Securing Metadata for Data Storage on the Edge

Cláudio Correia
claudio.correia@tecnico.ulisboa.pt

Instituto Superior Técnico
(Advisors: Professor Luís Rodrigues and Professor Miguel Correia)

Abstract. This report addresses the problem of securing storage services designed to support edge computing, that runs primarily on fog nodes. We identify the major threats to the integrity, availability, and consistency of data maintained by the edge storage service and we propose techniques to secure this service. Achieving low latency is one of the main motivations for adopting edge computing, for this reason we give special attention to the trade-off between security and performance.

Table of Contents

1 Introduction

Cloud computing is a model for deploying Internet applications that allows companies to deploy services in shared infrastructures, typically large data centers, that are managed by cloud providers. The economies of scale that result from using large shared infrastructures reduce the deployment costs and make it easier to scale the number of resources associated with each application in response to changes in the demand. Cloud computing has been, therefore, widely adopted both by private and public services [1].

Despite its benefits, cloud computing has some limitations. The number of data centers that offer cloud computing services is relatively small, and they are typically located in a few central locations. Thus, clients that operate far from the data centers may experience long latencies. Also, many applications require data to be sent to a data center to be processed. For applications that produce large amounts of data, this model may require the consumption of significant network resources.

Many applications deployed in the cloud provide a range of services to clients that reside in the edge of the network: desktops, laptops, but also smartphones or even smart devices such as cameras or home appliances also known as the Internet of Things (IoT). The number and capacity of these devices have been growing at a fast pace in recent years. Many of these devices can run real time applications, such as augmented reality, that requires low latencies. Also, most of these devices have sensors that produce enormous quantities of information that need to be collected and processed [2].

One solution to address the latency and bandwidth requirements of new edge applications is to process the data at the edge of the network close to the devices, a paradigm called *edge computing*. To support edge computing, one can complement the services provided by central data centers with the service of smaller data centers, or even single servers, located closer to the edge. This concept is often named *fog computing* [3,4]. It assumes the availability of fog nodes that are located close to the edge and their numbers are several orders of magnitude larger than those of data centers in the cloud.

Cloud nodes are physically located in secure premises, administered by a single provider. Fog nodes, instead, are most likely managed by several different local providers and installed in physical locations that are more vulnerable to tampering. Therefore, fog nodes are substantially more vulnerable to being compromised [5,6], and developers of applications and middleware for edge computing need to take security as a primary concern in the design.

In this report, we address the problem of securing middleware for edge computing. We focus, more specifically, on securing causally-consistent storage services for the edge, such as the one proposed in Gesto [7]. These storage services keep replicas of relevant data objects at different fog nodes so that users can read and write data with low latency. Replicas coordinate to ensure that users observe a consistent view of the data: we are particularly interested in storage systems that can offer causal consistency [8], since it is the strongest type of consistency that can be enforced without risking blocking the system when partitions or failures occur. Because fog nodes have limited resources, a storage service for the edge has to support partial replication, i.e., not all nodes store all data and some nodes may even not store any data at all but still participate in the coordination activities required to ensure data consistency.

Much work has been done in storage systems offering causal consistency on the cloud [9,10,11,12]. Since these systems rely on a fully trusted cloud, there is a recent interest in expanding these systems to the edge such as Gesto [7]. Therefore, this type of systems exhibits similar security vulnerabilities to those of fog nodes. In this context, this report identifies the major threats to the integrity, availability, and consistency of data maintained by the edge storage service and proposes techniques to secure this service. Because achieving low latency is one of the main motivations for adopting edge computing, we give particular attention to the trade-off between security and performance.

We look for lightweight cryptographic techniques such as digital signatures to assure integrity while keeping a reasonable trade-off with availability. We also provide the necessary data in messages that can be used to verify that the consistency model holds.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we introduce the key concepts relevant for our work and in Section 4 we address related work. Section 5 describes the proposed architecture to be implemented and then Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work and Section 8 concludes the report.

2 Goals

This work addresses the problem of securing storage services for the edge while ensuring low latency and efficient bandwidth usage. Namely, it is concerned with the security of services that execute on fog nodes, i.e., nodes that are placed in close proximity to edge devices in order to provide cloud services with low latency and efficient bandwidth usage. More precisely, it focuses on the security of a storage service for the edge.

> *Goals:* This work aims at designing security mechanisms that address the major threats to the integrity, availability, and consistency of data maintained by the edge storage service.

If a fog node becomes compromised, it may attempt to corrupt the data it stores, discard stored information, corrupt control messages in an attempt to prevent the proper coordination of correct replicas, or serve faulty or stale data to the client. All these actions may violate the consistency properties of the storage service, or simply render it unresponsive or untimely. A fundamental challenge is to provide an appropriate level of security without compromising the performance of the system, given that one of the main goals of supporting edge computing is to provide services with low latency to clients. To address these faulty behaviors we will need to use a number of complementary techniques, from cryptographic signatures that may allow clients to verify the integrity of data, to replication techniques to ensure the availability of data.

The project will produce the following expected results.

> *Expected results:* The work will produce i) a set of algorithms to secure storage services provide by fog nodes; ii) an implementation of these mechanisms for a storage service using Unix nodes, iii) an extensive experimental evaluation using a deployment on an experimental testbed such as Grid'5000 [13].

3 Background

In this section, we will introduce relevant concepts that are important to the understanding of this document. We will begin by presenting the edge computing model in Section 3.1. In Section 3.2 we will explain why causal consistency is important in data stores and in Section 3.3 we will present two use cases of applications on the edge that depend on causality. Finally, in Section 3.4 we will present security properties and mechanisms that are useful in the edge.

3.1 Edge Computing

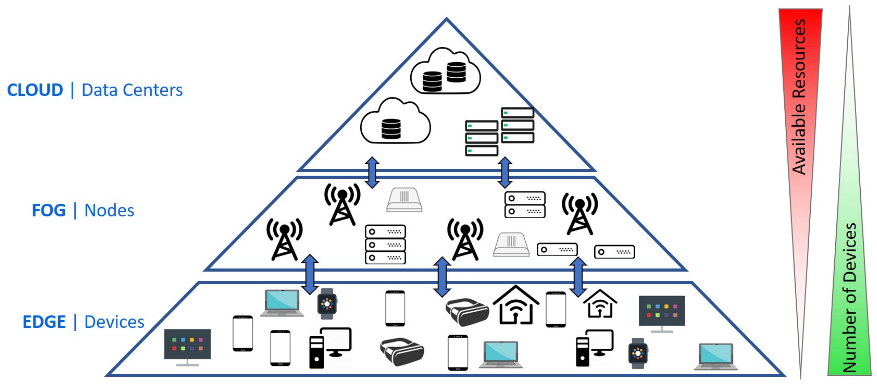

Fig. 1. Three-layer fog computing architecture.

The emergence of IoT and its use of services in the cloud causes the need for proximity between computing resources and the edge. Edge computing is a model that takes advantage of the computing capabilities of devices that are at the edge of the network. This computation near the edge devices merges with the idea of fog computing, allowing all edge devices to take advantage of the close computation offered by fog nodes. The existence of distributed fog nodes close to the periphery of the network provides rich computing resources with low latency, opening a window to a broad new kind of applications.

Figure 1 shows a general architecture of fog-edge computing, which consists of multiple infrastructures layers. The number of nodes in each of these layers decreases as we move from the cloud to the edge. On the other hand, the memory and computation power increases from the edge to the cloud. In the cloud layer, we may find data centers while in the fog layer we may find ISP servers, private data centers, and 5G towers. Finally, in the edge, we may find all kinds of devices such as desktops, laptops, tablets, mobile devices, sensors, and actuators. In this architecture, the devices located in the edge layer will communicate mainly with those located in the fog layer. In addition, fog nodes will also have to communicate with the cloud where all data will be stored in a more secure and persistent way.

Devices in the edge establish a connection with fog nodes and execute operations. Fog nodes are responsible for the propagation of such updates and the correspondent metadata. These fog nodes are located in an exposed location that increases the risk of being attacked, raising the probability of such nodes to become malicious as discussed in Zhang et al. [5], Mukherjee et al. [6] and Zhou et al. [14] surveys. If a fog node becomes malicious, it can generate or manipulate metadata in a way that it affects the correct functioning of the system.

Current data stores use metadata between servers to have data consistency. In the next section, we will describe in more detail the importance of having consistency over replicated data and the implicit trade-off involved in ensuring such guarantees. Most importantly we describe why many practical systems adopt causal consistency.

3.2 Causal Consistency

Data consistency assures developers about the state of geographically replicated data after undergoing multiple operations. With this guarantee, the development of applications that uses replicated data becomes easier and more efficient. In this section we show why causal consistency is a relevant choice and in Section 3.3 two examples show why data consistency is important.

