# Omega: a Secure Event Ordering Service for the Edge

*(extended abstract of the MSc dissertation)*

Cláudio José Pereira Correia
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues and Professor Miguel Correia

***Abstract*—Edge computing is a paradigm that extends cloud computing with storage and processing capacity close to the edge of the network, with the aim of supporting latency sensitive applications such as augmented reality or mobile gaming. Edge computing is often materialized by using many fog servers placed in multiple geographic locations. Fog nodes are likely to be more vulnerable to tampering than nodes placed in large central data centers and, therefore, it is important to secure the functions they provide from malicious faults.**

**A key building block of many distributed applications is an ordering service, that is able to keep track of cause-effect dependencies among events and that allows events to be processed in an order that respects causality. In this thesis we present the design and implementation of a secure event ordering service for fog nodes. Our service, named Omega, leverages the availability of a Trusted Execution Environment (TEE), namely of the *Intel SGX* enclave, to offer to fog clients guarantees regarding the order by which events are applied and served, even when the fog nodes become compromised. To assess the performance of our techniques, we have built a key-value store that offers causal consistency for the edge that makes extensive use of Omega. Experimental results show that, despite the overhead associated with the use of the TEE, the ordering service can be secured without violating the latency constraints of time-sensistive edge applications.**

## I. Introduction

Cloud computing is a model for deploying Internet applications that allows companies to execute 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 demand. Cloud computing has been, therefore, widely adopted both by private and public services.

Despite its benefits, cloud computing has some limitations. The number of data centers that offer cloud services is relatively small, and they are typically located in a few central locations. For instance, Google currently maintains 16 data centers; and only 3 of these data centers are not located in North America or Europe. Thus, clients that operate far from these data centers may experience long latencies [1]. 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 or online games, that require low latencies when accessing the cloud. In fact, it is known that a response time below 5ms–30ms is typically required for many of these applications to be usable [2]. Also, most of these devices have sensors that produce enormous quantities of information that need to be collected and processed.

One solution to address the latency requirements of new edge applications is to process 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 individual servers, located closer to the edge. This concept is often named *fog computing* [3]. It assumes the existence of fog nodes that are located close to the edge. The number of fog nodes is expected to be several orders of magnitude larger than the number 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 exposed to tampering. Therefore, fog nodes are substantially more vulnerable to being compromised [4], and developers of applications and middleware for edge computing need to take security as a primary concern in the design.

In this paper, we address the problem of *securing a middleware service for edge computing*. Specifically, we focus on securing an *event ordering service* that is able to keep track of cause-effect dependencies among events and that allows events to be processed in an order that respects causality. The ability to keep track of causal relations among events is at the heart of distributed computing and, as such, an ordering service is a fundamental building block for many applications such as storage services [5], graph stores, social networks, online games, among others. The idea of providing an event ordering service is not new (a notable example of such a service is Kronos [6]) but, to the best of our knowledge, we are the first to address the problem of providing secure implementations that may be safely executed in fog nodes.

Our service, named *Omega*, leverages the wide availability of support for Trusted Execution Environments (TEE), namely of Intel SGX *enclaves*, to offer fog clients guarantees regarding the order by which events are applied and served, even when fog nodes become compromised. We take particular care to use lightweight cryptographic techniques to ensure data integrity while keeping a reasonable tradeoff with availability. A key goal is to secure the ordering service without violating the latency constraints imposed by time-sensitive edge applications. We achieve this by using enclaves only for a few important operations. In particular, applications run outside the TEE and use the enclave to selectively request proofs over the order of operations. Also, the interface of Omega is, as it will be discussed later, richer than that of services such as Kronos.

To illustrate the use of Omega, and also to assess its performance in practice, we have built a key-value store, named OmegaKV, that offers causal consistency[7]. OmegaKV is a secure extension of causal-consistent key-value stores that have been designed for the web, such as [5], [8]. We are particularly interested in extending key-value stores that offer causal consistency, since this is the strongest consistency model that can be enforced without risking blocking the system when network partitions or failures occur. Clients of OmegaKV can perform write and read operations on data replicated by fog nodes, and are provided with the guarantees that writes are applied in causal order and that reads are also served in an order that respects causality.

We experimentally assessed the performance of Omega using a combination of micro-benchmarks and its use to secure the metadata required by the OmegaKV key-value store. Our experimental results show that Omega introduces an additional latency of approximately 4ms, which is much smaller than the latency required to access central cloud data centers, and that, contrary to cloud based solutions, allows latency values in the 5ms-30ms range, as required by time-sensitive edge applications.

## II. Background and Related Work

### A. Edge Computing and Fog Nodes

The emergence of IoT and the stress it places on services that operate in the cloud motivates the use of computing resources close to the edge. Edge computing is a model of computation that aims at leveraging the capacity of edge nodes to save network bandwidth and provide results with low latency. However, many edge devices are resource constrained (in particular, those that run on batteries) and may benefit from the availability of small servers placed in the edge vicinity, a concept known as fog computing. Fog nodes provide computing and storage services to edge nodes with low latency, setting the ground deploying resource-eager latency-constrained applications, such as augmented reality.

### B. Securing Fog Services

The fact that fog nodes are dispersed among multiple geographic locations, close to the edge, increases the risk of being attacked and becoming malicious [4]. A compromised fog node may delete, copy, or alter operations requested by edge devices, causing information to be lost, leaked, or changed in such a way that it can lead the application to a faulty state. To address this challenge, one needs to resort to a combination of techniques, from which we highlight *replication* and *hardening*.

