Omega: a Secure Event Ordering Service for the Edge

Cláudio Correia, Miguel Correia, and Luís Rodrigues

Abstract—The edge computing paradigm extends cloud computing with storage and processing capacity close to the edge of the network, which can be materialized by using many fog nodes placed in multiple geographic locations. Fog nodes are likely to be vulnerable to tampering, so it is important to protect the functions they provide from attacks. A key building block of many distributed applications is an ordering service that keeps track of cause-effect dependencies among events and that allows events to be processed in an order that respects causality. This paper presents 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), based on SGX technology, to offer fog clients guarantees regarding the order in which events are applied and served, even when fog nodes are compromised. We have also built OmegaKV, a key-value store that uses Omega to offer causal consistency. Experimental results show that the ordering service can be secured without violating the latency constraints of time-sensitive edge applications, despite the overhead associated with using a TEE. Omega introduces an additional latency of approximately 4ms, that contrary to cloud based solutions, allows latency values in the 5ms-30ms range, as required by time-sensitive edge applications.

Index Terms—Fog computing, Edge computing, Security, IoT, Intel SGX.

1 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 service providers [2]. The economies of scale that result from using large shared infrastructures reduce the deployment costs and make it easier to scale the resources associated with each application in response to changes in demand. Cloud computing has been, therefore, widely adopted both by private and public services.

Many applications deployed in the cloud provide a range of services to clients that reside in the *edge* of the network: desktops, laptops, smartphones, and 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 latency when accessing the cloud. In fact, it is known that a response time below 5ms–30ms is typically required for many of these applications [3].

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* [4]. 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* [5]. It assumes the availability of fog nodes 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 [6], [7], 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 middleware 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 [8], graph stores, social networks, online games, among others. The idea of providing an event ordering service is not new (an example is Kronos [9]) 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*, has as main goals to provide the following guarantees over data stored in fog nodes:

- *Integrity:* A fog node cannot modify application data without this being detected.
- *Freshness:* A fog node cannot return an old version of data, without this being detected.
- *Causal Consistency:* A fog node cannot modify the causal order of events without being detected.

- *An earlier version of this paper appeared in the Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'20) [1]. In this journal version, we include additional analysis on use cases, techniques and more evaluation results for Omega.*
- *Cláudio Correia, Miguel Correia and Luís Rodrigues are with INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal. (Email: claudio.correia@tecnico.ulisboa.pt, miguel.p.correia@tecnico.ulisboa.pt, ler@tecnico.ulisboa.pt)*

Manuscript received June 2020; revised ....

Omega leverages the wide availability of support for Trusted Execution Environments (TEE), namely of Intel SGX *enclaves* [10], 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.

Omega is the first system that provides an ordering service that allows clients to access and navigate the history of all events in a secure and efficient manner, despite intrusions in the Omega node. Clients can crawl the event history without having to constantly access the enclave. All events are ordered and stored in the untrusted zone and the client is only required to access the enclave to get the root of the event history.

To illustrate the use of Omega and to assess its performance, we have built a key-value store named *OmegaKV*, that offers causal consistency [11] for the edge. OmegaKV is an extension of causal-consistent key-value stores that have been previously designed for the cloud [8], [12]. 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 [12]. 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 OmegaKV. 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 [3].

2 Background and Related Work

Edge computing [4] 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 [5]. Fog nodes provide computing and storage services to edge nodes with low latency, setting the ground for deploying resource-eager latency-constrained applications, such as augmented reality.

2.1 Securing Fog Services

While some edge infrastructures may be located in secure premises, many applications will require a number of edge servers to be placed in vulnerable locations (e.g., Road Side Units [13]). Having fog nodes dispersed among multiple geographic locations, close to the edge, increases the risk of being attacked and becoming malicious. Therefore, the security of edge services is a growing concern [6], [7], [14]. 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 achieve our goal we leverage secure hardware as a means to harden the implementation. TEE offers 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.

The *Intel Software Guard Extensions* (SGX) are a set of functionalities introduced in the 6th generation Intel Core microprocessors that implement a form of TEEs named *enclaves* [10]. The potential benefits of this technology for the fog have already been recognized by Intel [15], [16] and it has already been used in practice [17]. 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). A limitation of current SGX implementations is that the protected memory region, named enclave page cache, is limited to 128 MB [18]. 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 and LVI [19] exist, Intel continues to investigate how to mitigate these issues. From a security perspective, it is also relevant to maintain a small trusted computing base and to reduce the attack surface.

With time new systems have emerged to alleviate the SGX limitations. SCONE [20] supports secure Linux containers that offer I/O data operations efficiently; in Omega all enclave operations are done in memory thus avoiding the use of I/O operations. HotCalls [21] offers mechanisms to reduce the overhead between enclave and non-enclave communication; Omega could leverage HotCalls to further reduce latency.

ROTE and LCM [22], [23] propose efficient monotonic counters that Omega could use to persistently store its state and prevent rollback attacks. ROTE requires replicas to synchronize when a new monotonic counter is required, which can be a source of delays in edge applications.

2.2 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 [11], 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 [24].

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 [25]. Second, it causes developers to maintain complex code, that is duplicated in many slightly different variations.

Kronos [9] 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 any 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 a 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.

2.3 Edge Storage

To unleash their full potential, fog nodes should not only provide processing capacity, but also cache data that may be frequently used [26]; otherwise, the advantages of processing on the edge may be impaired by frequent remote data accesses [27]. Consequently, a key ingredient 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* [8], [12]. 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.

Systems such as CloudPath [27], Pathstore [28], FogStore [29], and EdgeCons [30] were designed exactly to offer data consistency and storage at the edge. However, none of these systems addresses the security vulnerabilities that fog nodes face. A compromised fog node can create, delete, and/or manipulate the data maintained by these storage systems, leaving the storage and the applications that depend on it in an unpredictable state, as described in Section 3. OmegaKV is protected from these attacks by leveraging Omega and the security properties that Omega provides.