Typical data stores located in the cloud offer data consistency. Usually, these systems such as Saturn [9] and COPS [10] penalize availability to offer consistency. In this work, we generalize the idea of a data storage and one important characteristic they all have in common is the goal of achieving the best data consistency possible.

In the context of replicated data access, we can divide data consistency into two categories: *strong consistency* and *weak consistency.*

In *strong consistency*, all replicas observe a consistent state. In other words, replicas apply all the updates in the same order, serializing the updates. Such a strict model comes at a heavy price. Replicas must coordinate to synchronize each update, severely penalizing the system availability. In fact, it has been proved that it is impossible offer simultaneously *strong consistency*, full availability, and tolerance to network partition, a fact known as the CAP theorem [15].

To increase availability, it is necessary to weaken the consistency guarantees. *Weak consistency* models offer high availability by relaxing some consistency proprieties. Subsequently, replicas no longer behave like a single unreplicated data item and users can observe behaviours that do not occur in a centralized system.

Causal consistency [8] is one of the most relevant *weak consistency* models. This consistency model captures the causal order between operations and enforces the visibility of this order in the entire system. In other words, given two updates, a and b, if a potentially causes b, then in all replicates in the system it is only possible to observe the update b after update a is applied. If b is causally dependent on a is denoted $a \rightsquigarrow b$.

We are interested in systems that can offer causal consistency, because it has been shown that this is the strongest consistency model that can be offered without compromising availability [16]. Systems such as Gesto [7] offer causal consistency on the edge. Since the edge is still a current subject of research, we expect to see in the near future additional systems that support causal consistency in this setting. In the next section, we will present two use cases, that illustrate some applications for the use of data stores in the edge with causal consistency.

3.3 Causally-Consistent Edge Applications

Applications that take advantage of the edge can offer users low latency services because many times users can find a server just a hop away. Thus, the existence of these servers near the edge as fog nodes requires a new model capable of handling data consistency and data security. We present two distinct examples of applications that run on the edge, where causality is an important guarantee. Additionally, we show how a malicious node can compromise the system in case that no safeguards are in place for ensuring the system's security.

Messaging Application Many social networking platforms have developed their messaging applications, typically involving features like group chat, allowing multiple users to exchange messages among them. In this context, a fog node can aggregate and process multiple messages and send them to the cloud, reducing latency and saving bandwidth usage. It is common for such group chats to be administered by users which have the ability to add and/or remove other chat users. A good example of why it is important to maintain causal consistency guarantees is when an administrator revokes a user's access to the chat. This information must be propagated to all replicas that maintain the chat state. Another user, on another replica, can observe this update and send a message to the chat under the assumption that the revoked user will not read it. The message must also be propagated to all replicas that maintain the chat state. Note that the "revoke" update and the "new message" update are two different operations, issued at different sites by different users. They can be propagated and delivered to different sites in different orders. Still, each replica must ensure that the revoked user can never read this new message, since the message has been added after the user has lost access to the chat. Causal consistency enforces such guarantees.

If the fog node that originally received the "new message" update generates metadata correctly, the remote replicas will know the correct order to apply these updates, ensuring causality. However, if the fog node that generated the metadata is malicious, it may intentionally generate the metadata in the wrong order, and the remote replicas will apply the updates in an order that does not respect causality. One consequence will be the revoked user reading the new message, even though it has been added after the user has lost authorization for the chat.

Mobile Interactive Multiplayer Game There has been a growing trend for the development of mobile multiplayer games for smartphones which offer the ability for users to trade objects within the game. Currently, these games require users to connect to the cloud to avoid object replication attacks; this offers high latency which forces users to wait several seconds until the transactions are accepted. Replication attacks can be accomplished if a user trades objects offline and then resets his state before the trade, allowing them to trade the same object multiple times. By taking advantage of fog nodes, these transactions could be processed closer to the edge to lower the latency, reduce the waiting time, and provide a better experience to mobile game users. Another important factor is the number of users that can simultaneously be using such an app.

A good example is Pokemon Go [17]. When the game was launched, there were millions of players around the world using a server located in the cloud that resulted in high bandwidth usage. A game like Pokemon Go could be processed in a fog node, reducing both the latency and bandwidth usage. We now present a use case where a user trades and sells an object, the kind of operations that multiplayer games offer. In particular, we show the importance of causality in such operations. In a trade scenario where the user Alice executes two operations: 1) user Alice trades object A for B, 2) and then sells object B. The second operation depends on the first. When these operations are propagated to other replicas, they must be applied in the same order, 1 and then 2.

However, if a malicious fog node generates metadata in a different order, a remote replica may try to sell object B from Alice, before the trade, breaking the consistency of the application, since Alice does not possess object B in the remote replica. In the next section, we will present security properties and mechanisms that can avoid such attacks.

3.4 Edge Security

In the fog computing model, fog nodes bring resources closer to the edge. However, the security guarantees offered by the cloud cannot be moved so easily. In the cloud model, data centers are isolated and well protected. On the other hand, fog nodes will be more exposed, and subsequently, it is not possible to use the same techniques as in the previous model, this challenge is discussed in Zhang et al. [5], Mukherjee et al. [6] and Zhou et al. [14] surveys. Data centers in the cloud model are few, which allows easy key management and secure communication channels between them. However, since there are large numbers of fog nodes, it is not viable for a single fog node to know the identity of all fog nodes in the network.

Fog nodes can be subjected to several malicious attacks, therefore, leaving the system in a faulty state. Due to the exposed location of fog nodes, an attacker could gain physical access to a fog node. Denial-of-Service (DoS) is also a possible attack; an attacker can gain control of multiple edge devices and execute a Distributed Denial-of-Service (DDoS). It is also possible to monitor the amount and duration of the user's data accesses and predict an attack based on the user's behavior. Considering the number of fog nodes, it is difficult to detect if a specific node is under attack, thus attackers have more opportunities and time to discover weaknesses in the system.

To address these issues, we must initially enforce that only authorized edge devices can use resources in the fog, based on some digital signatures. After this, we will introduce the security properties that we will enforce on the edge and address the mechanisms that help us deal with the challenge that is the huge number of devices and their heterogeneity and which allows building a base of trust in the fog, even if a fog node goes rogue.

3.4.1 Key Security Properties In this section, we describe some key concepts in information security that allow us to reach security in a system regardless of the underlying implementation. The focus of this work is to ensure some of the following properties in a storage system located in the edge of the network. To ensure the correct execution of our system, we must offer security guarantees. Achieving these guarantees in the edge forces us to tackle new challenges, such as the large number of devices and their heterogeneity. These challenges are a problem for key distribution. Another challenge is how to have a base of trust in an edge environment when a fog node may become malicious. In the following paragraphs, we introduce some key security properties and how to achieve these properties in an edge environment. Our focus is to secure the metadata generated by a data store.

Authentication is the act of an entity proving to be what it claims to be, in other words, the ability to provide some proof of its identity. In fog computing, both the edge devices and the fog nodes require authentication. So, to avoid an untrusted party to pretend to be a legitimate node and receive data from a device, fog nodes must authenticate towards the edge devices. Similarly, edge devices must authenticate towards fog nodes, so that a fog node can impose some policy, such as allowing an authorized device to access a piece of data.

This mutual authentication requires an infrastructure with some base of trust. Since the cloud is trusted and the fog nodes are well connected, this infrastructure should use the cloud to help authenticate both parties. Examples of such infrastructures are Public Key Infrastructure (PKI) [18] that uses trusted entities to certify public keys or Identity-Based Encryption (IBE) [19] where there is a trusted entity responsible for generating private keys for the devices.

Data Integrity consists of the property that no unauthorized entities have manipulated the data. For example, when a data store sends a metadata message, an intruder should not be able to substitute a false message for a legitimate one. Also, if a cloudlet stores data, it should be able to detect if the stored data has been tampered with. Due to the fog nodes proximity to the edge, an attacker can easily intercept messages, or physically access a fog node and tamper with data. The integrity of messages in our system must be assured, from the moment a fog node generates a message until it reaches its destination. This is important since fog nodes are responsible for propagating messages.

In the metadata example, it is critical to detect a tampered message because even the smallest change in the metadata, like a number in the timestamp, can break the causality of the system. An efficient way to enforce integrity requires entities to calculate a hash of the data and generate a digital signature using a secret, something like a Message Authentication Code (MAC). If the metadata also brings a MAC, other entities in the system can verify the integrity of the message.

Confidentiality is a property used to keep the content of the information from all but those entities authorized to read it. When a data store system propagates metadata from an update, replicas may or may not be interested in that message. Thus, every replica must be able to read the content. Since this work focuses on the correct behavior of the system, confidentiality is not one of the main goals. However, an outside entity can derive information if it can read in plain-text all the messages in the system. For example, it is possible to infer a device location by reading the signatures in the metadata and what cloudlet received the update.

To ensure confidentiality, it is required that every message source ciphers the content using some secret, shared only among the authenticated and authorized entities in the system.

