Omega: a Secure Event Ordering Service for the Edge

Cláudio Correia Miguel Correia Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{claudio.correia, miguel.p.correia, ler}@tecnico.ulisboa.pt

***Abstract*—Edge computing is a paradigm that extends cloud computing with storage and processing capacity close to the edge of the network that 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 secure the functions they provide. 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. In this paper 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) based on Intel 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.**

***Keywords*-Security, IoT, Fog, Edge, Intel SGX**

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 [1].

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 [2]. Thus, clients that operate far from these data centers may experience long latencies [3].

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 [4].

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* [5]. 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* [6]–[8]. It assumes the availability 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 [9], [10], 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 [11], graph stores [12], [13], social networks [14], online games [15], among others. The idea of providing an event ordering service is not new (an example is Kronos [16]) 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.

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.

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 [17] for the edge. OmegaKV is an extension of causal-consistent key-value stores that have been previously designed for the cloud [11], [18]–[20]. 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 [21]–[23]. 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.

II. Background and Related Work

A. Edge Computing and Fog Nodes

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 for deploying resource-eager latency-constrained applications, such as augmented reality.

B. 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 [24]). 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 [9], [10], [25]. 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 [26]) and, in some cases, to detect compromised nodes. Techniques such as Byzantine quorums [3], [27] 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 if the fog node can execute quorum validations in the background and is hardened.

Hardening [28] 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. TEEs have been identified as one of the most promising technologies to secure computation and sensitive data in fog nodes [29].

***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* [30], [31]. The potential benefits of this technology for the fog have already been recognized by Intel [32] and it has already been used in practice [33], [34]. 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 [35]. A limitation of current SGX implementations is that the protected memory region, named enclave page cache, is limited to 128 MB [36]. 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 [37], [38] exist, Intel continues to investigate how

to mitigate these issues.

With the availability of Intel SGX new systems have emerged to alleviate SGX limitations. SCONE [39] 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. ROTE and LCM [40], [41] propose efficient monotonic counter that Omega could use to persistently store its state and prevent rollback attacks.

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 [42], [43], logical Lamport clocks [17], vector clocks [44], [45], hybrid clocks [46], 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 [47], [48].

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 [49], [50]. Second, it causes developers to maintain potentially complex code, that is duplicated in many slightly different variations.

Kronos [16] 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 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.

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 [51]; otherwise, the advantages of processing on the edge may be impaired by frequent remote data accesses [52]. By using cached data, requests rarely need to be served by data centers. 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* [11], [18]. 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 [53].

Very recently, two key-value stores that leverage SGX have been proposed: ShieldStore [54] and Speicher [55]. Both have been designed to operate in data centers at the cloud layer. Omega is a more general service, that can be used to implement a key-value store but also other services at the fog layer. The authors of ShieldStore suggest that a Merkle tree could be used to store data outside the enclave, but they have not implemented that strategy. As it will be discussed, Omega, that was developed concurrently with ShieldStore and Speicher, does use and evaluate the use of a Merkle tree in its implementation. Speicher uses a table in memory and stores within the enclave one hash per row of this table being a limitation on system scalability. Additionally, when this table becomes full, Speicher uses the enclave to store data on disk having a heavy latency cost. Pesos [56] is secure object store that 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 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 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.

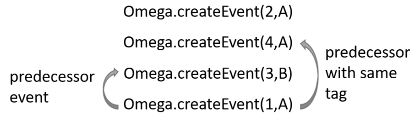

Fig. 1. predecessorEvent and predecessorWithTag functions.

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 [57] 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.

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). In Section IV-B, 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), as shown in Figure 1. 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). Finally getId and getTag extract the application level event identifier and tag that have been securely bound with the target logical timestamp.

TABLE I
The Omega API.

Register a tag with Omega void registerTag (EventTag tag)
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)

Note that, although Omega is inspired by services such as Kronos, 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 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 totally order concurrent operations.

B. Example Use Cases

Many applications, such as online augmented-reality multi-player games, assisted car driving, and distributed key-value stores, can leverage an event ordering service such as Omega. In the following, we use two of these examples to illustrate how the API exported by Omega can be used for different purposes.