Recently two key-value stores that leverage SGX have been proposed: ShieldStore [17] and Speicher [31]. Both have been designed to operate in data centers at the cloud layer. Omega is a more general ordering service, which can be used to implement a key-value store but also other services at the fog layer. Pesos [32] is secure object store in the cloud that takes advantage of SGX. Pesos assumes a secure third party to persistently store the data, while OmegaKV stores the data locally in the untrusted part. All these systems require that operations call an enclave, incurring with a non-negligible latency overheard. This overheard may result from: i) the enclave context switch; ii) calling the *malloc* function that may involve encrypting and decrypting data in memory; iii) cryptographic operations to prove that the computation was performed inside the enclave; iv) and the use of mechanisms that overcome the limited memory that enclaves suffer. Omega solves this challenge by using the enclave as a root of trust for just a few important operations, and to generate secure data structures. After contacting the enclave once, clients can perform multiple read operations without calling the enclave, with the same integrity and authenticity assurances.

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.

3 Violations of the Event Ordering

Prior to describing the design and implementation of Omega, it is worth enumerating the problems that might occur if the event ordering service is compromised. In this discussion, we assume that the event ordering service is executed in a fog node and that the clients of the service are edge nodes, servers in cloud data centers, or other fog nodes. In this work, we assume that clients are non-faulty and we only address the implications of a faulty implementation of the event ordering service.

The 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 by 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 stale history, by omitting all events subsequent to a given event in the past (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.

4 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 two last features are relevant as they allow a client to check if the information provided by a fog node is fresh and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3078520, IEEE Transactions on Dependable and Secure Computing

TABLE 1
The Omega API.

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

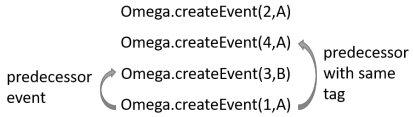

Fig. 1. predecessorEvent and predecessorWithTag functions.

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 [33] of all timestamp requests it receives, effectively defining a total order for all events that occur at the fog node. Any linearization of the event history is consistent with causality.

4.1 Omega API

The interface of the Omega service is depicted in Table 1. 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.). In Section 4.2, we provide examples that illustrate how tags can be used by different applications. 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 logical timestamp.

Note that although Omega is inspired by services such as Kronos, it offers an interface that makes different trade-offs. First, it allows clients to associate events with specific objects / tags and to fetch all previous events that have updated that specific object; Kronos 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. Unlike Kronos, Omega automatically establishes a linearization of all operations, which simplifies the design of applications that need to totally order concurrent operations.

To execute a CreateEvent, it is mandatory to authenticate the client. Other methods in the API do not change the state, and cannot compromise the integrity even if invoked by an attacker. Note that Omega does not offer confidentiality; it only aims at offering integrity and freshness.

4.2 Example Use Cases

Many applications can leverage an event ordering service such as Omega. Examples are applications based on stateless functions, online shops, assisted car driving, online augmented-reality multiplayer games, stream processing engines, social networks, city-scale smart surveillance, and distributed key-value stores. In the following, we use a set of use cases to illustrate how the API exported by Omega can be used for different purposes.

4.2.1 Applications Based on Stateless Functions

New computing models such as microservices [34] and serverless computing [35] allow applications deployed in the cloud to scale seamlessly. These models are based on stateless functions (or services) that are typically small, low complexity, easy to develop, and fast to launch and terminate. Stateless functions typically rely on external services to store and retrieve persistent state. A service such as Omega can provide the methods that allow functions to create and read persistent events securely and with low latency, encapsulation the complexity associated with ensuring the integrity and freshness of data.

Stateless functions can be used by applications to process large amounts of data close to the edge, reducing the volume of data propagated in the network: the raw data is processed by a stateless function to reduce its size and later migrated to the cloud or served to edge clients (for instance, compressing or image background subtraction). Omega can securely store the metadata relative to these images.

Video surveillance for traffic control is an example of an application that can use stateless functions. A camera can be an edge client that creates events in Omega whenever there are variations in the image. Subsequently, images are processed by a stateless function that performs the image processing in background. To ensure the integrity and order of the images, the camera device leverages Omega by generating an event for each image with the correspondent image

hash, by calling createEvent(imageHash,cameraID); Later the data is migrated to the cloud and, if required, the images integrity can be verified by recovering the background and recalculating the correspondent hashes. Moreover, Omega also ensures the correct order of images, reading the previous events using lastEventWithTag(cameraID) and predecessorEvent. Note that image order may be important to reconstruct events, such as an accident or a crime.

In this use case, a malicious fog node can manipulate the content of an image to harm a client. Specifically, the node might add illegal content to an image and later use it against the client. Omega prevents this attack by ordering the events, so reconstructing the entire image sequence through the image hashes allows proving there was no image manipulation. Additionally, all functions that perform computation on the image can also verify this hash to guarantee that they are accessing the correct image.

4.2.2 Video Conferencing Applications

Video conferencing applications can leverage fog nodes to save network bandwidth and offer low latency to their clients. For example, a video conferencing application in a corporate campus can leverage a fog node as a broker of the video streams, so local streams stay within the intranet instead of going to the cloud. In this use case, the application can leverage Omega to locally and securely store access control data. Applications can access this data avoiding accessing the distant cloud, obtaining low response time, and even tolerating faults when the cloud is unreachable [36].

The fog node multicasts the video streams encrypted from the source to the local clients, while Omega locally stores legitimate user lists. A possible implementation is to assume the existence of a unique entity (system owner) capable of creating events on Omega. Although only one entity is capable of creating events, all these events are public. The system owner can be a TEE running on the fog node in parallel with Omega or a special edge client.

In the first case, the TEE needs to store the stream secret (such as a symmetric key) and leverage Omega to store the access control lists. To remove and add users, the system owner creates events on Omega, createEvent(addUserA/ removeUserA, conference1). To read the control data, the system owner can simple scroll through the events generated with the tag corresponding to the conference (lastEventWithTag(conference1) and then use predecessorEvent). To avoid crawling the entire event history, the EventId could be a hash of the legitimate users. In the second, the events are generated in a similar fashion, but the users must run a shared key protocol to generate the video stream secret (tree-based Diffie-Hellman [37]).