Non-Repudiation is a property which prevents an entity from denying previous actions. For example, if Alice generates and sends a message, there can be no doubt that Alice is the source of the message. This property is important to identify malicious entities in a system. Once a cloudlet receives an update from a device, it generates the correspondent metadata and propagates this message to other nodes. Other nodes can only accept this message if it arrives digitally signed by the cloudlet. If a cloudlet generates and propagates incorrect metadata, it is necessary to detect and identify the responsible cloudlet. If the metadata its signed by the private key of the source cloudlet, it is possible to identify the responsible cloudlet. Non-repudiation makes it impossible for a malicious cloudlet to refute its responsibility.

Non-repudiation can be obtained using techniques similar to the ones used to achieve integrity, for instance, by requiring every entity to sign their messages digitally.

Availability is the degree to which a system can satisfy requests. High availability is an important feature that our work must ensure, providing that if the system is not capable of responding to requests, then clients may stop using the system, or break Service Level Agreement (SLA). Expensive security protocols and mechanisms, that exhibit poor performance, may prevent the system from satisfying request in due time, therefore contributing to availability loss. To help in keeping the system availability, we will use light cryptographic techniques that have low overhead.

3.4.2 Mechanisms To achieve the security properties mentioned above, we can take advantage of cryptographic primitives giving entities access to data respecting some policy. Edge devices are constrained in resources. Therefore we will look for mechanisms that preferably place the heavy computation on the fog side. We also look for an efficient cryptography scheme that can deal with the huge number of edge devices.

After this, we introduce Identity-Based Encryption, a scheme that uses asymmetric keys and provides an efficient key distributed solution, something vital due to the huge number of devices in the edge. This scheme allows entities to authenticate each other without the need to exchange credentials before establishing communication.

Then, we will discuss Trusted Execution Environments, an emerging technology inside of a processor offering a new base of trust. As a fog node is responsible for receiving and propagating updates, clients must have some proof of trust to use a fog node. Since we assume a fog node can be attacked and become malicious, we need to offer the client some base of trust.

Identity-Based Encryption (IBE)[19] is a scheme that uses asymmetric keys and allows any entity to generate someone else's public key locally, which enables any pair of users to verify each other's signature without the need to exchange public keys.

The cloud model uses the PKI that requires public keys to be shared. In PKI, entities exchange public keys to authenticate each other, by proving they own the private correspondent key. In this scheme the public keys come in a certificate, which contains expiration date for key revocation. This scheme implies that one entity must have the certificate of another to be able to authenticate. Thus, a fog node would have to store as many certificates as there are fog nodes, to confirm digital signatures of other fog nodes.

In a system with many entities, the distribution of the public keys may become a burden. In IBE, public keys can be generated locally, using as input a string that identifies the target identity. However, the private key of a user must be generated by a trusted third party. One such party is called Private Key

Generator (PKG). The PKG only provides the private key to an authenticated entity, meaning that entities must authenticate towards the PKG to obtain a private key correspondent to their identity.

For example, if Alice receives a message from Bob, she can locally generate Bob's public key and decipher the message, knowing that only Bob has the corresponding private key (providing authentication and integrity). Alice is not required to exchange keys with Bob. A public key for Bob can be generated by providing a string, in this case, "Bob". The PKG can only give the corresponding private key to the entity Bob.

In this scheme, public keys do not possess certificates and therefore no expiration date. Subsequently, key revocation is done differently from PKI. Key expiration can be done by adding the current date to the string when generating public keys. For example, when Alice generates a public key of Bob, she uses the string "Bob || current-date". This method implies that all entities in the system must authenticate every day if the granularity of the "authentication lease" is one day, for example.

Trusted Execution Environments (TEE) [20,21] In the fog model, fog nodes can become compromised and lead to leakage of private information or leaving systems in a faulty state. In particular, the exposed location of such nodes makes them more vulnerable. TEE is a secure area in the processor that offers a higher level of security than the operating system. Code that executes inside a TEE is isolated from the operating system, providing integrity and confidentiality, even if the operating system is compromised.

TEE is a natural choice to secure the computation and sensitive data in fog nodes. Edge and fog devices are a very compelling opportunity to use processors with this technology. With it, fog nodes can securely store private keys and offer clients a base of trust. If fog node providers choose to have this kind of technology, will have an additional security guarantee over the fog model.

A processor with this technology can operate in one of two modes, *normal world* or *secure world. Normal world* is where the operating system and applications run without any special security guarantees, while in *secure world* mode, application code has isolation, integrity, and confidentiality. This division between two different modes requires developers to build applications in two parts, one for each world. One important goal of these architectures is enabling any application to take advantage of this secure environment. Therefore, device hardware configuration provides a TEE API, where applications in *normal world* can request secure operations in *secure world.* These requests require the processor to switch context between the two modes. When the processor switches from normal mode to secure mode, it fetches the last state of the application that will run in secure mode, decrypts it and verifies its integrity. When it goes from secure to normal mode, it encrypts this state and stores it in memory. Between context switch, this state is encrypted and decrypted using a key that is secure in a chip element.

TEE also offers a remote call to verify that *secure world* inside the processor has not been tampered with, which is known as *attestation*.

Examples of architectures with such TEE are: *Intel SGX* which is an architecture that already exists in many Intel processors and is a practical solution to the fog (Intel Fog Reference Design [22]); *ARM TrustZone* that exists in processors focused on the mobile devices, offering low-power consumption; *AMD SEV* that can offer processor at a low price.

4 Related Work

This section discusses the reviewed literature in the areas of data consistency and security in edge-fog. To achieve the goals in Section 2, we searched for ideas present in the following works.

Our work will focus on providing security guarantees in a data store at the edge, so we will try to define what kind of data store makes sense at the edge and how to reach the security guarantees for this data store. To achieve this goal, we will identify and then solve the following questions:

– **Architecture**, what kind of architecture can a data store at edge have?
– **Operations**, what are the operations that a data store have?
– **Messages**, what kind of messages can exist in a data store?
– **Authentication**, how to provide authentication for clients and fog nodes?
– **Integrity**, how to ensure message integrity?
– **Confidentiality**, how ensure that only authorized entities can read the content of a message?
– **Non-Repudiation**, how to identify the origin of a message?
– **Availability**, how to respond to the highest possible number of requests?
– **Untrusted Nodes**, how to avoid or deal with a fog node becoming malicious?

The remainder of this section is organized as follows: Section 4.1, presents three systems that offer causal consistency on a distributed data store. This work will help answer what kind of architecture, operations, and messages exists in such systems. Section 4.2, addresses previous work related to security that can be applied in fog computing and helps to answer the other questions presented above.

4.1 Causal Consistent Data Stores

In this work, we assume that a data store is a distributed repository for persistently storing data, which is replicated among multiple remote replicas. So that a remote replica can apply an update respecting causal order, it is necessary to generate metadata associated to each update. For the cloud level, there has been considerable research on how to generate and propagate metadata efficiently. However, most works assume that cloud nodes can trust each other, and do not use sophisticated security mechanisms (other than using secure channels when exchanging metadata among nodes). We will introduce the use of stronger security mechanisms in the forthcoming sections.

We start by presenting several alternatives to offer causal consistency in data stores while keeping metadata size small and low latency for clients. In the following paragraphs we describe Saturn and COPS, which detail architectures for the cloud layer and GESTO, that is located at the edge. These are the kind of storage systems that we pretend to secure. Therefore, we will be able to answer what type of architecture, operations, and messages exists in such systems.

4.1.1 Saturn [9] Saturn is a metadata distribution system at the data center level that ensures the causality of updates in a geo-replicated storage system. In Saturn, a client is required to connect to a data center before he can perform read and write operations. To ensure causal order of the updates, Saturn generates metadata for each update. The metadata associated with each update is called *label*. Data centers are responsible for generating labels for each update and propagating them to remote replicas and clients. Saturn assumes that replicas are organized in a tree for an efficient metadata propagation.

A label in Saturn contains the following components: a *type* (that can have two different vaues, namely update or migration); a *source* identifier of the data center that generated the label; a *timestamp* that consists of a single scalar; and a *target* that indicates which data item should be updated (or a data center in the case of migration).

When a client connects to a new data center, also known as *migration*, first he needs to attach to this new data center. The client sends his label, and the data center verifies that it has all the necessary updates to ensure that the client's causal history is respected. If an update is missing, the client will have to wait until the data center can perform all the required updates. Once the client is attached to a data center, he can perform write and read operations.

A *read* request from a client implies that he has performed attachment and therefore the server contains all the required updates to respect the causal past of the client. Thus, the data center only needs to return the most recent value and its label; the client will replace his old label with the new one. After replacing the label, the client becomes causally dependent on this new value. This dependency is captured by the label.