Replication consists in relying on multiple fog nodes instead of a single node. If enough fog nodes are used, it may be possible to mask arbitrary faults (often designated Byzantine faults) and, in some cases, to detect compromised nodes. Techniques such as Byzantine quorums [1] can be used for this purpose. Although they require contacting multiple fog nodes, this is the only way to ensure that critical information is not lost due to a compromised fog node, as such a node may become silent. Unfortunately, contacting and voting on the output of multiple fog nodes increases the latency of operations and may defeat the very purpose of fog computing. Therefore, we assume that many applications will be able to make progress while contacting a single fog node, specially is the fog node can execute quorum validations in the background and is hardened.

Hardening consists in using software and/or hardware mechanisms to reduce the ability of the adversary to compromise a device. Using the appropriate techniques it may be possible to prevent a compromised fog node from altering information unnoticed, effectively reducing the amount of damage an infected fog node can cause. A relevant mechanisms in this context is the use of a TEE, a secured execution environment with guarantees provided by the processor. The code that executes inside a TEE is logically isolated from the operating system (OS) and other processes, providing integrity and confidentiality, even if the OS is compromised. Therefore, the use of a TEE is a natural choice to secure computation and sensitive data in fog nodes.

*Intel Software Guard Extensions* (SGX) is a set of functionalities introduced in sixth generation Intel Core microprocessors that implement a form of TEEs named *enclaves* [9]. The potential benefits of this technology for the fog have already been recognized by Intel and it has already been used in practice. Applications designed to use SGX have two parts: an untrusted part and a trusted part. The trusted part runs inside the enclave, where the code and data have integrity and confidentiality; the untrusted part runs as a normal application. The untrusted part can make an Enclave Call (ECALL) to switch into the enclave and start the trusted execution. The opposite is also possible using an Outside call (OCALL). The SGX architecture implements a number of mechanisms to ensure the integrity of the code, including an *attestation* procedure that allows a client to get a proof that it is communicating with the specific code in a real SGX enclave, and not an impostor [10]. A limitation of current SGX implementations is that the protected memory region, named enclave page cache, is limited to 128 MB.

### C. Event Ordering

Most distributed applications need to keep track of the order of events. Different techniques can be used for this purpose, from synchronized physical clocks, logical Lamport clocks [7], vector clocks, hybrid clocks, and others. In most cases, the event ordering service is a core component of the application and if this service is compromised the correctness of the application can no longer be ensured.

In many cases, applications use their own technique to order events, so the implementation of the ordering service is intertwined with the application logic. This approach has two important drawbacks: first, it is hard to keep track of chains of related events across multiple applications. Second, it causes developers to maintain potentially complex code, that is duplicated in many slightly different variations, at different applications.

Kronos [6] was recently proposed as an alternative approach that consists in offering event ordering as a service and can be used by multiple applications, although it was designed for the cloud and does not implement security measures. In the context of edge computing, implementing the event ordering as a separate service that is provided by fog nodes makes it easier to harden the implementation, increasing the robustness of the applications that use such secured version of the service. In this paper we follow this path and describe the design and implementation of Omega, a secure event ordering service to be executed at fog nodes.

### D. Edge Storage

To unleash their full potential, fog nodes should not only provide processing capacity, but also cache data that may be frequently used; otherwise, the advantages of processing on the edge may be impaired by frequent remote data accesses. By using cached data, requests rarely need to be served by data centers. Consequently, a fundamental service of edge-assisted cloud computing is a storage service that extends the one offered by the cloud in a way that relevant data is replicated closer to the edge. Therefore, in this paper we also describe the implementation of a storage service to be provided by fog nodes, that we have named OmegaKV. This storage service extends key-value stores designed for the cloud that offer *causal consistency* [5]. This consistency criteria is particularly meaningful for edge computing, given that it was shown to be the strongest consistency criteria that can be offered without compromising availability.

Very recently, two key-value stores that leverages SGX have been proposed: ShieldStore [11] and Speicher [12]. Both were designed for the cloud, and they ensure data integrity outside the enclave by calculating a Merkle tree. However, both systems only implement a flat Merkel tree, ie with one level only. This implies that they do not take advantage of the logarithmic growth property it offers. Moreover, in both cases, the leaves of the Merkel trees have considerable size, which is not efficient as shown in Section VII-C. Pesos [12] is another secure object store that also takes advantage of SGX. Pesos was also built for the cloud and assumes a secure third party to persistently store the data, while OmegaKV stores the data locally in the untrusted part.

Needless to say, any storage service that offers causal consistency needs to keep track of the causal order relations among read and write operations. Instead of embedding such operations in the code of OmegaKV, our implementation makes extensive use of Omega. As a result, OmegaKV illustrates the benefits than can be achieved by having an event ordering service implemented at the fog level, and also shows how applications can leverage the fact that Omega is secured to harden their own behaviour.

## III. Violations of the Event Ordering

Before we describe the design and implementation of Omega, it is worth enumerating the problems that might occur if the event ordering service is compromised. In our solution, we assume that the event ordering service is executed in a single fog node and that the clients of the service are: edge nodes, servers in clouds, or other fog nodes. We also assume that clients are always non-faulty and we only address the implications of a faulty implementation of the event ordering service.

The detailed API of the Omega service will be described later in the text. For now, just assume that clients can: i) register events with the event ordering service in an order that respects causality and, ii) query the service to obtain a history of the events that have been registered. Typically, clients that query the event ordering service will be interested in obtaining a subset of the event history that matches the complete registered history (i.e., it has no gaps), and that is fresh (i.e., includes events up to the last registered event).