4.2.3 Online Augmented-Reality Games

In augmented-reality games, users are able to interact with virtual objects that are placed at multiple physical locations. Examples of online augmented-reality multiplayer games are Pokémon GO, Ingress, or Temple Treasure Hunt. In such games, players can interact indirectly by taking or placing virtual objects on specific locations. For instance, player A could drop some object that players B and C would try to catch. The interaction with these objects can be modeled by *drop* and *catch* events that can be coordinated by a fog node close to the physical location of the virtual object to ensure faster interactions. In this case, the state of the game can be modelled as a function of a totally ordered log of events executed by clients. Without a service such as Omega, a compromised fog node could present to different clients different serialization of events, causing the state of the client application to diverge. For instance, the fog node could report to player A that she has caught the object before player B did and report to player B exactly the opposite.

The extension of such game to use Omega could follow a structure similar to the one discussed for the messaging application, given that, at a higher level of abstraction, both applications maintain a log of events. There are, however, some interesting aspects of the Omega API that can only be illustrated by the current example. In the game, it would be possible to assign a different tag to each virtual object. This would allow the application to keep track of actions that manipulate a given object. However, in this case, the ability to keep track of causal relations among events of different tags (using the method predecessorEvent) is relevant, given that the ownership of some object may be a pre-condition to manipulate some other object (e.g. a player may be required to hold a key in order to remove an object from a vault). Also, the fact that Omega linearizes all events is helpful for scenarios where players concurrently attempt to do the same actions and a serialization is required (e.g. if players B and C try to concurrently catch the same object, only one should succeed). In this situation, the time of arrival of the event to the createEvent API function determines the winner, and the clients must then crawl the object history until they find the earliest pickup event or the client identifier.

4.2.4 Key-Value Stores

Key-value stores are widely used in cloud computing today, and a large number of designs have been implemented [38]. Most of these systems support geo-replication, where copies of the key-value store are kept in multiple data centers. To answer the low latency requirement from edge applications, key-value stores will require to extend their services to the edge and use fog nodes as replicas. Many geo-replicated key-value stores, such as COPS [8] or Saturn [12], support causal consistency. As the name implies, causal consistency requires the ability to keep track of causal relations among multiple put and get operations. This can be achieved with the help of a service such as Omega. We have decided to implement an extension for an existing key-value store to illustrate the benefits of Omega. Therefore, we postpone further discussion on how to use Omega for the implementation of key-value stores to Section 6, where we present OmegaKV.

5 Omega Design and 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 they face. Then, we describe in detail the most important aspects of the implementation.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3078520, IEEE Transactions on Dependable and Secure Computing

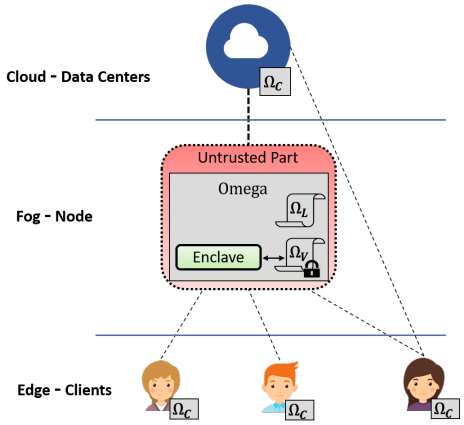

Fig. 2. Omega architecture. Ω_C is Omega client, Ω_V is Omega Vault and Ω_L is the event log.

5.1 System Architecture and Interactions

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 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 paper. 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.

As previously mentioned, we take advantage of Intel SGX. The use of an enclave could lead to memory constraints in our implementation. However, as explained in Section 5.4, Omega is not constrained by the memory available to the enclave. In Omega, the enclave generates a secure data structure that is stored in the untrusted zone; clients can read this data without calling the enclave and still obtain security guarantees.

5.2 Components of the Omega Implementation

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 component named Omega Vault and (Section 5.4); ii) Event log that are used to preserve the Omega state (Section 5.4); iii) an implementation of each method in the API (Section 5.5).

5.3 Threat Model and Security Assumptions

We assume the fog computing model where the network architecture is split into three parts: cloud, fog, and edge. This work addresses the security challenges in the fog layer, protecting fog nodes. Although edge clients and the cloud can also suffer attacks, the methods to address those attacks (e.g., endpoint and cloud security controls) are outside the scope of this paper. The cloud and its services are considered trustworthy, i.e., are assumed to fail only by crashing (essentially, we make the same assumptions as the related work [8], [9], [12]). 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 [39]. We assume the existence of a collision-resistant hash function. In practice we use SHA-256, also recommended by NIST [39]. We use the implementations provided by the SGX SDK (inside the enclave) and Java (outside).

Finally SGX has some limitations and vulnerabilities that are not addressed by our work, and that can be tackled by mechanisms that are orthogonal to the techniques described in this paper. Namely, SGX can be subject to denial of service and side-channels attacks [40], and looses all state upon reboot. To address the latter, Omega could leverage solutions such as ROTE [23] and LCM [22].

5.4 The Omega Vault and the Event Log

Most systems based on SGX require that every operation calls the enclave [17], [31], incurring in a non-negligible

latency overheard and stressing the enclave. Omega strives to overcome this challenge by designing techniques that use the enclave as a root of trust for just a few important operations and leaving the rest of the operations to be carried out without the enclave intervention. This can be beneficial for operations such as predecessorEvent which requires Omega to store all events generated in the past so that clients can crawl the event history. Additionally, Omega is required to securely store different pieces of information, such as the Omega private key, 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). 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 *Omega Vault* needs to maintain the last event generated for each tag (lastEventWithTag) and to ensure that the untrusted components cannot replace the last event by an older event. To ensure that the untrusted zone cannot tamper with the data outside the enclave, we use a Merkle tree [1] over this data and store only the top hash in the enclave. The Merkle tree preserves the data integrity efficiently: it is scalable since the enclave stores a single hash regardless the size of the Merkle tree; and it offers low latency as the number of hashes that need to be computed grows logarithmic with the size of the system (see Sec. 7.2.1). For example, if Omega stores 131072 different tags, the vault only needs to compute 17 different hashes when executing the lastEventWithTag operation.