For a *write* request, the client sends the data key (target), the new value, and his current label. Then, the data center applies the update and generates a new label, with a greater timestamp than the one of the client. Afterward, the new label is propagated to the other replicas and returned to the client, replacing his old label. When a remote replica receives the new data and the new label, it first verifies if it has all the necessary updates to guarantee that the causal order is respected, then it applies the update and finally stores the label.

An essential feature of Saturn is its data center topology: since the replicas are organized as a tree, the propagation of the metadata between them is more efficient. Also, the size of metadata is constant, independently of the number of replicas in the system. However, due to the reduced size of the metadata, the updates have more false dependencies that correspond to an increase in latency when applying the updates.

4.1.2 COPS [10] COPS is another geo-replicated system at the data center level that offers causal consistency for a distributed key-value store.

In COPS, in contrast to Saturn, clients are responsible for handling their causal history. To achieve this, clients maintain, in memory, a graph with their dependencies, hereafter referred to as *context*. The context is composed of objects ID and their corresponded version; this version is a single timestamp. In COPS, replicas do not assume any particular architecture to propagate metadata.

Whenever the client *reads* a new object, it updates its context with the new version of this object. The data center only needs to return the latest object and the correspondent version.

For a *write* operation, the client first generates the *nearest dependencies* base on his current context. Nearest dependencies are obtained by eliminating updates that are directly dependent, and therefore significantly smaller in size than the context. Then, the client sends the nearest dependencies, the object key, and the new value to the data center. Once the data center receives the update, it assigns the key a version number and returns it to the client. Since the client holds a session with the data center, the update can be immediately committed at the local data store. In a remote replica it must verify that it has all nearest dependencies locally, before committing the update.

Although the *migrations* are not specified in COPS, it is still possible for a client to migrate from a data center to another. Since clients hold their entire causal past, once they arrive at the new data center, they can just wait until all the required updates are committed in the new data center before reading.

The main feature in COPS is that the dependency context is stored in the client, allowing a write to be performed at any data center while still offering causal consistency. A direct downside is the memory and computation required by the client to maintain his context.

4.1.3 Gesto [7] Gesto is also a system that offers causal consistency on a data store, however with architecture at the edge level. This system takes advantage of nodes close to the edge to build an architecture that can offer causality from the cloud to the edge. These nodes, which provide resources and are outside the data centers, closer to the edge devices, are called cloudlets, conceptually similar to fog nodes.

Gesto has a significantly different architecture from previous systems. It uses a star topology, where we can find in the middle a single broker and all the cloudlets in one region are connected to the broker of that specific region. Clients then connect to cloudlets and perform updates. The metadata generated to these updates is composed of two timestamps, known as multipart timestamp, one for the regional broker and another for the local replica (cloudlet). This metadata only has meaning within the region where it was generated.

To *migrate* from one replica to another, a client uses his multipart timestamp. When arriving at a new replica, the client issues an *attach* request and sends his multipart timestamp. A replica keeps track of the last update that it has received for each regional replica and the highest regional timestamp. Thus, a replica can accept the attachment request when the following conditions are verified: having a highest regional timestamp than the client, and when it has seen a higher timestamp from the previous replica of the client.

To *write* an object, a client generates a unique identifier and forwards the update along with his current multipart timestamp to the local replica. The replica then increments the local timestamp, applies the update and propagates the metadata to the regional broker. When metadata arrives at the broker, "merges" the metadata consistent with causality, using the regional timestamp and propagates to the replicas that cache the associated data.

A *read* operation expects the client to have already attached to the local replica. Therefore, the replica can simply return the requested data and correspondent metadata.

This system has small metadata and an efficient migration between replicas to tackle the high dynamism of the network, offering an interesting solution to guarantee causal consistency on the edge. A consequence of small metadata is that the system can have high false dependencies between regions.

4.2 Providing Security in Fog

The existing systems that offer causality in data stores (Section 4.1) are mostly focused on performance, and thus disregard the need for enforcing security guarantees. However, providing that the implementation of such systems is moved to the fog, new security challenges arise [5,6,14]. To find an answer to such challenges, we survey existing literature which dwells on securing multiple layers of the fog infrastructure, including the cloud, fog, and edge. It is noticeable that, although there is excellent research in this area, there are very few solutions available which provide security, particularly in communications. In this section, we present some ideas and works that are relevant for our solution.

In Section 4.2.1, we start by introducing three schemes that can help authenticate entities in the edge-fog. Then, Section 4.2.2 presents two different techniques that establish secure communication between entities in the network with integrity, confidentiality, and non-repudiation. Finally, in Section 4.2.3, we take a close look at previous work which mitigates the effect of compromised fog nodes on a system's infrastructure.

4.2.1 Authentication

As observed in Section 3.4.1, authentication is an important property in fog computing. However, due to the heterogeneity of edge devices and low latency requirements, specific techniques are required to offer authentication at edge-fog layers. The remainder of this section is devoted to presenting three schemes that try to hold this property in such an environment. We first present a lightweight authentication scheme based on hardware, then an architecture that allows devices to use resources from a cloudlet in an offline environment, finally a mutual authentication scheme that uses symmetric keys.

Lightweight Hardware Based Secure Authentication Scheme for Fog Computing [23] This scheme offers a unique way for devices to authenticate themselves towards a fog node without cryptographic keys. This is possible by taking advantage of the heterogeneity of edge devices. Since chip manufacturing is an unpredictable and uncontrollable process, it is possible to have physically unclonable functions (PUF) in a device. These functions guarantee that, given a set of inputs, the output will be unique for each device. When a manufacturer builds a device runs a series of challenges on this device and stores the responses to these challenges. With this pair of challenges and responses, he builds a model using machine learning techniques. This model can be created using methods such as neural network or logistic regression; the training phase uses the challenges as input and the responses from the PUF as output.

Once the device is online it will authenticate towards a server; the server will require to have access to the extracted model. Then the server will issue a set of challenges to the device and run the same challenges in the local model. When the device responds, the server compares the responses, and he can authenticate the device.

This technique avoids the use of cryptographic keys to authenticate devices and requires little memory to store a model. However, the distribution of these models is a problem equal to the distribution of cryptographic keys. Also, it does not offer non-repudiation since the server can steal the device identity using the extracted model.

Establishing Trusted Identities in Disconnected Edge Environments Computing [24] This work presents an architecture that provides authentication of entities to a cloudlet in a disconnected environment; it addresses a special type of cloudlets, named *tactical cloudlets*, that are built to operate mostly offline. Their main goal is to give support for first responders, search and rescue teams, military personnel, and others operating in crisis environments. In this architecture, a cloudlet is separated into two components: a server and a radius server. The server is responsible for all the computation while the radius server is responsible for authenticating devices so that these can use server resources.

The trust is established by the following steps:

1) Bootstrapping Process- Every deployment of a tactical cloudlet starts with a clean state, generates a new pair of private and public keys for the cloudlet and sets a deployment duration.

2) Pairing Process- Where the cloudlet generates the device credentials. First, the cloudlet administrator connects the device via USB or Bluetooth to the server. Then the server uses IBE to generate a private and public key for the device, using the device ID (android device ID) and a BLS certificate (BLS

certificate is the device public key signed by the server's public and private keys). Afterwards, the server sends these three elements plus the radius server certificate to the device and deletes the device's private key locally.

3) WiFi Authentication Process- The client establishes a TLS connection with the server. This connection requires the authentication of the radius server. The device can verify if the radius server has the correspondent private key from the certificated received during pairing. Then, through the encrypted channel, the device sends its credentials Public key and BLS certificate as username: password. The radius server checks these credentials and if it succeeds allows the device to use the server's WiFi network.

4) API Request Process- The service is provided using http request/response protocol. To encrypt the communication the device can use any of the previous cryptographic elements or generate a new one and share it with the server.

5)Automatic and Manual Device Credential Revocation- The cloudlet will refuse connections after the deployment duration, and a cloudlet administrator can manually delete a device from the cloudlet revoking his credentials.

To authenticate devices in an offline environment, this architecture assumes that a single cloudlet is fully trusted which is something that is not practical in our model. The BLS [25] certificate uses short signatures that are half the size of a DSA (Digital Signature Algorithm) signature for a similar level of security. This is something that we can use in our solution since devices can be resource constrained. It is particularly a good security idea to separate the authentication from the system.

Octopus: An Edge-Fog Mutual Authentication Scheme [26] This system allows any edge device and any fog node to authenticate each other mutually. In Octopus, the network model has three layers: first, a Registration Authority (RA) located at the cloud; second, a layer composed of multiple fog regions, each fog region is composed of multiple fog nodes; third, an edge layer where users are located.

This scheme is based on the RA generating a symmetric key for each pair of user fog node and distributing these keys among the fog nodes of the region where the user is located. This symmetric key can easily be generated by the user, using only a hash function. To establish a connection, both user and fog node need to prove they possess the shared secret by exchanging nonces and then establish a session key using this shared secret.