Informally, a faulty event ordering service can: i) Expose an event history that is incomplete (omitting one or multiple events from the history); ii) Expose an event history that depicts events in the wrong order, in particular, in an order that does not respect the cause-effect relations among those events; iii) Expose a history that is stale, by omitting all events subsequent to a given event in the past (that is falsely presented as the last event to have occurred); iv) Add false events, that have never been registered, at arbitrary points in the event history. These behaviours break the causal consistency and may leave applications in an unpredictable state.

Omega is able to prevent these attacks, ensuring data consistency at the edge despite fog nodes vulnerabilities. An important fact to note is that the Omega system does not prevent a fog node from refusing to communicate, i.e. a fog node can omit messages from a client. However, the Omega system allows a client to check if the node is omitting messages, allowing the client to migrate to another fog node and inform Omega about this malicious node. In the next paragraphs, we describe the Omega API that clients can use to avoid the attacks described above.

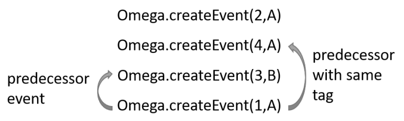

Figure 1. predecessorEvent and predecessorWithTag functions.

## IV. Omega Service

Omega is a secure event ordering service that runs in a fog node and that assigns logical timestamps to events in a way that these cannot be tampered with, even if the fog node has been compromised. Clients can ask Omega to assign logical timestamps to events they produce, and can use these logical timestamps to extract information regarding potential cause-effect relations among events. Furthermore, Omega keeps track of the last events that have been registered in the system and also keeps track of the predecessor of each event. These last features are relevant as they allow a client to check if the information provided by a fog node is fresh and complete (i.e, if a compromised fog node omits some events in the causal past of a client, the client can flag the fog node as faulty). More precisely, Omega establishes a linearization [13] of all timestamp requests it receives, defining a total order consistent with causality for all events occurring in the fog node.

### A. Omega API

The interface of the Omega service is depicted in Table I. Omega assigns, upon request, logical timestamps to application level events. Each event is assumed to have a unique identifier that is assigned by the client of the Omega service, so Omega is oblivious to the process of assigning identifiers to events, which is application-specific. Omega also allows the application to associate a given tag to each event. Again, Omega is oblivious to the way the application uses tags (tags can be associated to users, to keys in a key-value store, to event sources, etc.), but requires all tags to be registered before they are used (*registerTag*). The *createEvent* operation assigns a timestamp to a user event and returns an object of type *Event* that securely binds a logical timestamp to an event and a tag.

Clients are not required to know the internal format used by Omega to encode logical timestamps, which is encapsulated in an object of type *Event*. Instead, the client can use the remaining primitives in Omega to query the order of events and to explore the event linearization that has been defined by Omega. The primitive *orderEvents* receives two events and returns the oldest according to the linearization order. The client can also ask Omega for the last event that has been timestamped (*lastEvent*), or by the most recent event associated with a given tag (*lastEventWithTag*). Given a target event, the client can also obtain the event that is the immediate predecessor of the target in the linearization order (*predecessorEvent*), or the most recent predecessor that shares the same tag with the target (*predecessorWithTag*), as shown in Figure 1. Finally *getId* and *getTag* extract the application level event identifier and tag that have been securely bound with the target event.

| |
|---|
| *Register a tag with Omega*<br>void registerTag (EventTag tag) |
| *Create a timestamped event with a given identifier and a given tag*<br>Event createEvent (EventId id, EventTag tag) |
| *Order two events and return the first*<br>Event orderEvents (Event $e_1$, Event $e_2$) |
| *Return the last event timestamped by Omega*<br>Event lastEvent () |
| *Return the last timestamped event with a given tag*<br>Event lastEventWithTag (EventTag tag) |
| *Return immediate predecessor of a given event*<br>Event predecessorEvent (Event e) |
| *Return the most recent predecessor with the same tag*<br>Event predecessorWithTag (Event e) |
| *Return the application level identifier of an event*<br>EventId getId (Event e) |
| *Return the tag associated with an event*<br>EventTag getTag (Event e) |

Table I
The Omega API.

Note that, although Omega is inspired by services such as Kronos [6], it offers an interface that makes different tradeoffs. First, it allows clients to associate events with specific objects / tags and to fetch all previous events that have updated that specific object; Kronos instead requires clients to crawl the event history to get the previous version of a particular object. Second, Kronos requires the application to explicitly declare the cause-effect relations among objects. This is more versatile but more complex to use than Omega, that automatically defines a causal dependency among the last operation of a client and all operations that this client has performed or observed in its past. Finally, unlike Kronos, Omega automatically establishes a linearization of all operations, which simplifies the design of applications that need to fully order concurrent operations consistent with causality.

## V. Omega Implementation

In this section, we describe the design and implementation of the Omega service. We start by presenting the system architecture, the system model and the threats the system face. Then, we describe in detail the most important aspects of the implementatio.