Our vault implementation supports multi-threaded operation. The data address space is sharded, and each shard is maintained in an independent Merkle tree. This allows the concurrent execution of multiple threads inside the enclave, as long as they are updating different shards. This substantially improves the throughput sustained by the Omega service. Note that even when multiple threads are used, Omega still ensures the serialization of all events: the existence of a sequential history makes the task of crawling the event log easier. This means that the assignment of the last event identifier is still executed in mutual exclusion inside the enclave. However, the fraction of the Omega code that needs to be executed serially is so small, when compared with the remaining code this does not impair the performance. In fact, with the number of cores we have tested (up to 16), we could not observe any significant degradation resulting from the need to serialize events.

The *Event Log* is inspired by blockchain technology [41] and has two main objectives: 1) to store all events generated in Omega so that clients can securely crawl the event history (with predecessorWithTag and predecessorEvent); 2) do so while avoiding the use of the enclave, by maintaining data structures that allow clients to read the data from the untrusted zone and still achieve security guarantees regarding the relative order of the events observed.

To achieve the first objective, the event log records of all events generated; we opted to implement it as a key-value store where events are stored using their unique identifier (assigned by the application) as key. Every time Omega makes a look-up for a specific event (e.g., when a client crawls the event history) it simply finds the event and returns it to the client; the Omega client then verifies the integrity of the event before the value is returned to the application. 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. Since all events possess a digital signature produced by the enclave when they were created, clients can verify the event integrity and have a guarantee that they are reading a correct event. Omega is also an event ordering service and therefore also needs to guarantee the correct order of these events.

The second objective is achieved with a mechanism similar to to the one used to preserve the sequence of blocks in blockchains: each event keeps the unique identifier of the predecessorEvent and the predecedsorWithTag, as shown in Figure 1. This creates a link between consecutive events. In a blockchain data structure like Bitcoin's and Ethereum's, blocks are cryptography linked through a hash; in our case, the event ID is sufficient to securely link the events. These links are secure and cannot be tampered with because every event ID is unique (nonces) and each event has a signature. This data structure allows clients to read data/events with the guarantee that the order of events is correct.

In most cases, clients can request the most recent event from a given tag (lastEventWithTag) and then crawl the causal past of that particular event (predecessorWithTag and predecessorEvent). Note that, in this case, only the first operation requires a call to the enclave and all the other operations can be executed just by reading the event log. This allows the client to crawl the event log from the untrusted zone while still ensuring integrity, authenticity, and the order of the events.

It is worth highlighting the importance that some simple optimizations, as the ones provided by the predecessorWithTag operation, may have for a client at the edge. A system like Omega is capable of processing hundreds of events per second. In the case of an edge client that is only interested in events generated with a certain tag, it can use the operation predecessorWithTag to quickly obtain all the events of that tag. Instead, if the client had access to only the predecessorEvent operation, it would have to crawl through all events that were generated for all tags, several hundreds or thousands. The client would incur in a high latency penalty, especially because it would have to verify digital signatures of all these events despite not being interested in them. With the operation predecessorWithTag, clients can more easily and efficiently get access to the events they are looking for.

5.5 Implementation of the Omega API

Clients invoke the Omega API via a client library. In this way, clients do not need to be aware of the specifics for

communication with the Omega server. In fact, as we discuss here, different methods use different communication primitives to interact with the enclave. Also, some of the methods can be executed directly by the client library and do not require any message exchange with the enclave. In the next paragraphs, we describe the implementation of each primitive in detail.

The method createEvent is the only method that modifies the state of the Omega server in the fog node. The method createEvent is used to create a new event in the server. The state of an event is a tuple that contains the following fields: i) a unique timestamp, that is associated to the event by the server (in the current implementation, this timestamp is a sequence number); ii) the *EventId*; iii) the associated *EventTag*; iv) the *EventId* of the last event generated by Omega; v) the *EventId* of the last event generated by Omega with the same tag. The identifiers of the predecessor events are maintained in the Omega vault. The new tuple is signed with the private key of the Omega server. Subsequently, the Omega server replaces the identifier of the last event generated by the identifier of the new event and replaces the identifier of the last event generated with the given tag, by the new event. As noted, these variables are maintained in the secured Omega vault. Then, the tuple is also stored in the *event log*, maintained in the non-secured portion of the fog node. Finally, the tuple that represents the event is returned to the client.

The methods lastEvent, lastEventWithTag, predecessorEvent, and predecessorWithTag do not change the state of the Omega. When the server receives a lastEvent request it extracts the last event it has processed from the vault (i.e, a tuple with the fields enumerated in the previous paragraph) to the client. Similarly, when the server receives a lastEventWithTag request, it uses the vault to extract the previous request and sends it to the client. The requests predecessorEvent and predecessorWithTag are executed collaboratively by the client library and the server. The client library, that is aware of the internal structure of the *Event* tuple, extracts the timestamp of the event. This event identifier is sent to the server that fetches the complete event tuple associated to that identifier from the event log. Finally, the full tuple associated with the desired event is returned to the client.

Lastly, the methods orderEvents, getId, and getTag require no communication with the enclave, and are implemented directly on the library. The first method extracts the timestamp field from each tuple, compares their values, and returns the tuple with lower timestamp. The other two simply return the corresponding fields from the input tuple.

Note that several of the methods described above require the Omega server to extract information from the vault and/or from the event log. The integrity of the information maintained in the vault is ensured by construction. Also the server can always check the validity of records extracted from the event log (since each tuple is signed with the private key of the server, which is securely stored in the enclave). However, the Omega server cannot prevent the non-secured portion of the fog node from deleting information from stable storage, making the vault, the log, or both unavailable. In this case, the part of Omega that runs inside the enclave detects the corruption, stops operating, and reports an error.