Registration Phase: The first time a user connects, he must register his identity in the RA. Since the RA is in the cloud, it is assumed that the user can establish a secure connection (example using PKI). Then RA generates and stores a long-lived random master secret key for the user and sends him this secret. Then, the RA computes a key for each fog nodes: $k_{FN}^{U} = H(ID_F, ID_{FN}, S_U)$, where H is a hash function, ID_F is the fog region identifier, ID_{FN} the fog node identifier and S_U is the user master secret. After it, the RA sends each key to the corresponding fog node in the region encrypted with the fog node public key. Each fog node upon receiving, stores $\langle ID_U, k^U_{FN} \rangle$, where ID_U is the user identifier and k^U_{FN} the correspondent key.

Authentication Phase: When a user connects to a fog node it sends the following message $\langle HelloFog, ID_U, r_U \rangle$, where r_U is a random nonce. Then the fog node also generates a random nonce r_{FN} and encrypts both nonces using the symmetric key correspondent to the users, received in the previous phase, $E(k^U_{FN}, (r_U, r_{FN}))$, where E is a symmetric key encryption function that receives a key and a text as parameters. This ciphertext is sent to the user, then the user computes k^U_{FN}, decrypts and checks validity of r_U. Since only an authorized fog node can have k^U_{FN} to generate such a message, the fog node is authenticated. Next, the user generates a session key k_s and sends $E(k^U_{FN}, (r_{FN}, k_s))$ to the fog node. Finally the fog node decrypts and checks validity of r_{FN}, if the r_{FN} is correct the fog node accepts k_s as a session key.

Octopus is efficient in the user's side since the user only uses symmetric keys and stores a single master secret key. The user only needs to communicate with the cloud if he moves to a different fog region. Otherwise, he can locally generate the key to communicate with a fog node, offering a low latency when migrating in a single region. However, in this scheme fog nodes are required to compute a cryptographic function at the beginning of the *Authentication Phase*, which allows the fog node to suffer from Denial of Service from authenticated devices. Also, fog nodes store a key for each user in a region even if they never communicate with that user.

4.2.2 Secure Communication

After an edge device and a fog node establish mutual authentication, they require secure communication with integrity, confidentiality and non-repudiation, for a device using the fog node resources. In the fowling paragraphs we describe multiple research efforts aimed at allowing a remote entity to process information from edge devices in a secure way. First, we present a system that shares symmetric keys with middleboxes allowing them to perform specific operations. Then we present a protocol to propagate messages using IBE.

Multi-Context TLS (mcTLS) [27] In today's Internet, HTTPS is widely used to protect communications between two entities. HTTPS is based on TLS handshake; this scheme allows the server to authenticate himself to the client through his certificate and establish a symmetric key between them. Companies and business are implementing middleboxes that are located in between the client and the server. These middleboxes have the job for Cache, Compression, IDS, Parental Filter, among others. A middlebox is seen as a fog node because it can, for example, aggregate and process information from a sensor and only pass the message to the cloud if a piece of relevant information appears.

As communication is encrypted it makes it harder for middleboxes to do their job and mcTLS has developed a solution that allows middleboxes to read and change messages while keeping confidentiality between endpoints.

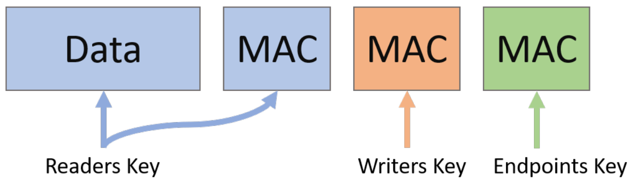

Fig. 2. Message in mcTLS.

This solution is based on using multiple cryptographic keys; each key offers a capability over a message. One key for reading a message, a second to change the content of the message, and a third to verify if a message was modified. A message comes along with 3 MACs, signed with each key. The data comes encrypted using the readers key, as shown in Figure 2.

A *reader* only owns the readers key. He can decipher and read the data of a message and verify the *MAC* correspondent to his key. A writer owns the readers and writers key; he can read and modify the data. If he alters the data, he must compute and replace two *MAC*s, the reader's and the writer's *MAC*s. The *endpoints* are the client and the server; they share the endpoint key and own all other keys and they have an extra *MAC* to detect if any middlebox changes the message.

These keys are distributed by the endpoints. A middlebox only has access to a specific key if both endpoints agree on it. Then they send the key back to that middlebox, allowing it to do its job. This allows a more fine-grained access control over a message. However, this scheme still requires that all communications pass through the cloud having high latency, something that we want to avoid by processing data in the edge. If the client establishes multiple connections, he has to store many keys for each of them, which makes it difficult to manage the keys. In this system, integrity is ensured by using MACs, confidentiality is offered by the readers key, and non-repudiation does not exist, since multiple parties possess the same key.

Privacy and Confidentiality in Context-Based and Epidemic Forwarding [28] Normally protocols that disseminate information through nodes in a network require these nodes to know the exact identity of the destination and, in most cases, they must exchange public keys. In Context-based forwarding [28] a node can send a message by only knowing partial information of the destination, its context, and without any key exchange. Additionally, the message has privacy and confidentiality.

The destination can be a single entity or a group. Thus destination is defined as a set of attributes such as location, workplace, name, organization. For each of these attributes, an entity receives a private key from a trusted third party. In this design, a message is separated in header and content.

The *content* is encrypted with the destination‘s public key using Identity-Based Encryption, however, the string used to generate the public key is a concatenation of the destination attributes.

The *header* is composed of all the destination attributes, to identify the destiny of a message. These attributes are encrypted using Public Encryption with Keyword Search (PEKS) to ensure privacy. PEKS allows a node to verify if some keyword exists in a ciphertext, without reading its content. This way, nodes can compare their attributes against the PEKS in the header maintaining the destiny privacy. When a node shares all attributes of a header, then he can decipher the content.

This architecture offers an accessible manner to manage the system keys taking advantage of IBE. It is also interesting the idea that a message can be sent through the network without the need to know the destination, only that it has a set of attributes. At the edge due to a large number of fog nodes, it is interesting to be able to send a message to another fog node without knowing its entity. Thus avoiding having to identify all the nodes that replicate a particular content.

In this system, confidentiality is provided by using a key which is the concatenations of the destination attributes. Integrity and non-repudiation are not guaranteed since multiple entities can generate the same messages if they possess the same attributes. But if the IBE public key is generated using a unique identifier, we can send the message with a hash encrypted with the private key of the origin, offering integrity and non-repudiation.

4.2.3 Untrusted Nodes

Fog nodes or cloudlets are devices in the fog layer that have significant computing resources, low latency to the (local) edge devices, and are well connected to the network. These fog nodes are in physical locations that make them more vulnerable to tampering. Thus, to achieve the goals in Section 2 we must consider that these nodes may become malicious. In our work clients will perform updates in a fog node and it will be responsible for generating metadata to ensure causal consistency in the system. If a fog node becomes rogue, he could swap the order of the updates or generate wrong metadata, subsequently, breaking the causal consistency.

Here, we introduce two different ideas to avoid malicious fog nodes:

1) Using a quorum protocol to maintain availability in the occurrence of a malicious node, thus we present a system that uses quorums to deal with Byzantine faults. One possible behavior of a byzantine node is dropping messages, a quorum protocol allows to hide this kind of behavior.

2) Using Trusted Execution Environments (TEE). Due to the recent focus on fog computing applying these TEEs on the fog would be a natural choice to secure computation and sensitive data [21]. Therefore, we present a specific example of a system using Intel SGX.

Minimal Byzantine Storage [29] Malicious fog nodes can be seen as servers in the presence of a Byzantine fault. A Byzantine fault is the loss of the correct behaviour of the system and can result in an arbitrary behavior, even one that can be considered malicious. In the topic of storage systems research can be found on designing systems that can tolerate f byzantine faults. Usually, these systems rely on a *quorum*, which is a subset of servers in a system that has a shared variable. Clients perform read and write operations only on one quorum of servers. The number of servers is a crucial metric since, in the presence of a failure in a quorum, the rest of correct servers must ensure availability for the clients.

An example of such system is Minimal Byzantine Storage [29], they present Small Byzantine Quorums with Listeners (SBQ-L). This protocol requires $3f + 1$ servers to tolerate up to f byzantine faults. They offer atomic semantics, a guarantee that the sequence values read by any given client are consistent with the global serialization. This is possible by using the “Listeners” pattern that detects and resolves ordering ambiguities created by concurrent accesses to the system. Next, we show the size of the write quorum Qw and the read quorum Qr, where n is the number of servers.

$$Qw = \frac{n + f + 1}{2}, Qr = \frac{n + 3f + 1}{2}$$

When a client performs a *write*, first he queries a quorum of serves Qw to determine the new timestamp and this timestamp must be higher than all previously seen by the quorum. Then he sends the data and the new timestamp to all servers and waits for acknowledgments from the Qw.

