrusion detection*. Pearson Education, 2004.

[3] M. Rak, N. Suri, J. Luna, D. Petcu, V. Casola, and U. Villano, "Security as a service using an sla-based approach via specs," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 2. IEEE, 2013, pp. 1–6.

[4] "Amazon Compute Service Level Agreement," accessed August 2018. [Online]. Available: https://aws.amazon.com/compute/sla/

[5] "Cloud File Sharing and Storage for your Business," accessed July 2018. [Online]. Available: https://www.dropbox.com/

[6] P. Wang, D. J. Dean, and X. Gu, "Understanding real world data corruptions in cloud systems," in *2015 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2015, pp. 116–125.

[7] "Facebook temporarily loses more than 10% of photos in hard drive failure," accessed July 2018. [Online]. Available: https://www.computerworld.com/article/2531672/disaster-recovery/facebook-temporarily-loses-more-than-10--of-photos-in-hard-drive-failure.html

[8] "S3 data corruption ?" accessed July 2018. [Online]. Available: https://forums.aws.amazon.com/thread.jspa?start=0&threadID=22709&tstart=0

[9] "Corrupt iCloud data causes iOS home screen crash," accessed July 2018. [Online]. Available: https://www.macobserver.com/tmo/article/corrupt-icloud-data-can-cause-ios-springboard-home-screen-crash

[10] "Amazon S3," accessed July 2018. [Online]. Available: https://aws.amazon.com/s3/

[11] "DriveHQ Service Level Agreement," accessed July 2018. [Online]. Available: https://www.drivehq.com/premium/DriveHQSLA.aspx

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 30.

[14] "On Public and Private Blockchains," accessed July 2018. [Online]. Available: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/

[15] W. Ren, R. W. Beard, and E. M. Atkins, "A survey of consensus problems in multi-agent coordination," in *Proceedings of the 2005, American Control Conference, 2005.* IEEE, 2005, pp. 1859–1864.

[16] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[17] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[18] V. Buterin *et al.*, "Ethereum white paper, 2014," *URL https://github.com/ethereum/wiki/wiki/White-Paper*, 2013.

[19] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On security analysis of proof-of-elapsed-time (poet)," in *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 2017, pp. 282–297.

[20] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, 2014.

[21] F. B. Schneider, "Replication management using the state-machine approach, distributed systems," 1993.

[22] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba, "A taxonomy of blockchain-based systems for architecture design," in *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, 2017, pp. 243–252.

[23] N. Oualha, J. Leneutre, and Y. Roudier, "Verifying remote data integrity in peer-to-peer data storage: A comprehensive survey of protocols," *Peer-to-Peer Networking and Applications*, vol. 5, no. 3, pp. 231–243, 2012.

[24] Z. Hao and N. Yu, "A multiple-replica remote data possession checking protocol with public verifiability," in *Data, Privacy and E-Commerce (ISDPE), 2010 Second International Symposium on*. IEEE, 2010, pp. 84–89.

[25] Z. Hao, S. Zhong, and N. Yu, "A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability," *IEEE transactions on Knowledge and Data Engineering*, vol. 23, no. 9, pp. 1432–1437, 2011.

[26] W. Luo and G. Bai, "Ensuring the data integrity in cloud data storage," in *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*. IEEE, 2011, pp. 240–243.

[27] Y. Zhu, H. Wang, Z. Hu, G.-J. Ahn, H. Hu, and S. S. Yau, "Cooperative provable data possession," *Beijing: Peking University and Arizona University*, 2010.

[28] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling security in cloud storage slas with cloudproof." in *USENIX Annual Technical Conference*, vol. 242, 2011, pp. 355–368.

[29] "Decentralized cloud object storage that is affordable and easy to use." accessed July 2018. [Online]. Available: https://storj.io/

[30] B. Liu, X. L. Yu, S. Chen, X. Xu, and L. Zhu, "Blockchain based data integrity service framework for iot data," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 468–475.

[31] E. Gaetani, L. Aniello, R. Baldoni, F. Lombardi, A. Margheri, and V. Sassone, "Blockchain-based database to ensure data integrity in cloud computing environments," 2017.

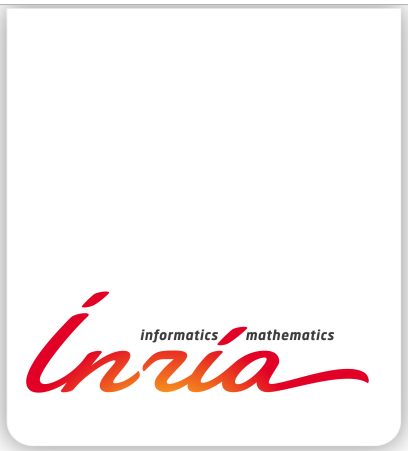
[32] "An open standard for creating a timestamp proof of any data, file, or process." accessed July 2018. [Online]. Available: https://chainpoint.org/

[33] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[34] "Docker," accessed July 2018. [Online]. Available: https://www.docker.com/

[35] A. T. Wonjiga, L. Rilling, and C. Morin, "Defining Security Monitoring SLAs in IaaS Clouds: the Example of a Network IDS," Inria Rennes Bretagne Atlantique, Research Report, Mar. 2019. [Online]. Available: https://hal.inria.fr/hal-02079860

# Contents




**RESEARCH CENTRE**
**SOPHIA ANTIPOLIS – MÉDITERRANÉE**

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399