### A. System Architecture

The Omega service is executed on fog nodes and is used by processes that run in the edge or in cloud data centers, as shown in Figure 2. Both the edge devices and the cloud can use Omega to create and read events on the fog node in a secure manner. For instance, edge devices can make updates to data stored on the fog node that are later shipped to the cloud (in this case, edge devices create events and the cloud reads them). Moreover, the cloud can receive updates from other locations and update the content of the fog node with new data that is subsequently read by the edge devices. For the operation of Omega, we do not need

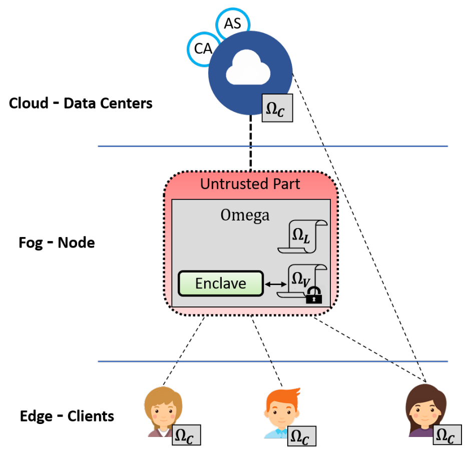

Figure 2. Omega architecture. CA is certification authority, AS is attestation server, $\Omega_C$ is Omega client, $\Omega_V$ is Omega Vault and $\Omega_L$ is Omega event log.

to distinguish processes running on the edge devices from processes running on the cloud, we simply denote them as *clients*. The method used by clients to obtain the address of fog nodes is orthogonal to the contribution of this document. We can simply assume that cloud nodes are aware of all fog nodes (via some registration procedure) and the edge devices can find fog nodes using a request to the Domain Name System (DNS), e.g., using a name associated with the application, or to the cloud, e.g., using an URL associated with the application.

The implementation of Omega assumes the existence of two external components, that are executed in the cloud and are assumed to be secure. These components are a *Certification Authority* (CA), that is used to generate public key certificates, and an *Attestation Server* (AS) , which is used when a fog node binds to the Omega implementation via a binding procedure (described in Section V-C). The techniques used to ensure the correctness of these two external components are orthogonal to this work (e.g. using standard Byzantine fault-tolerance techniques).

An important aspect of Omega is how to maintain the functionality of the system in case a fog node is compromised. To tackle this issue, Omega takes advantage of Intel SGX, as show in Figure 2; Omega generates all events inside the enclave, i.e., it executes *createEvent* operations inside the enclave. Moreover, all events take a digital signature obtained inside the enclave using the private key of the fog node, also stored inside the enclave. Omega includes the following modules: i) a protocol used by clients to ensure that they are interacting with the correct implementation of Omega running on the enclave and not with a compromised version of the same service (Section V-C); ii) two sub-components named vault and event log that are used to preserve the Omega state (Section V-D);

## B. Threat Model and Security Assumptions

The cloud and its services (AS, CA) are considered trustworthy, i.e., are assumed to fail only by crashing (essentially, we make the same assumptions as the related work [6], [5], [8]). Clients running on edge devices are also considered trustworthy and may also fail only by crashing.

Due to their exposed location, fog nodes can suffer numerous attacks and be compromised (an attacker might even gain physical access to a fog node). We assume that fog nodes may fail arbitrarily. They receive operations from clients and communicate with the cloud, so we assume that a faulty fog node can: modify the order of messages in the system; modify the content of messages; repeat messages (replay attack); tamper with stored data; and generate incorrect events. All these actions, if not addressed carefully, may lead the system to a faulty state, cause Omega to break the causal consistency of the events, and therefore affect the correctness of applications that use Omega.

We do not make assumptions about the security and timeliness of the communication, except that messages are eventually received by their recipient.

We also assume that each fog node has a processor with Intel SGX, which allows running a TEE designated enclave, as depicted in Figure 2. Both clients and fog nodes have asymmetric key pairs $(K_u, K_r)$. The private key of the fog node $K_r^F$ never leaves the enclave. For public key distribution, we consider the existence of a Public Key Infrastructure (PKI). We do the usual assumptions about the security of TEEs/enclaves (data executed/stored inside the enclave has integrity and confidentiality ensured) and cryptographic schemes (e.g., private keys are not disclosed, signatures cannot be created without the private key, and the hash function is collision-resistant). For obtaining digital signatures efficiently we use Elliptic Curve Cryptography (ECC), specifically the ECDSA algorithm with 256-bit keys, which is recommended by NIST. We assume the existence of a collision-resistant hash function. In practice we use SHA-256, also recommended by NIST. We use the implementations provided by the SGX SDK (inside the enclave) and Java (outside). Interestingly, this involves converting public keys from little endian (enclave) to big endian (Java).

## C. Client Binding

Before a client invokes any method of the Omega API,it has to execute a *client binding* procedure. The purpose of this procedure is to ensure that the client has the following guarantees: i) it has a secure connection to a software component; ii) this software component is running on an enclave in an Intel processor with SGX; iii) the software version is the same version as the one registered in Intel's attestation servers (which is assumed to be the correct version of the software component). This is also known as the attestation procedure. One limitation of the attestation procedure defined by Intel is that it involves multiple communication steps, including a connection to Intel servers (to ensure that the enclave is created on an

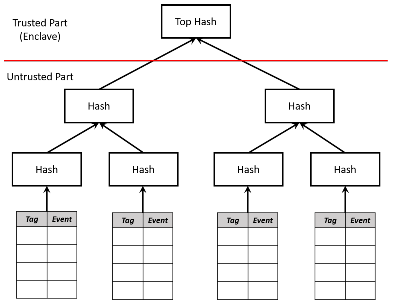

Figure 3. Merkle tree stored in the Omega vault in the untrusted zone of the fog node (with $N$ = 4).