To perform a *read*, the client queries a quorum of servers Qr and waits for replies. He may receive more than one message from a server if writes are in progress. The client will maintain the replies until Qw servers agree on the same data and timestamp.

A strong aspect of this work is that it relies on asynchronous communication and offers a protocol that maintains system availability in the case of a server losing his correct behavior. In particular if a server drops messages instead of propagating updates. A client has visibility when a server is misbehaving; he could inform the system if that is the case. However, this requires a few messages to be exchanged at every operation penalizing the system latency.

Harpocrates: Giving Out Your Secrets and Keeping Them Too [30] Content Distribution Networks (CDNs) is a geographically distributed network of proxy servers, which is located close to the users at the edge, and host static content. The CDNs offer low latency and high scalability like fog computing and are already used for applications that provide static content, such as images and videos to users. However, providers of such CDNs are trying to evolve to fog computing and are implementing computation inside the CDN servers and one example of it is “Cloudflare CDN”.

Many CDNs work as a proxy for the origin web server, working as "man in the middle." When a client does a DNS lookup for some server, the result may be the IP of a CDN server instead of the intended server. This technique hides the origin server IP and is reliable protection against DoS. However, for a CDN server to authenticate towards a user, it requires keys or other types of sensitive information, and the owners of applications may not fully trust the CDNs providers. To tackle this issue, Harpocrates uses trusted execution environment, Intel, Software Guard Extensions (Intel SGX).

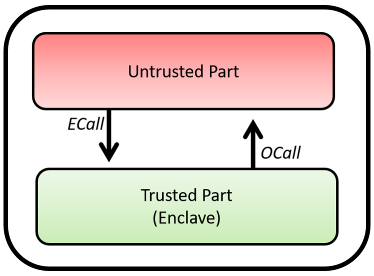

Fig. 3. An Application in Intel SGX.

Intel SGX is an extension of the processor architecture and enables applications to use a TEE, known as *enclave*. Applications must be built in two parts: an untrusted part and a trusted part, Figure 3. The trusted part will run inside the enclave, where the code and data have integrity and confidentiality. The untrusted part can make an Enclave Call (ECALL) to switch into the enclave and start the trusted execution. The CPU verifies the integrity of the code and data of the trusted part and executes. The code inside the enclave makes an Out Call (OCALL) and returns to the untrusted part. When leaving the enclave, the CPU encrypts the data and stores a hash of the data. Another important feature of the SGX is the *attestation* that allows a client to execute a remote call over the CPU to have a proof that he is communicating with the specific code in a real SGX enclave, and not an impostor.

In Harpocrates the main goal is to serve client requests at the edge, while keeping secure the origin master secret. This secret is needed to authenticate the server. The solution is to store the master secret inside the enclave. Each enclave has an initialization function that will contact the origin server. The origin server will use remote attestation to authenticate the code, and the SGX then will send the master secret through this secure channel. This allows the CDN proxy to verify cookies and read messages content, without direct access to the master secret.

The protected memory size in an enclave can contain tens of megabytes, typically 64 MB or 128 MB [31]. Therefore, it is essential to minimize the memory usage inside the enclave. In particular, the use of more memory also increases the swap time from enclave and out. While attacks against SGX like Foreshadow [32] exist, Intel continues to investigate how to mitigate these issues. However, SGX could be implemented inside fog nodes, offering a trust base for the edge devices. In our work, such guarantees could help to secure cryptographic keys or execute secure functions.

4.3 Discussion

In this section, we discuss the previously presented work, especially the ideas presented in the related work that can help us to reach a solution. We start by looking at four tables (Table 1, 2, 3, and 4) that summarize this information, then we focus on answering the questions presented at the beginning of Section 4.

In order to apply security to a data store on the edge first we need to define what type of data store makes sense at the edge. In Table 1 we can see what type of features are essential for this kind of data store. It should be noted that although Gesto is designed to support edge computing its topology is not scalable as the tree topology in Saturn.

Tables 2, 3, and 4 provide an overview of the problems the surveyed systems try to solve, the technologies they use, and how these technologies can help us achieve our goals. Looking at Table 2 we have an overview of what kind of challenges each system is able to solve in their own context. In Table 3 it is visible which technologies are used by these systems to achieve their solutions. Finally, Table 4 compares which of these technologies may be useful to reach our objectives in the context of a data store on the edge.

Systems	Server Topology	Edge Support	Client Controls his Causal Past	Operations	Metadata
Saturn	Tree	—	—	W,R,M	Scalar O(1)
COPS	—	—	yes	W,R	Graph O(K)
Gesto	Star	yes	—	W,R,M	two Scalars O(1)

Table 1. Comparison of the different data store solutions. K is the number of all data objects in the system. W, R and M represent write operation, read operation, and migration operation, respectively.

– **Architecture** An important factor in our work is to identify which communication channel may exist in such systems. COPS [10] do not have a particular architecture to propagate messages. Gesto [7] uses a star topology in a region. If a region has many cloudlets, then the broker can become a bottleneck. In Saturn [9], messages are propagated in a tree topology, which is a scalable and efficient way to propagate metadata.

Systems	**Authentication**	**Integrity**	**Confidentiality**	**NR**	**Tolerant to Malicious Nodes**	**Efficient for the Client**
PUF	yes	—	—	—	—	yes
Tactical Cloudlet	yes	yes	yes	yes	—	—
Octopus	yes	—	yes	—	—	yes
mcTLS	yes	yes	yes	—	—	—
Context-Based Forwarding	yes	—	yes	—	—	—
MinByz	—	—	—	—	yes	—
Harpocrates	yes	—	—	—	yes	yes

Table 2. Concerns address by the surveyed systems. NR is non-repudiation of the message origin.

Systems	**Symmetric Keys**	**Model**	**IBE**	**PKI**	**TLS**	**BLS**	**Quorums**	**Intel SGX**
PUF	—	yes	—	—	—	—	—	—
Tactical Cloudlet	—	—	yes	—	yes	yes	—	—
Octopus	yes	—	—	yes	—	—	—	—
mcTLS	yes	—	—	—	yes	—	—	—
Context-Based Forwarding	—	—	yes	—	—	—	—	—
MinByz	—	—	—	—	—	—	yes	—
Harpocrates	—	—	—	—	—	—	—	yes

Table 3. Techniques used by the surveyed systems.

Techniques	**Authentication**	**Integrity**	**Confidentiality**	**Device NR**	**Cloudlet NR**	**Updates Availability**	**Order Integrity**	**Efficient for the Client**
Symmetric Keys	yes	yes	yes	—	—	—	—	yes
Model	yes	—	—	—	—	—	—	yes
IBE	yes	yes	—	yes	yes	—	—	—
PKI	yes	—	—	yes	yes	—	—	—
TLS	yes	—	yes	yes	yes	—	—	—
BLS	yes	yes	—	—	—	—	—	yes
Quorums	—	—	—	—	—	yes	yes	—
Intel SGX	yes	—	—	—	yes	—	yes	yes

Table 4. Concerns of our system that can be solved using the previously presented techniques. NR is non-repudiation of the message origin.

- **Operations** Like in Saturn, COPS, and Gesto, clients can execute read and write over data in a server. Clients are also able to migrate between replicas.
- **Messages** All these systems have in common the decoupling of metadata from data, associated with an update. Also, metadata is the most relevant part of the system, and for this reason, we look for ways to secure these metadata. Metadata size is an important factor to reach better performance in a data store. Saturn and Gesto offer relatively small size metadata, due to only propagating scalars as timestamp and unique identifiers. However, COPS send a dependency graph in every metadata, increasing the size considerably. As a result of this trade-off, COPS has less false dependencies than Saturn and Gesto.
- **Authentication** Our solution requires mutual authentication from the client and the cloudlet. Clients need to be authenticated to check if such entity has the authorization to operate over some data. Cloudlets must authenticate as well, avoiding a client sending confidential data to an untrusted party impersonating a cloudlet. Octopus [26] and the PUF scheme [23] are lightweight schemes, but they do not offer non-repudiation over cloudlet messages. In the concept of tactical cloudlet [24], they assume the cloudlet is trusted and clients have physical access to establish a key exchange, something not practical for our solution. On the other hand, they use BLS certificates. We could use BLS signatures to generate MACs. In tactical cloudlet [24] they also use IBE to generate a key pair for a client. We can use IBE to generate a key pair for every client and cloudlet. However, a PKG is necessary and it can be placed in the cloud. The clients must interact with the cloud, but only once. So they can receive their private key, then they can authenticate directly to the cloudlet.
- **Integrity** In mcTLS [27] there are several MACs however in our systems the fog nodes must propagate messages and not modify them. So, we just need a MAC signed by the origin fog node.
- **Confidentiality** Using a symmetric key like in mcTLS is impractical because it requires huge management to distribute so many keys. Using something like context-based forwarding [28] implied that all the fog nodes shared the same attributes, so it would be easier to share a symmetric key between all the fog nodes.
- **Non-Repudiation** In mcTLS [27] the keys are always shared by more than one entity, so it is impossible to have Non-Repudiation. In context-based forwarding [28] we can generate private keys for each entity based on a unique identifier instead of attributes, and MACs can be signed using this key offering non-repudiation.
- **Availability** Our system must maintain a similar level of availability as the data stores in Section 4.1.
- **Untrusted Nodes** Another important part of our work is how to detect and avoid malicious nodes from generating harmful metadata that can leave the system in a faulty state. We mention two ways to do this: the first were quorums [29], requiring a device to contact multiple cloudlets for each update. If a cloudlet replies/sends an incorrect message, the client could