*1) **Fog-Assisted Car Driving**:* Edge computing has the potential to play a key role in vehicular networks, an area whose significance is growing given the increasing number of sensors deployed in current cars and the increasing autonomous functions that cars can execute. An important component of vehicular networks is the vehicle-to-infrastructure (V2I) communication [58], [59], that allows vehicles to share the information they produce and also to consume information that can improve their autonomous behaviour. The infrastructure in the V2I is made up of Road Side Units (RSU)s that are situated at multiple points along roads; RSUs provide resources to local vehicles and inform the cloud of local events. Recent research indicates that fog nodes are promising candidates to operate as upgraded RSUs, with more memory and processing

power [24].

Among the many services that can be provided by RSUs/ fog nodes, one is to support information sharing regarding road events or constraints such as accident warnings, congestion control, driving conditions, curve speed warning, stop signal gap assist, speed limits, road-weather information, and others. These features will allow to deploy intelligent traffic lights and other smart cities applications [60]. In such scenarios, the operation of traffic lights, and the routes used in urban areas can be optimized using the information provided by fog nodes. Unfortunately, these applications also make RSUs appealing targets for cyberterrorism, and there is a growing concern regarding the risks and attacks to vehicular networks [59], [61], [62]. If an attacker could selectively select which events are propagated to a given traffic light or to a given vehicle it could easily distort the perception of the actual road conditions, and would be able to manipulate traffic and generate congestion.

A service such as Omega can play an essential role to secure such infrastructure by providing the means that allow intelligent traffic lights and intelligent cars to check if the RSU is providing a fresh and gapless record of the incidents reported by other vehicles. Vehicles could use the Omega service to report occurrences to fog nodes, and use the Omega ability to securely crawl the event log to detect malicious gaps in the information reported. If a fog node is detected to be compromised, the traffic light could simply fallback to a fixed round-robin schedule of green/red signs to ensure traffic safety.

*2) **Key-Value Stores**:* Key-value stores are widely used in cloud computing today, and a large number of designs have been implemented [63]–[65]. Most of these systems support geo-replication, where copies of the key-value store are kept in multiple data centers. Geo-replication is relevant to ensure data availability in case of network partitions and catastrophic faults, but it is also instrumental to serve clients with lower latency than what would be possible with a non-replicated system. However, as discussed previously, cloud-based geo-replication may not suffice to achieve the small latencies required by novel latency-critical applications. Therefore, extending key-value stores to operate on fog-nodes is a relevant research challenge. Many geo-replicated key-value stores, such as COPS [11], Saturn [18], or Occult [66], 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 VI, where we present OmegaKV.

V. 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.

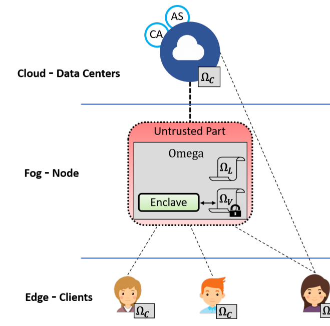

Fig. 2. Omega architecture. CA is certification authority, AS is attestation server, Ω_C is Omega client, Ω_V is Omega Vault and Ω_L is the event log.

A. 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.

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-D). 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 [67], [68]).

As previously mentioned, we take advantage of Intel SGX. The use of an enclave could lead to memory constraints in our implementation. However, as will be explained in Section V-E, Omega is not constrained by the memory available to the enclave. This is a fundamental advantage of Omega and a key distinctive feature with regard to related systems such as ShieldStore [54]. In Omega, only the top hash of a Merkel tree is required to be stored in the enclave, the rest of the tree is stored in RAM in the untrusted zone. Also, the cost of Omega functions only grows logarithmically with the size

of the dataset, as opposed to ShieldStore, Speicher [55], and Pesos [56], whose cost grows linearly. Scalability is a key attribute of Omega.

B. 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 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-D); ii) two sub-components named vault and event log that are used to preserve the Omega state (Section V-E); iii) an implementation of each method in the API (Section V-F).

C. 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 [11], [16], [18]–[20]). 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 [69] with 256-bit keys, which is recommended by NIST [70]. We assume the existence of a collision-resistant hash function. In practice we use SHA-256 [71], also recommended by NIST [70]. 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).

D. Client Binding

Before a client invokes any method of the Omega API must 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 [30]. One limitation of the 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 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. Our solution is inspired in Excalibur [72], a service designed for the Trusted Platform Module [73] that also aims at preventing clients from attesting directly all servers. However, Omega uses substantially different techniques, in particular, Excalibur requires the transmission of keys in the network, which is significantly less robust than Omega's protocol. 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 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.