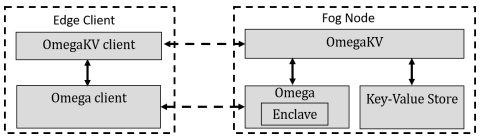

Fig. 3. OmegaKV service components.

6 Omega Key-Value Store

OmegaKV is an extension to key-value 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. 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, as illustrated in Figure 3. 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. 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 $hash(k \oplus v)$.

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 server 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 securely 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.

By leveraging Omega, OmegaKV is capable of offering another operation, getKeyDependencies. This operation takes two input arguments, the target key, and a limit. This operation will read all predecessors of the key up to the limit number, and return key-value pairs. When the limit is

TABLE 2
SGX-based systems comparison. RYW is read your writes.

	integrity and freshness	scalability	consistency	secure history
Speicher	O(n)	no	RYW	yes
Enclavecache	no	–	RYW	no
Securekeeper	no	–	linearializability	no
Concerto	(upon request)	yes	RYW	yes
ShieldStore	O(n)	yes	RYW	no
OmegaKV + Omega	O(log_n)	yes	causal	yes

zero, OmegaKV crawls to the end of Omega history. With this operation, clients are able to obtain the dependencies of a given key.

As OmegaKV resorts to Omega, OmegaKV inherits many of the Omega security properties. Thus, OmegaKV compares positively with other key-value stores based on SGX, as shown in Table 6. OmegaKV maintains data freshness through Omega Vault that has a logarithm cost, while other systems have a linear cost [17], [31]. Concerto [42] verifies data integrity in a deferred manner and at client request, a solution that is not practical for the edge. Interestingly Concerto also implements a Merkle tree outside the enclave, but needs to pass as input the entire path from the leaf to the root through the Ecall. Omega Vault leverages the *user_check* parameter allowing the enclave to directly access the Merkle tree nodes in untrusted memory. Another important factor is the scalability that these techniques hold. While a single top hash is sufficient to ensure integrity in Omega Vault, Speicher stores a table inside the enclave that needs to be flushed to disk. Most key-value stores, such as EnclaveCache [43], offer Read your Write data consistency. Securekeeper [44] performs consensus among replicas that can result in long latencies. Omega, similarly to Saturn, orders all events with timestamps to capture causal consistency. Also, OmegaKV offers a complete history of operations by resorting to the event log.

7 Evaluation

This section is divided in two parts. First, 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. Second, we show the impact of using Omega to secure a concrete service, 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 Omega.

7.1 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.

Low latency communication is one of the main motivations for fog computing. Computation close to the network edge can achieve low latency: in 5G cellular networks that can include computing capacity in the access nodes; in Content Delivery Network (CDN) as recent work has pushed computation to CDN nodes; in Vehicle-to-Infrastructure (V2I) communications when cars interact directly with Road Side Units (RSUs). In all these cases, low communication latency is a consequence of direct (1-hop) communication. In our experiments, clients and fog node were deployed in our laboratory, in the same network, emulating a 5G station communicating with a terminal (1-hop). The latency has been tuned to be aligned with the expected latency of 5G networks and future mobile edge computing (MEC) networks (below 1ms, according to Imtiaz et al. [45]). Cloud services were executed on an Amazon Elastic Compute Cloud (Amazon EC2) datacenter in London, selected as the closest in Round Trip Time (RTT) to our lab located in Lisbon, in t2.micro virtual machines. This setting captures many realistic scenarios where clients are diverted to the closest datacenter. The observed experimental latency is consistent with values obtained by others [46].

The Intel SGX SDK and the code for the enclave are in C/C++. Omega was implemented in Java 11 and the Java Native Interface (JNI) was used as a bridge between Java and C++. For persistent storage we use the Redis key-value store [38] and Jedis, the Java Redis client library, to interact with Redis.

7.2 Omega Server Side Performance

We first provide an overview for the performance of Omega and the main functions executed in Omega. We will discuss the Omega server-side performance, i.e. discarding the client's cryptographic overhead. Then we discuss the performance of the event log, a component capable of offering minimal latency for edge clients.

7.2.1 Omega Operations

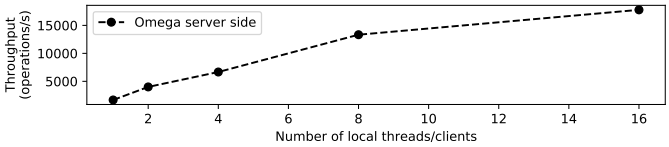

Fig. 4. Server side scalability of Omega's createEvent (1 to 16 threads).

We now present the results from two experiments that aim at assessing the performance of Omega, in particular of the operations that are mainly executed in the enclave. We have measured the performance of the createEvent operation, as this is the most expensive of all operations provided by Omega and involves updating the Omega vault.

In the first experiment, we show that the performance of Omega can scale as more threads are allocated to the service. Figure 4 depicts the maximum number of operations per second that our implementation can execute as the number of threads increase. It can be seen that the throughput of the system increases almost linearly up the 8 threads (the number of real cores in the machine that we have used). This is possible because cryptographic operations are performed in parallel within the enclave and the Omega vault is sharded. Updates to different shards can also be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3078520, IEEE Transactions on Dependable and Secure Computing

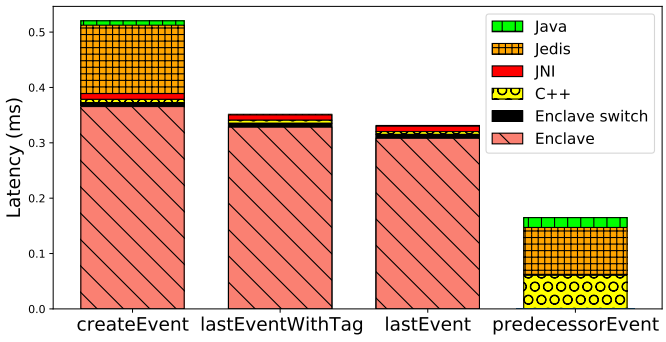

Fig. 5. Server side operation latency for createEvent, lastEventWithTag, predecessorEvent, and lastEvent.

executed concurrently, without blocking each other. Note also that the derivative of the line is below 1; this is due to the overhead induced by the synchronization required to enforce the serialization guarantees offered by Omega and due to leveraging hyperthreading.