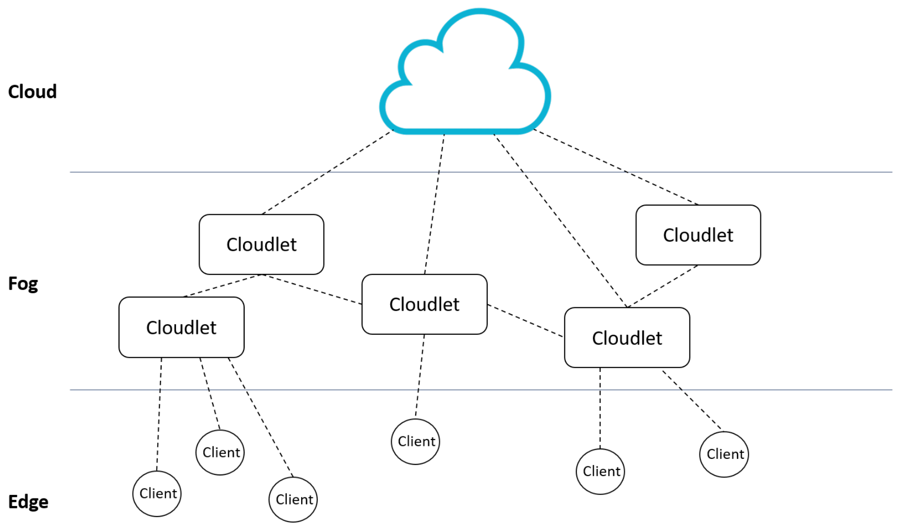

Fig. 4. Communication channels.

notify the cloud for such cloudlet. This solution requires multiple connections that can be a problem for devices with limited resources; the second solution was using TEE, such as Intel SGX [30], a solution where clients can have a base of trust in cloudlets. However, enclaves possess small memory size and using this in our solution requires techniques to avoid this low memory issue.

As discussed, none of the previous work can fully address our challenges, and therefore we choose the best of each system to build one that can fulfill our goals. In the next section we present the architecture that a data store should have at the edge. Then we present our solution based on our discussion.

5 Architecture

In our scheme clients are edge devices and execute read and write operations on a cloudlet, as illustrated in Figure 4. Cloudlets communicate between each other and with the cloud, in order to synchronize updates. Each cloudlet maintains replicas of data objects, the distribution and managing of such data is a responsibility of the data store.

Our goal is to maintain the functionality of the system in the eventuality the cloudlet becomes rogue. We assume clients are trusted. We also assume that each cloudlet has a processor with the Intel SGX extensions, as depicted in Figure 5. For the cryptographic scheme we chose IBE which implies that both clients and cloudlets have private keys, in particular, the private key of the cloudlet is inside the enclave.

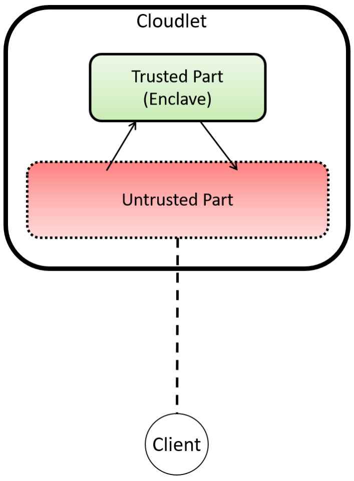

Fig. 5. Cloudlet architecture using the Intel SGX extension.

5.1 System Threats

In a data storage system located on the edge, the data is replicated by several cloudlets, subsequently clients read and write over the data stored in a cloudlet. If the data store ensures causality, cloudlets are responsible for generating metadata for each update. This metadata allows remote cloudlets to apply these updates in causal order. As mentioned before these cloudlets can be attacked, therefore they stop working correctly. Our work will focus on avoiding malicious cloudlets that can generate metadata in a way that causes the system to fail.

Our system has three entities: clients that are on the edge, cloudlets in the fog, and the cloud, shown in Figure 4. The cloudlet is the only untrusted entity. Clients and cloud are assumed to be reliable and will always behave correctly.

The causal consistency guarantee of a data store is highly dependent on the correct generation of metadata. On systems like Saturn [9], updates are serialized locally, and a server generates metadata representing that order. So, we can assume that a cloudlet has the responsibility of generating metadata that respects the order in which it applies the updates. These updates must be applied respecting causality. If the cloudlet is compromised, it could simply change the order of two updates that are causally dependent, to break the causality guarantee of the data store.

Metadata is also propagated through multiple cloudlets. It is necessary to prevent a cloudlet from modifying metadata without being detected. Additionally, it is essential to guarantee that the metadata generation for each update is correct and respects the causal order. However, a client that is connected to a single cloudlet cannot verify if the cloudlet is generating and propagating the correct metadata. Cloudlets that are receiving this metadata also do not know the order in which the source cloudlet received them, so they can not verify too if the order is correct. This problem is directly related to the fact that the only entity that knows the correct order by which the updates were applied is the origin cloudlet itself. Additionally, a cloudlet is also responsible for checking a client's past before accepting an operation to ensure that their data respects the client's causal past.

5.2 Solution

In our solution, we use Identity-Based Encryption scheme to generate cryptography keys, and cloudlets possess Intel SGX extension to offer a base of trust. In this section, we describe how we will achieve the goals defined in Section 2.

We first describe how to guarantee the properties in Section 3, such as Authentication, Data integrity, Non-Repudiation, and Availability. Then, we explain how our solution can deal with untrusted cloudlets.

Authentication Entities authenticate using private key that they acquire in the PKG. PKG is security located in the cloud. For a cloudlet to acquire his its private key, it has to offer an attestation proof to the PKG. On the other hand, a client can connect via HTTPS and authenticate to the PKG, like in the cloud computing model. After verifying the credentials, the PKG generates and sends the respective private keys. We need to send the private key through the network because, before booting, the Intel SGX contains all data in plain text, including the enclave data, making it impossible to place a private key in cloudlet bootstrap.

Data Integrity All messages in the system are accompanied by a MAC, which is signed using the entity's private key. Therefore, no entity outside the system can change a message without being detected. When a message is propagated between cloudlets an intermediate cannot modified it, because the enclave contains the private key, and only generates metadata from updates that come from clients.

Confidentiality Metadata must be visible to all entities in the system, so confidentiality is not an important property to ensure the correct operation of the system. However, it is possible to offer confidentiality of the messages. If the cloudlets are organized in a tree like the servers in Saturn, we can use Tree-based group key agreement [33] to establish a key between all the cloudlets and encrypt the metadata with this key.

Non-Repudiation Since every message has a MAC signed with the source private key, it is ensured that no other entity is responsible for generating such a message and the system only accepts messages that are correctly signed by a private key generated in the PKG.

Availability A cloudlet will accept requests from every authenticated and authorized client, meaning every client with a private key generated from the PKG can execute requests.

To guarantee that a cloudlet will always generate the correct metadata for each update, we use the Intel SGX extension. We take advantage of the enclave, every time a client makes a read or write operation. Clients will only accept a reply if it is signed with the private key of the cloudlet. This private key is stored inside the enclave. So, even if the untrusted zone is compromised it can never access this private key, therefore cannot generate wrong metadata. The enclave is responsible for knowing the most recent data value, and for generating the metadata.

However, the untrusted zone of the enclave can drop messages, it can reply correctly to a client and not propagate the respective metadata. If such an event occurs, the enclave will still be able to guarantee causal consistency, but the client updates will not be available elsewhere. To answer this issue it is required that the client contacts a second cloudlet and confirms that it has received updates. This solution is similar to the Minimal Byzantine Storage [29]; a client must contact a quorum of cloudlets to ensure that his updates are being propagated.

5.3 Optimizations

The Intel SGX enclave memory is limited, just a few hundred megabytes and a data store can hold thousands of objects. In addition, the clients can be devices that have few resources and the use of asymmetric keys can be costly. To deal with it, we consider additional measures for the two scenarios.

Enclave Memory So that the enclave can know the most recent value from a data object, it must store information for each object. This data must be stored in the untrusted zone. To avoid replay attacks, the enclave must store some information to guarantee the integrity of this data in the untrusted zone. Note that confidentiality is not necessary, only integrity.