E. 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,

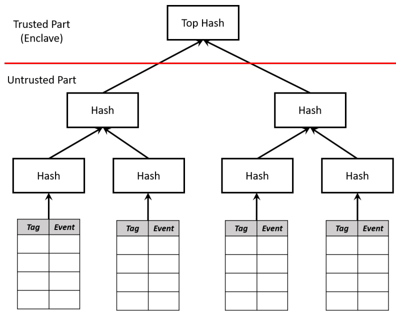

Fig. 3. Merkle tree stored in the Omega vault in the untrusted zone of the fog node (with $N = 4$).

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 its implementation needs 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 have not been changed. At the logical level, this is achieved by requiring the enclave to hash the vault every time it updates its content; the hash is stored at 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* [74]. 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 while storing only the top hash.

When the enclave requires 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.

Our implementation of the Omega vault is optimized to support multi-threaded operation. The tag 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, that it does not represent an impairment to 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.

F. 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 methods registerTag and createEvent are the only methods that modify the state of the Omega server in the fog node. The method registerTag registers tags, so it has to adjust the space allocated to the vault when the space available is exhausted (recall that Omega keeps track of the last event associated with each tag it observed); 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* (that must be a tag previously registered via the *registerTag* event); 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* that is 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 safely 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.

VI. 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. OmegaKV also ensures that all updates performed by edge clients on the fog node, if they are propagated to the cloud, are propagated 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 paper, 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.

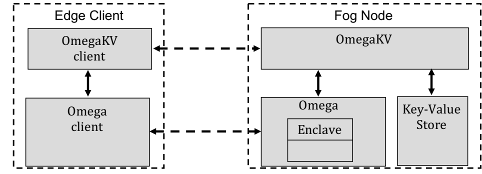

Fig. 4. OmegaKV service components.

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, 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 $hash(k \oplus v)$. The operation

Algorithm 1 OmegaKV Implementation

```
1: function PUT(k, v)
2:     event_id ← hash(k ⊕ v).
3:     e ← omega.createEvent (event_id, k)
4:     BEGIN ATOMIC
5:     (old_v, old_e) ← local_kv.get (k)
6:     if old_e = omega.orderEvents (old_e, e) then
7:         local_kv.put (k, (v, e))
8:     END ATOMIC
9: function GET(k)
10:    (v, e) ← local_kv.get (k)
11:    event_id ← getId (omega.lastEventWithTag (k))
12:    hash_val ← hash(k ⊕ v).
13:    if event_id = hash_val then
14:        return v
15:    else
16:        return error;
```

of the OmegaKV is summarized in Algorithm 1.

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 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

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, 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[1]). Cloud services are executed on a data center in London[2], using Amazon Elastic Compute Cloud (Amazon EC2) in t2.micro virtual machines

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 key-value store Redis [65]. In the experiments we executed 5000 operations and discarded the first and the last 500 to avoid outliers.

[1]This has been tuned to be aligned with the expected latency of 5G networks and future MEC networks [75].

[2]The datacenter was selected as the closest (in Round Trip Time (RTT)) to our lab. Our lab is located in Europe but not in the United Kingdom. This setting captures many realistic scenarios where clients are diverted to the closest datacenter in their region. The observed experimental latency is consistent with latency values collected by others [76].

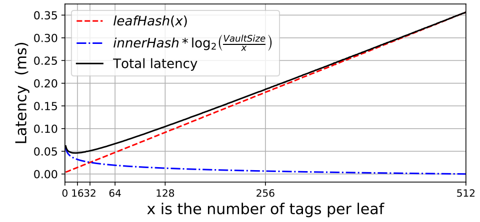

Fig. 5. Estimated optimal leaf size of the Merkle tree (vault of size 512).

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.

*1) **Merkle Tree Configuration**:* The Merkle tree used to store events is used on most of the Omega operations. Therefore, its proper tuning 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 suggests 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. 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 obtained using the formula above, that was fed with results obtained experimentally for the parameters $\mathit{leafHash}(x)$ and *innerHash*. The values suggest that leaves should not be large; in this case, for a vault size of 512, the formula suggests that 8 is the best leaf size.

Based on this observation, we decided to run multiple experiments on the real system, where we measured the

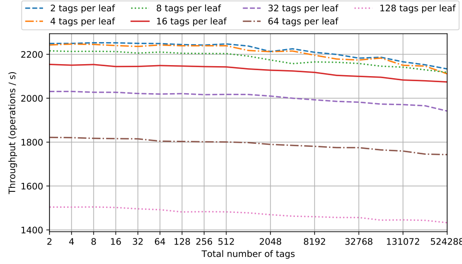

Fig. 6. Performance of the Merkle tree.