Figure 5 shows how each software component executed in an operation critical path contributes to the final latency. We show the latency for the createEvent, lastEventWithTag, predecessorEvent, and lastEvent operations of the Omega API. These operations use different components, that have different properties, and therefore exhibit different performance. All the results were obtained using the most recent implementation of Omega, that includes a number of optimizations with regard to previous versions, avoiding some redundant memory copies and comparisons. Therefore, these results are better than those of the original Omega code [1].

The createEvent operation takes around 0.5ms to be executed by the Omega server and is the slowest provided by Omega. It involves calling the enclave, verifying and computing digital signatures, calling the Omega Vault (acquiring the lock over the partition to be modified) and the event log, and, finally, replacing the value of *lastEvent* within the enclave in an atomic manner. The event log uses Redis (Section 5.4), so to store the event in the event log Omega needs to transform the event into a string. The latency introduced by this transformation (in green in Figure 5) together with the work required by Jedis to store the event in the Redis, leads to a penalty close to 0.1ms. Comparing with Figure 4, the 8 clients experiment obtains an average throughput of $13333 op/s$ corresponding to a latency just over $0.5ms$ validating these results.

The lastEventWithTag operation requires the enclave to verify the client's signature, then to use the Omega Vault to get the most recent event for a given tag. This operation also requires an access to the partition lock and the verification of the Merkle tree. Still, it is faster than createEvent. The enclave calculates a new digital signature with a nonce that comes from the client to ensure freshness. To execute this operation, Omega does not need to use Redis, since this event was stored at the time of its creation. This results in a noticeable reduction in latency compared to createEvent.

The lastEvent operation requires the enclave to verify first the client's signature, then to atomically read the most recent event stored inside the enclave, to compute a new digital signature (also with a client's nonce) and, finally, to return the event to the client. The visible latency difference between lastEvent and lastEventWithTag is due to the use of the Omega Vault and its Merkle tree. We conclude that the time spent inside the enclave is mostly associated with the operations required to verify and compute digital signatures. The use of the Merkle tree is very efficient, causing a small overhead. In the experiments, the system was storing 16384 different tags, using a Merkle tree with 14 levels.

The predecessorEvent operation is similar to predecessorWithTag. The most interesting aspect of this operation is that it does not require the use of the enclave, as it does not require freshness. However, the untrusted part still verifies the client's signature, which is the time spent on the C++ component in the graph (as expected, C ++ is much more efficient in cryptographic operations than Java). The figure shows that a substantial fraction of time is spent in the interface with Redis, namely transforming the stored string into a Java object to be returned to the client. The impact of the JNI is not noticeable in this case, as the C++ layer only returns a boolean instead of an entire event as in all other operations.

The observed latencies match the requirements of edge applications. For instance, in vehicular applications, the overall connection time of a vehicle with an RSU is typically around 18-21s for a vehicle moving at 120 km/h [47], which allows a vehicle to access multiple types of events such as congestion control, driving conditions, curve speed, and others. The latency of $0.6ms$ also matches the maximum tolerable delay for many other edge applications, such as the $\sim 7ms$ required for virtual reality gaming [48], the $\sim 10ms$ needed for augmented reality apps [49], and the $\sim 100ms$ needed for image processing [50].

7.2.2 Performance of the Event Log

We now evaluate the performance of the event log component. Figure 6 depicts the server side latency observed when an client performs read operations, as a function of the number of clients performing concurrent requests on Omega. The vertical bars plotted at each point represent the confidence interval at 99%. Each point is the average of 10,000 reads, but the variance was low, so the confidence intervals are narrow (the lines are almost superimposed).

The top line of the experiment was obtained using a single-threaded version of Omega and a single Merkle tree (1 MT). This version presents the worst performance. The middle line shows results for the multi-threaded version using 512 partitions/Merkle trees. This version performs better but it is possible to observe a latency degradation when the processor can no longer execute the cryptographic operations concurrently in an efficient manner (this happens with 32 clients or more). Note that, in these experiments, the client is executing the lastEventWithTag operation, that uses both the enclave and the Omega Vault; threads executing this operation need to perform synchronized access to shared variables.

The line in the bottom of figure depicts the server side latency when a client performs the predecessorEvent operation in the multi-threaded Omega. In this case, the client almost does not notice any increase in latency, despite the concurrent execution of other clients. This happens because the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2021.3078520, IEEE Transactions on Dependable and Secure Computing

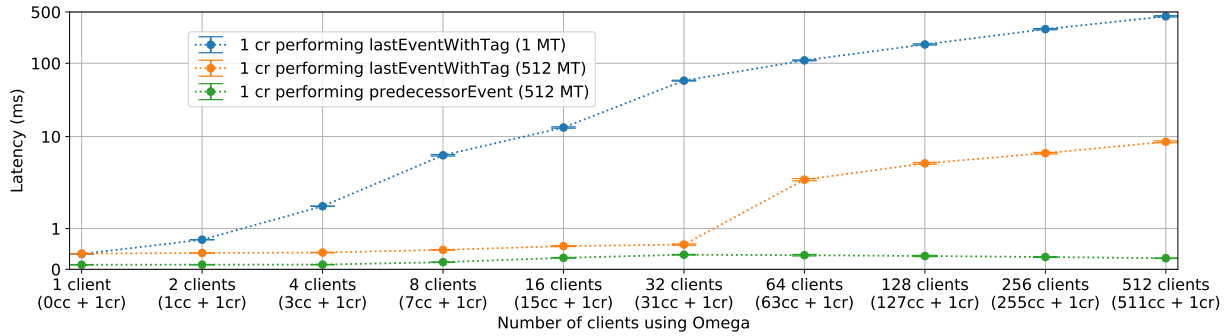

Fig. 6. Server side operation latency while the enclave is concurrently accessed. cc is client creating events, cr is client reading events, and MT is Merkle tree.