Intel CPU). This is a cumbersome process which conflicts with our goal of improving the overall event-ordering service latency. Therefore, we have resorted to a different scheme to perform client binding. The Omega client binding protocol relies on the Attestation Server (AS) that runs in the cloud. The AS runs Intel's attestation protocol with each fog node. It performs this attestation periodically, with a period that can be configured. If the fog node passes the attestation, the AS obtains from the Certificate Authority (CA) a certificate with an expiration date lower than the period, digitally signed with its private key $K_r^{CA}$. The attestation performed by the AS allows to establish a secure connection with the enclave. The AS uses this connection to acquire the public key of the fog node, which is added to the previously mentioned certificate. This certificate is sent to the Omega instance running on the enclave of the fog node and stored in the untrusted part. Instead of running the Intel's attestation procedure Clients of the Omega service just ask the Omega implementation to return the certificate that has been issued by the AS.

### D. The Omega Vault and the Event Log

Omega is required to safely store different pieces of information, such as the private key associated with the certificate signed by the AS, the last event generated by Omega, and also the last event associated with each tag. However, the enclave memory is limited to a few tens of megabytes and Omega must keep an arbitrary number of tags. Therefore, Omega requires a way to securely store the above information (in particular the last event for an arbitrary number of tags). Also, Omega must have access to events it has generated in the past, given that clients can use the *predecessorEvent* method to crawl the event history.

To satisfy these requirements, Omega uses two storage services with different properties, the vault and the event log. In both cases, Omega stores events in the untrusted zone. These events can be in plain text but we still need integrity, i.e., to ensure that the untrusted zone cannot modify these values in case the fog node is compromised. Given that events are signed by Omega, the untrusted zone cannot modify individual events; however it can delete events or replace new events by older events. We now describe the implementation of these two services.

The *event log* is just a record of all events generated, so we opted to implement this component as a key-value store where events are stored using their unique identifier (assigned by the application) as key. Everytime Omega makes a look-up for a specific event (for instance, when a client crawls the event history) it simply checks the integrity of the event before the value is returned to the client. If an event cannot be found in the key-value store, this is a sign that the untrusted components of the fog node have been compromised.

The *vault* is harder to implement, because it needs to maintain the last event generated for each tag and to ensure that the untrusted components cannot replace the last event by an older event. Therefore, checking the integrity of the event returned is not enough: the Omega vault implementation must ensure that the values were not tampered. At the logical level, this is achieved by requiring the enclave to hash the vault every time it updates its content and to store the hash in the enclave itself. However, a naive implementation that would actually keep a single hash for the entire vault would not perform well because, as we have noted, the application may use a large number of tags and computing a hash of all these tags may take a long time. Also, it is not straightforward to ensure that the hash function yields the intended value if the values being hashed are to many to fit inside the enclave and may be changed by the adversary while the hash is being computed.

To address the problems above, the implementation of the Omega vault uses the following techniques. First, the content of the vault is stored as a *Merkle tree*. While conceptually the vault is just a table, maintained in the untrusted zone, where each line is a tag (index) and a column for the event (see Figure 3); in the implementation this table is splited into $N$ parts, and for each part, the enclave computes a hash to ensure integrity. Since the enclave may not have enough memory to store all these hashes, we use a Merkle tree such that the enclave only needs to store the top hash. All the hashes are calculated inside the enclave. In particular, SGX exhibits an attribute *user_check* that allows passing a pointer of the untrusted zone memory space as an argument in an ECALL, so that the enclave can access data that is in the untrusted part. This way, the enclave can verify and generate the Merkle tree hashes when needed in the untrusted zone requiring only to store the top hash.

When the enclave has to modify one part of the table it needs to: compute the Merkle tree to verify the data, then change the data and, finally, recalculate a few of the Merkle tree hashes (as many as the depth of the tree). These operations must be performed in an atomic manner, otherwise an attacker could change the table between the two Merkle tree calculations and the enclave would not be able to detect it. To ensure the atomicity of the combined operations, the enclave calculates the hashes in parallel, i.e., it calculates the old hash and the new hash of the table simultaneously so that in the end it can simply replace the old one.

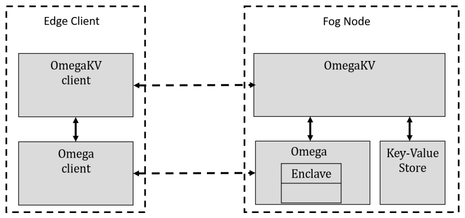

Figure 4. OmegaKV service components.

Besides, it should be noted that both the Merkle tree and the leaves are in memory. The enclave access this data in memory and modifies it in memory, the untrusted part is responsible for asynchronously write to disk all changes made by the enclave. This, unlike other systems prevents the enclave from having to access the disk, which has a latency penalty.

At the event that a fog node becomes malicious, the untrusted part can modify these values in memory and/or block the communication of the enclave with the outside. In this event, if a client tries to use the Omega service they will not receive any response as the enclave will not respond without having integrity from the data store in the untrusted part. This way the client can alert to the CA entity that is running in the cloud about a possible malicious fog node.