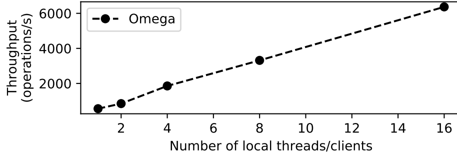

Fig. 7. Scalability of Omega's *createEvent* implementation (1 to 16 threads).

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.

*2) **Executing Omega Operations**:* We now present the results from two experiments that aim at assessing the performance of the Omega implementation, 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 7 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 16 threads (the number of available 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 and

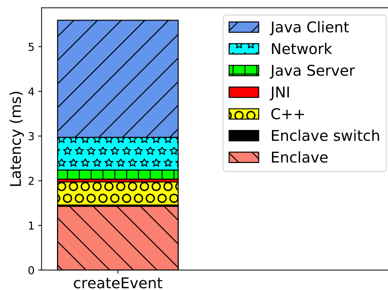

Fig. 8. Operation latency for *createEvent*.

updates to different shards can also be 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.

In the second experiment, we measure the relative impact of the latency introduced by the Omega's implementation in the client operation. Figure 8 shows how each individual software component that is executed in the client critical path contributes to the latency. This breakdown can be used to estimate the performance of Omega in other networks. 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 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 (as describe in Section V-F). Thus, the main contributor to the latency are the cryptographic functions executed in the client and in the enclave. In the client, $2ms$–$2.5ms$ are required to compute and verify digital signatures. On the server side, most of the time is also spent in the processing digital signatures.

The observed latencies match the requirements of edge applications. For instance, in vehicular applications, safety application require warnings to be generated in less than $100ms$ [77], [78]; as depicted in Figure 8, creating an event with Omega has a latency close to $5ms$, which is considerably below the $100ms$ threshold, allowing to create and deliver multiple events to vehicles using Omega on an RSU/fog node and still meet the deadline. Also, the overall connection time of a vehicle with an RSU is tipically around 18-$21s$ for a vehicle moving at 120 km/h [79], [80], which allows a vehicle to access other types of events such as congestion control, driving conditions, curve speed, and others. The $5ms$ Omega latency also matches the maximum tolerable delay for many other edge applications, such as the value of $\sim 7ms$ required for Virtual Reality gaming [81], [82] and the $\sim 10ms$ needed for Augmented Reality apps [83].

C. 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 [65] to keep their state persistent. Also, all system use messages that are cryptographically signed using our protocol described in Section V-F. 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

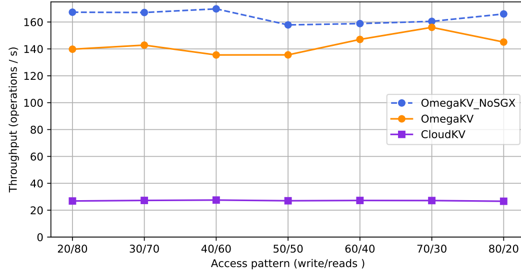

Fig. 9. Access pattern throughput (writes/reads).

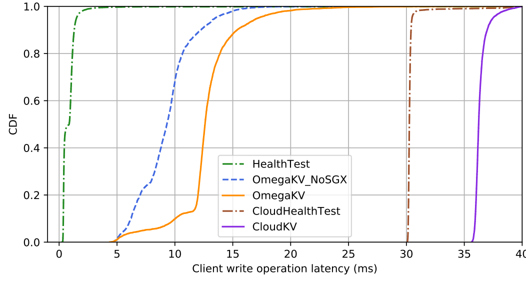

Fig. 10. Write operation latency of a fog node and cloud.

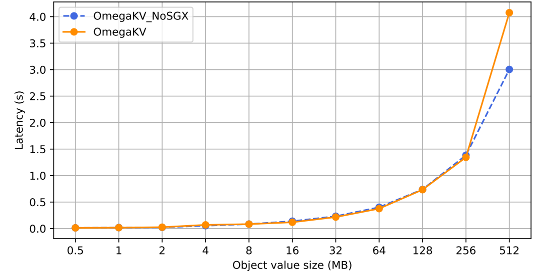

Fig. 11. Write operation latencies w/ and w/o SGX.