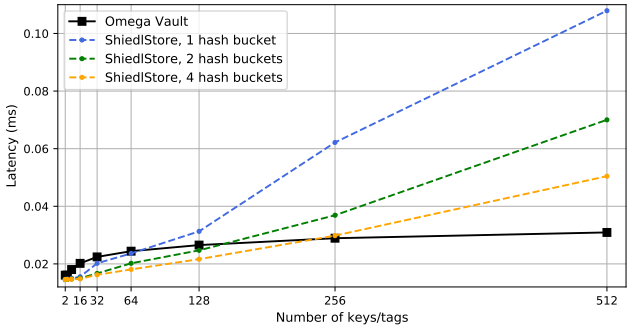

Fig. 7. Performance of Omega Vault vs the ShieldStore hash bucket data structure.

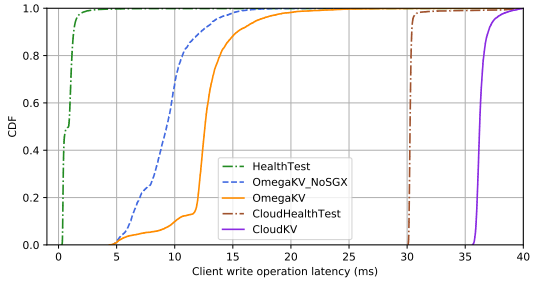

Fig. 8. Write operation latency of a fog node and cloud.

client thread does not need to call the enclave and can avoid the use of synchronization primitives (such as the locks on each partition/Merkle tree); therefore the implementation can quickly read the event log event and return to the client. These results clearly demonstrate the benefits that can be achieved when the operations can be executed without incurring the overheads associated with the enclave calls. One can also observe the same latency difference between the predecessorEvent operation and the lastEventWithTag visible in Figura 5, where predecessorEvent has a latency close to $0.4ms$ and the lastEventWithTag just over $0.1ms$. Futher, comparing with Figure 4 it is possible to observe the slight increase in latency beyond 8 clients.

7.2.3 Performance of the Omega Vault

In this section we evaluate the performance of the Omega Vault component, not to be confused with OmegaKV. To evaluate the performance of Omega Vault we chose to compare it with an open-source system, ShieldStore [17]. ShieldStore also uses cryptographic techniques to store data outside the enclave in a secure manner, with integrity, freshness, and confidentiality. However, ShieldStore uses a flat Merkle tree to ensure data integrity; a Flat Merkle tree fails to offer the logarithmic cost that Omega Vault offers. Furthermore, the solution proposed by ShieldStore uses a linked list on the leaves of the flat Merkle tree, named hash buckets. Linked lists impose a linear cost when the system grows. This linear cost is visible in Figure 7; when the number of keys increases, ShieldStore has a linear growth in latency. However, Omega demonstrates a logarithmic growth in latency, since it leverages a pure Merkle tree. This experiment shows clearly that is preferable to implement a pure Merkle tree over linked lists.

7.3 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 [38] to keep their state persistent. Also, all systems 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 the integrity of stored data, and they do not need to use JNI interface.

Figure 8 compares the latency that a client experiences when using the services OmegaKV, OmegaKV _NoSGX, and CloudKV. For a better understanding of the graph, we measure the ping operation to calculate the round-trip time from the client to the fog node and to the cloud; this is shown as HealthTest line for the fog node and

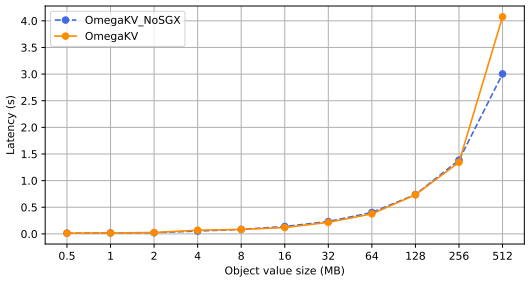

Fig. 9. Write operation latencies w/ and w/o SGX.

CloudHealthTest for the cloud[1]). 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. OmegaKV can offer latency values in the 5ms–30ms range required by time-sensitive edge applications [3].

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 9. 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.

8 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 different locations. These cannot be physically secured with the same level of trust than cloud premises. This paper moves towards better resiliency by describing the design and implementation of a secure 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. Our evaluation shows that, despite the costs incurred with the use of the enclave, less than 0.4ms, the use of Omega based applications can still provide much smaller latency and higher throughput than current cloud based solutions.

1. The latency was tuned to be aligned with the expected latency of 5G networks and future MEC networks (below 1ms [45]). Note that 5G towers have a short-range compared to 4G even with no obstructions [51]. Clients can lose the signal if they physically move away from the towers.

Acknowledgments

This work was partially supported by the Fundação para a Ciência e Tecnologia (FCT) via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the "Programa Operacional Regional de Lisboa na sua componente FEDER" with ref. Lisboa-01-0145-FEDER-029271), Project NG-STORAGE (PTDC/CCI-INF/32038/2017), project UIDB/ 50021/ 2020, and by the European Commission under grant agreement number 830892 (SPARTA).

References

[1] C. Correia, M. Correia, and L. Rodrigues, "Omega: a secure event ordering service for the edge," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, València, Spain, Jun. 2020.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, 2010.

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

[4] Y. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing—a key technology towards 5G," *ETSI white paper*, vol. 11, no. 11, 2015.

[5] L. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, 2014.

[6] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu, "Data security and privacy-preserving in edge computing paradigm: Survey and open issues," *IEEE Access*, vol. 6, 2018.

[7] 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.

[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 *ACM Symposium on Operating Systems Principles*, Cascais, Portugal, Oct. 2011.

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

[10] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *International Workshop on Hardware and Architectural Support for Security and Privacy*, Tel-Aviv, Israel, Jun. 2013.

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

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

[13] J. Ni, A. Zhang, X. Lin, and X. Shen, "Security, privacy, and fairness in fog-based vehicular crowdsensing," *IEEE Communications Magazine*, vol. 55, no. 6, 2017.