## VI. Omega Key-Value Store

OmegaKV is an extension to key-values stores that have been designed for the cloud. It makes it possible to maintain a cache of some key-value pairs in the untrusted space of a fog node while still ensuring that clients observe up-to-date values of the cached objects, in an order that respects causality. This is achieved by resorting to the services of Omega. OmegaKV also ensures that all updates performed by edge clients on the fog node, if they are propagated to the cloud, they are propagated to the cloud in an order that respects causality. As discussed in Section IV, Omega cannot ensure availability in case the adversary compromises the fog node. For availability, clients of OmegaKV should write on multiple fog nodes eagerly or cache the updates they have made and replay them later, if and only if they discover that the fog node has failed to propagate those updates to the cloud. We omit those details in this document, given that here we use OmegaKV mainly to illustrate the use of Omega and as a means to assess the overhead introduced by this service.

OmegaKV is implemented by combining an untrusted local key-value store and Omega. The key-value store resides in the untrusted region of the fog node, and it is used to store the values persistently. Omega is used to keep track of the relative order of update operations that have been performed locally. Figure 4 illustrates the architecture of OmegaKV. As with the Omega service, the implementation of OmegaKV has components that run on a client library and components that run of the fog node.

OmegaKV uses Omega as follows. Every update performed on the local replica is associated with an event generated by Omega. The keys used in the OmegaKV are associated to *EventTag*s in Omega; thus Omega will store securely each update performed on each key. Also, for each update operation, an *EventId* is generated as a function of the content of the update; more precisely, if a client writes value $v$ on some key $k$, that update will be identified by $\mathit{hash}(k \oplus v)$. The algorithms pseudocode of the OmegaKV is only available in the full thesis.

To *put* a value on the OmegaKV, the client starts by creating an identifier for the put operation by hashing the concatenation of the key and the value. Then it contacts Omega to serialize the update operation with regard to other update operations (in a serialization that respects causality). Finally, the server replaces the old value of the key with the new one. The event generated by Omega is stored locally with the update value. This can be used subsequently to ensure that clients see updates in the right order.

To perform the *get* operation, the client reads the value and the associated event from the local key-value store and queries Omega for the last event to be associated with the target key. Then it uses the hash of the value that has been safely stored by Omega and compares it with the hash of the value returned by the untrusted code running on the fog node. This allows the client to check that the untrusted zone has not been compromised and that the value returned is, in fact, the last value written on that key.

Finally, when the fog node ships the updates to the cloud, these are shipped together with events generated by Omega. This allows the cloud to apply the updates in the correct order in the master replica (and in other fog nodes, if needed).

## VII. Evaluation

The evaluation section is divided in two parts. In the first part we evaluate Omega in isolation. The goal is to offer a better understanding of the relative cost of the different components of the Omega implementation. In the second part we show the impact of using Omega to secure a concrete service, namely OmegaKV. The goal is to provide insights on the tradeoffs involved when executing services securely on the cloud, insecurely on fog nodes, or securely on fog nodes leveraging the services of Omega.

### A. Experimental Setup

In our experiments, the fog node is a dedicated computer with a 3.6GHz Intel i9-9900K CPU which has 16GB RAM (this processor supports SGX). The fog node OS is Ubuntu 18.04.2 LTS 64bit with Linux kernel 5.0.8. We run the Intel SGX SDK Linux 2.4 Release. The client machines are computers with 2.5GHz Intel i7-4710HQ CPU and 16GB RAM. Both the clients and the fog node are deployed in our laboratory, in the same network, emulating a 5G station communicating with a terminal (i.e., a 1-hop communication). Cloud services are executed on Amazon Elastic Compute Cloud (Amazon EC2), using the London data center, in t2.micro virtual machines.

The Intel SGX SDK and the code for the enclave are in C/C++. For simplifying the application implementation on Omega, we created a layer in Java over the part of Omega that operates in the enclave. For this we used Java 11 and the Java Native Interface (JNI) was used as a bridge between Java and C++. For persistent storage we used the key-value store Redis[14]. Redis is a typical key-value storage system, providing value storage indexed by keys. Redis is an in-memory system but offers a snapshot mechanism for data persistence. As a result of Omega abstraction, any key-value system could be used, we chose Redis because it is simple and lightweight to use. In the experiments we executed 5000 operations and discarded the first 500 and the last 500 for warming up and to remove potential outliers

## B. Omega Configuration and Performance

We first discuss how to configure the Merkle tree used by Omega since the performance of the service is highly dependent on this configuration. Then we provide an overview for the performance of Omega using the selected configuration for the Merkle tree.

## C. Merkle Tree Configuration

The Merkle tree used to store events is used on most of the Omega operations. Therefore, its correct configuration is key to the performance of the service. To understand how to configure the Merkle tree it is important to notice that any operation that involves checking/changing the content of the Omega vault requires to perform a number of computations that is a function of the size of the vault but also on the size selected for the Merkle tree leafs. More precisely, let $x$ be the size of each tree leaf and *VaultSize* be the maximum number of entries that the vault can store. Any operation on the vault must compute the hash of the affected leaf node and then the hashes of all inner nodes of the tree. Computing the hash of the leaf node has a cost that is linear with the leaf size. We denote this cost *leafHash(x)*. Since we have implemented the Merkle tree as a binary tree, updating/checking an inner node involves hashing two values. We denote the cost of computing the hash of an inner node *innerHash*. The number of inner nodes that need to be computed grows logarithmically with the size of the vault and its exact value is $\log_2(\frac{\mathit{VaultSize}}{x})$. Therefore, the formula that captures the cost of performing operation on the vault is $\mathit{leafHash}(x) + \mathit{innerHash} * \log_2(\frac{\mathit{VaultSize}}{x})$.