Vault), they make no effort to verify the integrity of stored data, and they do not need to use JNI interface.

Figure 9 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 10 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 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 [4].

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 11. 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 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. 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, 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 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] 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.

[2] Google, "Data center locations," https://www.google.com/about/datacenters/inside/locations/index.html, accessed: 2019-10-04.

[3] 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.

[4] 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.

[5] 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.

[6] L. M. 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.

[7] Cisco, "Cisco delivers vision of fog computing to accelerate value from billions of connected devices. press release," https://newsroom.cisco.com/press-release-content?type=webcontent\&articleId=1334100, Jan. 2014, accessed: 2019-10-04.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Hollywood, CA, USA, Oct. 2012.

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the ACM International Conference on Management of data*, Indianapolis, IN, USA, Jun. 2010.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proceedings of the USENIX Annual Technical Conference*, San Jose, CA, USA, Jun. 2013.

[15] A. Chandler and J. Finney, "On the effects of loose causal consistency in mobile multiplayer games," in *Proceedings of the ACM Workshop on Network and System Support for Games*, Hawthorne, NY, USA, Oct. 2005.

[16] 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.

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

[18] 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.

[19] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro, "Cure: Strong semantics meets high availability and low latency," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*, Nara, Japan, Jun. 2016.

[20] S. Almeida, J. a. Leitão, and L. Rodrigues, "Chainreaction: A causal+ consistent datastore based on chain replication," in *Proceedings of the ACM European Conference on Computer Systems*, Prague, Czech Republic, Apr. 2013.

[21] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, 2002.

[22] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Donostia-San Sebastian, Spain, Jul. 2015.

[23] P. Mahajan, L. Alvisi, M. Dahlin *et al.*, "Consistency, availability, and convergence," *University of Texas at Austin Tech Report*, vol. 11, 2011.

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

[25] 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.

[26] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, 1982.

[27] D. Malkhi and M. Reiter, "Byzantine quorum systems," in *Proceedings of the ACM Symposium on Theory of Computing*, El Paso, TX, USA, May 1997.

[28] B. Gold, R. Linde, R. Peeler, M. Schaefer, J. Scheid, and P. Ward, "A security retrofit of VM/370," in *Proceedings of the AFIPS National Computer Conference*, New York, NY, USA, Jun. 1979.

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

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

[31] 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.

[32] 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.

[33] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using airbox," in *Proceedings of the IEEE/ACM Symposium on Edge Computing*, Washington DC, USA, Oct. 2016.

[34] R. Ahmed, Z. Zaheer, R. Li, and R. Ricci, "Harpocrates: Giving out your secrets and keeping them too," in *Proceedings of the ACM/IEEE Symposium on Edge Computing*, Bellevue, WA, USA, Oct. 2018.

[35] 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.

[36] 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.

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

[38] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves," in *Proceedings of the European Symposium on Research in Computer Security*, Heraklion, Greece, Sep. 2016.

[39] 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 *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Savannah, GA, USA, Nov. 2016.