[14] W. Zhou, Y. Jia, A. Peng, Y. Zhang, and P. Liu, "The effect of IoT new features on security and privacy: New threats, existing solutions, and challenges yet to be solved," *IEEE Internet of Things Journal*, vol. 6, no. 2, 2018.

[15] Intel Corporation, "Intel's fog reference design overview," https://www.intel.com/content/www/us/en/internet-of-things/fog-reference-design-overview.html, accessed: 2019-10-04.

[16] Z. Ning, J. Liao, F. Zhang, and W. Shi, "Preliminary study of trusted execution environments on heterogeneous edge platforms," in *ACM/IEEE Workshop on Security and Privacy in Edge Computing*, Bellevue, WA, USA, Oct. 2018.

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

[18] Intel Corporation, "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: 2019-10-04.

[19] J. van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *Symposium on Security and Privacy*, May 2020.

[20] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell *et al.*, "SCONE: Secure linux containers with intel SGX," in *USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, USA, Nov. 2016.

[21] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, 2017.

[22] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, Denver, CO, USA, Jun. 2017.

[23] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," in *USENIX Security Symposium Security*, Vancouver, Canada, Aug. 2017.

[24] B. Sanders, "The information structure of distributed mutual exclusion algorithms," *ACM Transactions on Computer Systems*, vol. 5, no. 3, 1987.

[25] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, 1998.

[26] E. Ahmed and M. Rehmani, "Mobile edge computing: Opportunities, solutions, and challenges," *Pervasive Computing*, vol. 70, 2017.

[27] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *ACM/IEEE Symposium on Edge Computing*, San Jose, CA, USA, Oct. 2017.

[28] S. Mortazavi, B. Balasubramanian, E. de Lara, and S. Narayanan, "Pathstore, a data storage layer for the edge," in *International Conference on Mobile Systems, Applications, and Services*, Munich, Germany, Jun. 2018.

[29] R. Mayer, H. Gupta, E. Saurez, and U. Ramachandran, "Fogstore: Toward a distributed data store for fog computing," in *IEEE Fog World Congress*, Santa Clara, CA, USA, Oct. 2017.

[30] Z. Hao, S. Yi, and Q. Li, "Edgecons: Achieving efficient consensus in edge computing networks," in *USENIX Workshop on Hot Topics in Edge Computing*, Boston, MA, USA, Jul. 2018.

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

[32] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: policy enhanced secure object store," in *ACM European Conference on Computer Systems*, Porto, Portugal, Apr. 2018.

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

[34] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, 2015.

[35] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017.

[36] B. News. (2020) AWS: amazon web outage breaks vacuums and doorbells. [Online]. Available: https://www.bbc.com/news/technology-55087054

[37] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, "On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees," in *ACM Conference on Computer and Communications Security*, Toronto, Canada, Oct. 2018.

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

[39] E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths," NIST, Special Publication 800-131 A r2, Mar. 2019.

[40] J. van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, Baltimore, MD, USA, Aug. 2018.

[41] M. Peck, "Blockchains: How they work and why they'll change the world," *IEEE Spectrum*, vol. 54, no. 10, 2017.

[42] A. Arasu, K. Eguro, R. Kaushik, D. Kossmann, P. Meng, V. Pandey, and R. Ramamurthy, "Concerto: A high concurrency key-value store with integrity," in *ACM International Conference on Management of Data*, Chicago, IL USA, May 2017.

[43] L. Chen, J. Li, R. Ma, H. Guan, and H.-A. Jacobsen, "EnclaveCache: A secure and scalable key-value cache in multi-tenant clouds using Intel SGX," in *Middleware Conference*, Davis, CA, USA, Dec. 2019.

[44] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "Securekeeper: confidential zookeeper using intel sgx," in *Middleware Conference*, Trento, Italy, Dec. 2016.

[45] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai, "A survey on low latency towards 5g: Ran, core network and caching solutions," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, 2018.

[46] "AWS inter-region latency," https://www.cloudping.co/, accessed: 2020-03-02.

[47] N. Lu, N. Cheng, N. Zhang, X. Shen, and J. W. Mark, "Connected vehicles: Solutions and challenges," *IEEE internet of things journal*, vol. 1, no. 4, 2014.

[48] S. Mangiante, G. Klas, A. Navon, Z. GuanHua, J. Ran, and M. Silva, "VR is on the Edge: How to deliver 360 videos in mobile networks," in *Workshop on Virtual Reality and Augmented Reality Network*, Los Angeles, CA, USA, Aug. 2017.

[49] R.-S. Schmoll, S. Pandi, P. Braun, and F. Fitzek, "Demonstration of vr/ar offloading to mobile edge cloud for low latency 5g gaming application," in *IEEE Consumer Communications & Networking Conference*, Las Vegas, NV, USA, Jan. 2018.

[50] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky *et al.*, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *ACM/IEEE Symposium on Edge Computing*, San Jose, CA, USA, Oct. 2017.

[51] VIAVI solutions. (2020) What is 5G technology? [Online]. Available: https://www.viavisolutions.com/en-us/5g-technology

Cláudio Correia owns a MSc in Computer Science and Engineering by the Instituto Superior Técnico (IST), Universidade de Lisboa and he is now a PhD candidate at IST and a researcher at INESC-ID Lisboa. His research interests are in the area of computer security and distributed and edge computing.

Miguel Correia (Senior Member, IEEE) is an Associate Professor with Habilitation at Instituto Superior Técnico, Universidade de Lisboa and a researcher at INESC-ID. His research is focused on (cyber)security and dependability (aka fault tolerance), typically in distributed systems, in the context of different applications (blockchain, cloud, mobile). He has more than 200 publications in these areas.

Luís Rodrigues (Senior Member, IEEE) is a professor at Instituto Superior Tecnico, Universidade de Lisboa and a researcher at INESC-ID. His research interests lie in the area of reliable distributed systems. He is co-author of more than 200 papers and 3 textbooks on these topics. He is also a senior member of the ACM.