Symmetric Key To avoid always performing operations with asymmetric metric keys, a client and an enclave can establish a secured channel based on a negotiated symmetric key. However, a private key is still needed for authentication towards the enclave and it will only be used to establish a symmetric key. Afterward, the client's MACs could be signed using this symmetric key.

6 Evaluation

Our main goal is to evaluate the overhead imposed by our solution, thus we will use similar metrics as in the previous system, such as Saturn [9] and Gesto [7]. We will evaluate our system in three different topics: 1) performance on the process of updates; 2) adaptability when new devices connect; 3) tolerance to malicious nodes.

6.1 Performance

As in many of the systems that offer causality on storage services, these systems use visibility, latency and throughput of remote updates as the main performance metrics. Thus, we will also measure this metric in this way, and discuss if the overhead is still accepted in a storage service in the edge.

Visibility latency can be defined as the difference between the update creation clock and the time this update becomes visible in a remote replica, while throughput is the number of updates that become visible in a replica over an amount of time.

6.2 Adaptability

Since the authentication phase usually has a significant computation penalty, we will measure the overhead that our system imposes when clients migrate between replicas. This is important due to the dynamism of the network at the fog and edge layer. We will measure the time it takes for a device to start using a fog node. It is important that after the authentication the entities no longer require the use of the cloud, a significant feature for services in the edge.

6.3 Tolerance to malicious nodes

To check the system's behavior in the presence of a malicious node, we simulate malicious nodes and measure the time the system takes to identify such node and revoke his identity. This malicious node can change the content of messages, try to break the causality of the storage service or drop messages.

7 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15 Deliver the MSc dissertation.

8 Conclusions

With the growth of resource constrained devices in the network periphery, known as edge devices, the need for cloud services also grows. With this number of devices, the amount of data being sent to the cloud will be too much for the current network bandwidth. Thus, arises a need for cloud resources closer to the edge, emerging the fog computing model.

Applications that take advantage of fog computing will require data consistency such as in the cloud model. However, the techniques used to maintain data consistency in the cloud are not easily transported to the fog computing. Currently, there is work on this type of systems such as the Gesto [7].

However, these systems are built on the basis that the data centers in the cloud are secure, focusing on performance and low latency requirements. In the fog model, we can no longer rely on secure servers, due to the exposed location of servers such as fog nodes, thus rising a need for edge security assurances.

In this work, we offer some security techniques to ensure the correct operation of storage applications with causal consistency at the edge. We focus mainly on the generation and propagation of network metadata in the occurrence of malicious fog nodes.

We leave for future work evaluating the overhead that the proposed solution brings to existing systems, and whether this overhead is acceptable for the low latency requirement that the fog computing model has.

Acknowledgments We are grateful to Manuel Bravo and Diogo Barradas for the fruitful discussions and comments during the preparation of this report. This work was partially supported by Fundação para a Ciência e Tecnologia (FCT) and Feder through the projects with references PTDC/EEI-COM/29271/2017 (Cosmos) and UID/ CEC/ 50021/ 2019.

References

1. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. In: Journal of Future Generation Computer Systems 25(6). (jun 2009)
2. Idex, G.: Cisco global cloud index: Forecast and methodology, 2016–2021. White Paper (mar 2016)
3. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the 1st Edition of the MCC Workshop on Mobile Cloud Computing, Helsinki, Finland (aug 2012)
4. Cisco: Cisco delivers vision of fog computing to accelerate value from billions of connected devices. press release. `https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=1334100` Accessed: 2018-11-15.
5. Zhang, J., Chen, B., Zhao, Y., Cheng, X., Hu, F.: Data security and privacy-preserving in edge computing paradigm: Survey and open issues. In: Journal of IEEE Access 6. (sep 2018)
6. Mukherjee, M., Matam, R., Shu, L., Maglaras, L., Ferrag, M.A., Choudhury, N., Kumar, V.: Security and privacy in fog computing: Challenges. In: Journal of IEEE Access 5. (sep 2017)
7. Afonso, N.: Mechanisms for providing causal consistency on edge computing. (nov 2018)

8. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. In: Journal of Communications of the ACM, 21(7). (jul 1978)
9. Bravo, M., Rodrigues, L., Van Roy, P.: Saturn: A distributed metadata service for causal consistency. In: Proceedings of the 12th European Conference on Computer Systems, Belgrade, Serbia (apr 2017)
10. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, Cascais, Portugal (oct 2011)
11. Akkoorath, D.D., Tomsic, A.Z., Bravo, M., Li, Z., Crain, T., Bieniusa, A., Preguiça, N., Shapiro, M.: Cure: Strong semantics meets high availability and low latency. In: Proceedings of IEEE 36th International Conference on Distributed Computing Systems, Nara, Japan (jun 2016)
12. Almeida, S., Leitão, J.a., Rodrigues, L.: Chainreaction: A causal+ consistent datastore based on chain replication. In: Proceedings of the 8th ACM European Conference on Computer Systems, Prague, Czech Republic (apr 2013)
13. Grid'5000. `https://www.grid5000.fr` Accessed: 2018-12-17.
14. Zhou, W., Jia, Y., Peng, A., Zhang, Y., Liu, P.: The effect of iot new features on security and privacy: New threats, existing solutions, and challenges yet to be solved. In: Journal of IEEE Internet of Things Journal. (jun 2018)
15. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, Portland, Oregon, USA (jul 2000)
16. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. In: Journal of Distributed Computing 9(1). (mar 1995)
17. LeBlanc, A.G., Chaput, J.P.: Pokémon go: A game changer for the physical inactivity crisis? In: Journal of Preventive Medicine 101. (aug 2017)
18. Housley, R., Ford, W., Polk, W., Solo, D.: Internet x. 509 public key infrastructure certificate and crl profile. Technical report (1998)
19. Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. In: Proceedings of 21st Annual International Cryptology Conference, Berlin, Heidelberg (aug 2001)
20. Ekberg, J.E., Kostiainen, K., Asokan, N.: The untapped potential of trusted execution environments on mobile devices. In: Journal of IEEE Security & Privacy 12(4). (jul 2014)
21. Ning, Z., Liao, J., Zhang, F., Shi, W.: Preliminary study of trusted execution environments on heterogeneous edge platforms. In: Proceedings of the 1st ACM/IEEE Workshop on Security and Privacy in Edge Computing, Bellevue, WA, USA (oct 2018)
22. Corporation, I.: Intel's fog reference design overview. `https://www.intel.com/content/www/us/en/internet-of-things/fog-reference-design-overview.html` Accessed: 2018-12-16.
23. Ning, Z., Liao, J., Zhang, F., Shi, W.: Preliminary study of trusted execution environments on heterogeneous edge platforms. In: Proceedings of the 1st ACM/IEEE Workshop on Security and Privacy in Edge Computing, Bellevue, WA, USA (oct 2018)
24. Echeverría, S., Klinedinst, D., Williams, K., A. Lewis, G.: Establishing trusted identities in disconnected edge environments. In: Proceedings of the 1st IEEE/ACM Symposium on Edge Computing, Washington, DC, USA (oct 2016)

25. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: Proceedings of International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia (dec 2001)
26. Ibrahim, M.H.: Octopus: An edge-fog mutual authentication scheme. In: Journal of IJ Network Security 18(6). (nov 2016)
27. Naylor, D., Schomp, K., Varvello, M., Leontiadis, I., Blackburn, J., López, D.R., Papagiannaki, K., Rodriguez Rodriguez, P., Steenkiste, P.: Multi-context tls (mctls): Enabling secure in-network functionality in tls. In: Proceedings of ACM Conference on Special Interest Group on Data Communication, London, United Kingdom (aug 2015)
28. Shikfa, A., Onen, M., Molva, R.: Privacy in context-based and epidemic forwarding. In: Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks Workshops, Kos, Greece (jun 2009)
29. Martin, J.P., Alvisi, L., Dahlin, M.: Minimal byzantine storage. In: Proceedings of the 16th International Conference on Distributed Computing, Berlin, Heidelberg (jun 2002)
30. Ahmed, R., Zaheer, Z., Li, R., Ricci, R.: Harpocrates: Giving out your secrets and keeping them too. In: Proceedings of the 3rd ACM/IEEE Symposium on Edge Computing, Bellevue, WA, USA (oct 2018)
31. Corporation, I.: Intel(r) software guard extensions developer reference for linux* os. `https://download.01.org/intel-sgx/linux-2.3/docs/Intel_SGX_Developer_Reference_Linux_2.3_Open_Source.pdf` Accessed: 2018-12-10.
32. Bulck, J.V., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., F. Wenisch, T., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In: Proceedings of 27th USENIX Security Symposium, Baltimore, MD (aug 2018)
33. Kim, Y., Perrig, A., Tsudik, G.: Tree-based group key agreement. In: Journal of ACM Transactions on Information and System Security 7(1). (feb 2004)