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

[41] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted

execution," in *Proceedings of the USENIX Security Symposium Security*, Vancouver, BC, CANADA, Aug. 2017.
[42] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, "Orbe: Scalable causal consistency using dependency matrices and physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing*, San Jose, CA, USA, Oct. 2013.
[43] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, "Gentlerain: Cheap and scalable causal consistency with physical clocks," in *Proceedings of the ACM Symposium on Cloud Computing*, Seattle, WA, USA, Nov. 2014.
[44] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, 1991.
[45] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Gers, France, Oct. 1988.
[46] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical physical clocks," in *Proceedings of the International Conference on Principles of Distributed Systems*, Cortina, Italy, Dec. 2014.
[47] B. Sanders, "The information structure of distributed mutual exclusion algorithms," *ACM Transactions on Computer Systems*, vol. 5, no. 3, 1987.
[48] M. Reiter and L. Gong, "Securing causal relationships in distributed systems," *Computer Journal*, vol. 38, no. 8, 1995.
[49] L. Alvisi and K. Marzullo, "Message logging: Pessimistic, optimistic, causal, and optimal," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, 1998.
[50] B. Lee, T. Park, H. Y. Yeom, and Y. Cho, "An efficient algorithm for causal message logging," in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, Oct. 1998.
[51] E. Ahmed and M. H. Rehmani, "Mobile edge computing: Opportunities, solutions, and challenges," *Pervasive Computing*, vol. 70, 2017.
[52] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara, "Cloudpath: A multi-tier cloud computing framework," in *Proceedings of the ACM/IEEE Symposium on Edge Computing*, San Jose, CA, USA, Oct. 2017.
[53] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, 1995.
[54] 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.
[55] 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.
[56] R. Krahn, B. Trach, A. Vahldiek-Oberwagner, T. Knauth, P. Bhatotia, and C. Fetzer, "Pesos: policy enhanced secure object store," in *Proceedings of the ACM European Conference on Computer Systems*, Porto, Portugal, Apr. 2018.
[57] 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.
[58] A. Nanda, D. Puthal, J. J. Rodrigues, and S. A. Kozlov, "Internet of autonomous vehicles communications security: overview, issues, and directions," *IEEE Wireless Communications*, vol. 26, no. 4, 2019.
[59] R. G. Engoulou, M. Bellaïche, S. Pierre, and A. Quintero, "Vanet security surveys," *Computer Communications*, vol. 44, 2014.
[60] C. Barba, M. Mateos, P. Soto, A. Mezher, and M. Igartua, "Smart city for VANETs using warning messages, traffic statistics and intelligent traffic lights," in *Proceedings of the 2012 IEEE Intelligent Vehicles Symposium*, Alcala de Henares, Spain, June 2012, pp. 902–907.
[61] M. S. Sheikh, J. Liang, and W. Wang, "A survey of security services, attacks, and applications for vehicular ad hoc networks (vanets)," *Sensors*, vol. 19, no. 16, 2019.
[62] R. S. Raw, M. Kumar, and N. Singh, "Security challenges, issues and their solutions for vanet," *International journal of network security & its applications*, vol. 5, no. 5, 2013.
[63] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of the ACM Symposium on Operating Systems Principles*, Stevenson, WA, USA, Oct. 2007.
[64] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010.
[65] Redis, "Key-value store," http://redis.io, accessed: 2019-10-04.
[66] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I can't believe it's not causal! scalable causal consistency with no slowdown cascades," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, USA, Mar. 2017.
[67] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, USA, Feb. 1999.
[68] L. Zhou, F. B. Schneider, and R. Van Renesse, "COCA: A secure distributed online certification authority," *ACM Transactions Computer Systems*, vol. 20, no. 4, 2002.
[69] ANSI, "X9.62-1998 public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA)," Sep. 1998.
[70] E. Barker and A. Roginsky, "Transitioning the use of cryptographic algorithms and key lengths," NIST, Special Publication 800-131 A r2, Mar. 2019.
[71] NIST, "FIPS 180-4, Secure Hash Standard," Aug. 2015.
[72] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu, "Policy-sealed data: A new abstraction for building trusted cloud services," in *Proceedings of the USENIX Security Symposium*, Bellevue, WA, USA, Aug. 2012.
[73] T. C. Group, "Tpm main specification level 2 version 1.2, revision 130," 2006.
[74] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of the Conference on the Theory and Application of Cryptographic Techniques*, Amsterdam, The Netherlands, Apr. 1987.
[75] 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.
[76] "AWS inter-region latency," https://www.cloudping.co/, accessed: 2020-03-02.
[77] G. Karagiannis, O. Altintas, E. Ekici, G. Heijenk, B. Jarupan, K. Lin, and T. Weil, "Vehicular networking: A survey and tutorial on requirements, architectures, challenges, standards and solutions," *IEEE communications surveys & tutorials*, vol. 13, no. 4, 2011.
[78] Y. C. 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.
[79] 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.
[80] J. Ott and D. Kutscher, "Drive-thru internet: Ieee 802.11 b for" automobile" users," in *Proceedings of the IEEE INFOCOM 2004*, Hong Kong, Mar. 2004.
[81] S. Mangiante, G. Klas, A. Navon, Z. GuanHua, J. Ran, and M. D. Silva, "Vr is on the edge: How to deliver 360 videos in mobile networks," in *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, Los Angeles, CA, USA, Aug. 2017.
[82] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, 2017.
[83] R.-S. Schmoll, S. Pandi, P. J. Braun, and F. H. Fitzek, "Demonstration of vr/ar offloading to mobile edge cloud for low latency 5g gaming application," in *Proceedings of the IEEE Consumer Communications & Networking Conference*, Las Vegas, NV, USA, Jan. 2018.