The formula above suggest that the optimal size of the leaf nodes of the Merkle should be very close to 1, given that the cost of hashing the leaf node grows linearly, while the cost of hashing the inner nodes grows logarithmically. However, it should be noted that there is a different cost of calculating a very large hash (*leafHash*) versus calculating many small hashes (*innerHash*). Figure 5 depicts the estimated cost of vault operation, on a vault of size 512 when the size of leaf nodes is varied from 1 to 512 entries. Note that when the leaf size is 1, the height of the Merkle tree is 9 and when the size of the leaf is 512 the entire vault is stored in a single leaf. The values in this figure were obtaining using the formula above, that was fed with results obtained experimentally for the parameters $\mathit{leafHash}(x)$ and *innerHash*. The values suggest that leafs should not be large; in this case, for a vault size of 512, the formula suggests that 8 is the best leaf size.

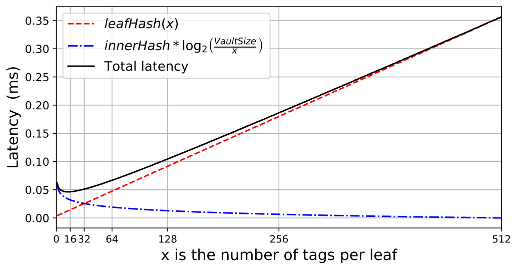

Figure 5. Estimated optimal leaf size of the Merkle tree for a vault of size 512.

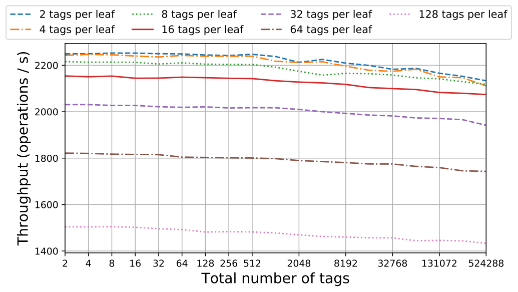

Figure 6. Actual performance of the Merkle tree as a function of the vault size and leaf size.

Based on this observation, we decided to run multiple experiments on the real system, where we measured the performance of the Omega vault implementation with different leaf sizes and different vault sizes. The results are depicted in Figure 6. As it can be observed, the best results are obtained for leaf sizes of 2 and 4 (in fact, the differences in performance for these two values is not significant) but quickly drops if larger leaves are used. Therefore, in all other experiments, we have used a leaf size of 2.

## D. Executing Omega Operations

We now present the results of a set of experiments that are aimed at assessing the performance of Omega when used in isolation. For this experiment we have measured the latency observed by a client when performing the most costing requests of the Omega interface. The *createEvent* operations involves modifying the Omega vault, which has a major latency penalty. We have measured how each software component that is executed in the client critical path contributes to the latency. The results are depicted in Figure 7.

Since the fog node is located one-hop away from the clients, the time spent in the network is not the main contributor to the latency observed by clients. The time lost from the Java layer to enclave is also small (from 1ms to 2ms). The time lost doing context switch is also

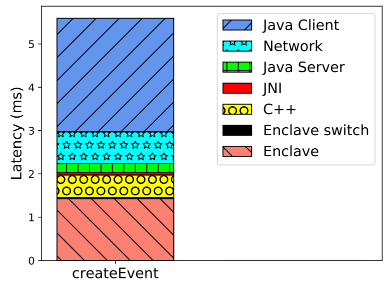

Figure 7. Operation latency for *createEvent*.

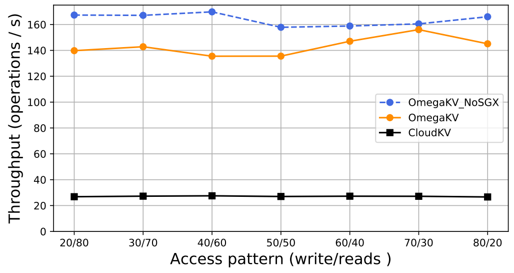

Figure 8. Access pattern throughput (writes/reads).

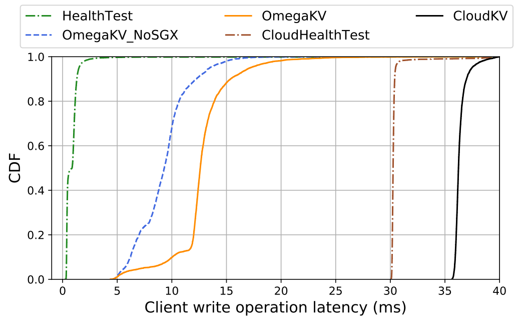

Figure 9. Write operation latency of a fog node and cloud.

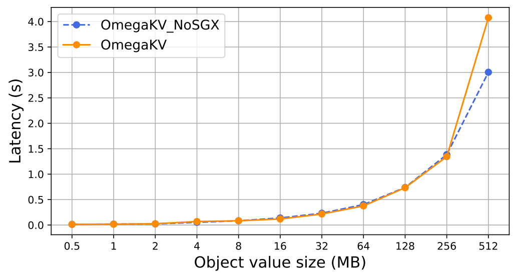

Figure 10. Write operation latencies w/ and w/o SGX.

considerably short, mainly because the enclave keeps very little state (taking advantage of the Omega Vault) and there is a small number of parameters passing in and out of the enclave. Thus, the main contributor to the latency are the cryptographic functions executed in the client and in the enclave. In the client, 2ms–2.5 ms are required to compute and verify digital signatures. On the server side, most of the time is also spent in the process of computing and verifying digital signatures.

## *E. Performance of the OmegaKV*

We now measure the impact of using Omega to make other services secure. For this purpose we compare the performance of OmegaKV, our Omega-based key-value store for the fog, with a similar non-secured service also running in the fog node (denoted OmegaKV_NoSGX), and with a version where security is achieved by running the service on the cloud (denoted CloudKV). All implementations of the key-value store have been developed in Java and use Redis [14] to keep their state persistent. Also, all system use messages that are cryptographically signed. The major difference among the implementations are that CloudKV and OmegaKV_NoSGX do not use the enclave (nor the Merkle tree used to implement the Omega Vault), they make no effort to verify date integrity, and they do not use JNI for interaction between Java layer and C++.

Figure 8 presents the maximum throughput that a client can achieve using the three systems. In the CloudKV implementation, the latency to the data center severely affects the throughput of the client; in our experiments the throughput of a cloud-based implementation is roughly 25% of the fog-based implementations. This was expected as one of the main motivations for using fog-nodes is to reduce the latency observed by clients. Interestingly, although the security mechanisms that are used in the Omega implementation introduced some amount of overhead (see the discussion in Section VII-B), this overhead is partially diluted when Omega is just a part of a larger system, that has many other sources of latency. In our experiments, OmegaKV offers a throughput that is approximately 18% smaller than the non-secured version of the same service but that is, nevertheless, much higher than the throughput supported by CloudKV.

Figure 9 compares the latency that a client experiences when using the services OmegaKV, OmegaKV_NoSGX, and CloudKV. For a better understanding of the graph, we use the ping operation to measure the round-trip time from the client to the fog node and to the cloud. This is shown as the HealthTest line for the fog node and the CloudHealthTest for the cloud. As expected the client can perform operations with much lower latency by using the fog node rather than using the CloudKV services that are in a data center, a reduction from 36ms to 12ms, close to 67%. OmegaKV has higher latency than OmegaKV_NoSGX, due to the use of the enclave. In absolute value we observe an increase in latency in the order of 4ms, which is non-negligible but still significantly smaller than the latency introduced by wide-area links. This allows OmegaKV to offer latency values in the 5ms–30ms range required by time-sensitive edge applications[2].

We also tested the performance of OmegaKV with different data sizes up to 512 MB (this is the maximum object size supported by Redis, our underlying persistent store). Results are shown in Figure 10. For this experiment we compared OmegaKV against OmegaKV_NoSGX. It is visible that our system follows the same latency as the traditional key-value store. This happens because, with large files, the overhead of the enclave and cryptographic operations becomes negligible when compared with the data transfer costs. It should be noted that OmegaKV transfers only one hash of the object to Omega; the object with tens of megabytes is stored in Redis.

## VIII. Conclusions

Fog computing can pave the way for the deployment of novel latency-sensitive applications for the edge, such as augmented reality. However, in order to fulfill its potential, we need to address the vulnerabilities that emerge when deploying a large set of servers on many different locations that cannot be physically secured with the same level of trust than cloud premises. This paper makes a step in this direction by describing the design and implementation of a middleware service that can be executed on fog nodes in a secure manner leveraging on the properties of trusted executions environments such as Intel SGX. In particular, we have proposed Omega, an event ordering service that can be used as a building block to build higher level abstractions. With the dual purpose of illustrating the use of Omega and of assessing its performance when used in practice, we have also designed and implemented OmegaKV, a causally consistent key-value store for the edge. Our evaluation shows that, despite the costs incurred with the use of the enclave mode to secure the Omega implementation, the use of Omega based applications can still provide much smaller latency and higher throughput than current cloud based solutions.

## Acknowledgments

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] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," *ACM Transactions on Storage*, vol. 9, no. 4, 2013.

[2] G. Ricart, "A city edge cloud with its economic and technical considerations," in *Proceedings of the International Workshop on Smart Edge Computing and Networking*, Kona, HI, USA, Jun. 2017.

[3] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the Workshop on Mobile Cloud Computing*, Helsinki, Finland, Aug. 2012.

[4] M. Mukherjee, R. Matam, L. Shu, L. Maglaras, M. A. Ferrag, N. Choudhury, and V. Kumar, "Security and privacy in fog computing: Challenges," *IEEE Access*, vol. 5, no. 6, 2017.

[5] M. Bravo, L. Rodrigues, and P. Van Roy, "Saturn: A distributed metadata service for causal consistency," in *Proceedings of the ACM European Conference on Computer Systems*, Belgrade, Serbia, Apr. 2017.

[6] R. Escriva, A. Dubey, B. Wong, and E. G. Sirer, "Kronos: The design and implementation of an event ordering service," in *Proceedings of the ACM European Conference on Computer Systems*, Amsterdam, The Netherlands, Apr. 2014.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, 1978.

[8] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

[9] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, Jun. 2013.

[10] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi, "Foundations of hardware-based attested computation and application to SGX," in *Proceedings of the IEEE European Symposium on Security and Privacy*, Saarbrücken, Germany, Mar. 2016.

[11] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "Shieldstore: Shielded in-memory key-value storage with SGX," in *Proceedings of the ACM European Conference on Computer Systems*, Dresden, Germany, Mar. 2019.

[12] M. Bailleu, J. Thalheim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani, "Speicher: Securing LSM-based key-value stores using shielded execution," in *Proceedings of the USENIX Conference on File and Storage Technologies)*, Boston, MA, USA, Feb. 2019.

[13] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, 1990.

[14] Redis, "Key-value store," http://redis.io, accessed: 2019-10-04.