TÉCNICO LISBOA

Omega: a Secure Event Ordering Service for the Edge

Cláudio José Pereira Correia

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Luís Eduardo Teixeira Rodrigues
Prof. Miguel Nuno Dias Alves Pupo Correia

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. Bernardo Luis da Silva Ferreira

November 2019

Acknowledgments

Firstly, I thank my advisors, Professor Luís Rodrigues and Professor Miguel Correia, for welcoming me as their student and for their patient guidance throughout this work. I would also like to thank the members of the commitee for their constructive feedback especially to Professor Bernado Ferreira for comments on a previous version of this work.

I want to thank my parents for their support, safety, and encouragement throughout my studies all these years and without whom this project would not be possible. Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

Abstract

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

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

Keywords

Security; Edge Computing; Fog Computing; Trusted Execution Environment

Resumo

A computação na preferia é um paradigma que estende a computação na nuvem com armazenamento e processamento para próximo da preferia rede, com o objetivo de suportar aplicações sensíveis à latência, como realidade aumentada e jogos moveis. A computação na preferia geralmente é concretizada usando vários nós da neblina instalados em pequenos centros de dados, localizados em várias localizações geográficas. Estes nós da neblina estão mais vulneráveis a modificações ilegítimas do que os nós localizados em grandes centros de dados, sendo crucial proteger as suas funções de falhas maliciosas.

Um componente essencial de muitas aplicações distribuídas é um serviço de ordenação, capaz de capturar as dependências de causa efeito entre eventos e que permita processar os eventos numa ordem que respeite a causalidade. Nesta tese apresentamos um serviço seguro de ordenação de eventos para nós da neblina. O serviço, denominado Omega, tira partido de hardware confiável baseado na tecnologia Intel SGX para oferecer aos clientes garantias quanto à ordem em que seus eventos são aplicados e servidos, até mesmo na eventualidade do nó se encontrar comprometido. Adicionalmente, a tese apresenta e avalia o OmegaKV, um sistema de armazenamento chave-valor que usa o Omega para oferecer coerência causal. Resultados experimentais demonstram que, apesar dos custos associados à utilização do enclave, o Omega permite reforçar a segurança das aplicações sem violar os requisitos de latência das aplicações com constrangimentos temporais.

Palavras Chave

Segurança; Computação na Periferia; Computação em Neblina; Hardware Confiável

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

IoT	Internet of Things
PKI	Public Key Infrastructure
CA	Certificate Authority
AS	Attestation Server
TEE	Trusted Execution Environments
DNS	Domain Name System
ECC	Elliptic Curve Cryptography
TPM	Trusted Platform Module
CDF	Cumulative Distribution Function
OS	Operating System

1

Introduction

Contents

The number of edge devices that consume and produce information is growing at a fast pace (the set of these devices is often called the Internet of Things (IoT)). These devices can benefit from the availability of computing resources, placed near the edge of the network, that can cache data, execute computation, or aggregate information, and that can be reached with low latency. The computing resources may be provided by small data centers, known as fog nodes, that complement the services offered by the cloud, offering low access latency and reducing the amount of information that needs to be exchanged on the global network. This paradigm, known as edge computing, raises several interesting challenges, including security issues.

Due to their proximity to the edge, fog nodes are more exposed to tampering and therefore more vulnerable to attack. If a fog node is attacked and becomes malicious, it may compromise the correct execution of edge applications. Thus, security becomes an important factor and one that should be taken into consideration by developers of edge applications.

This thesis addresses the problem of *securing a middleware service for edge computing*. In particular, this thesis presents a novel design for a secure *event ordering service* that edge applications can use, thus obtaining security guarantees over stored data and computation on the edge of the network.

1.1 Motivation

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]. Also, many applications require data to be sent to a data center to be processed. For applications that produce large amounts of data, this model may require the consumption of significant network resources.

Many applications deployed in the cloud provide a range of services to clients that reside in the edge of the network: desktops, laptops, but also smartphones or even smart devices such as cameras or home appliances, also known as the 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]. Also, most of these devices have sensors that produce enormous quantities of information that need to be collected and processed [5].

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* [6]. 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* [7–9]. It assumes the existence of fog nodes that are located close to the edge. The number of fog nodes is expected to be several orders of magnitude larger than the number of data centers in the cloud.

Cloud nodes are physically located in secure premises, administered by a single provider. Fog nodes, instead, are most likely managed by several different local providers and installed in physical locations that are more exposed to tampering. Therefore, fog nodes are substantially more vulnerable to being compromised [10, 11], and developers of applications and middleware for edge computing need to take security as a primary concern in the design.

In this thesis, we address the problem of *securing a middleware service for edge computing*. Specifically, we focus on securing an *event ordering service* that is able to keep track of cause-effect dependencies among events and that allows events to be processed in an order that respects causality. The ability to keep track of causal relations among events is at the heart of distributed computing and, as such, an ordering service is a fundamental building block for many applications such as storage services [12], graph stores [13, 14], social networks [15], online games [16], among others. The idea of providing an event ordering service is not new (a notable example of such a service is Kronos [17]) but, to the best of our knowledge, we are the first to address the problem of providing secure implementations that may be safely executed in fog nodes.

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

To illustrate the use of Omega, and also to assess its performance in practice, we have built a key-value store, named OmegaKV, that offers causal consistency [18]. OmegaKV is a secure extension

of causal-consistent key-value stores that have been designed for the web, such as [12, 19–21]. 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 [22–24]. Clients of OmegaKV can perform write and read operations on data replicated by fog nodes, and are provided with the guarantees that writes are applied in causal order and that reads are also served in an order that respects causality.

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

1.2 Contributions

This thesis analyzes, implements and evaluates techniques to design a service that allows applications on the edge of the network achieve security guarantees over the events they generate. The main contribution of this thesis is the following:

- A novel secure event ordering service called Omega, which allows edge applications to generate causal consistent events on fog nodes. However, a fog node can become malicious and for this reason, Omega takes advantage of TEE to achieve a trust base inside the fog node. Applications use Omega to ensure that their events have integrity, freshness and data consistency.

1.3 Results

This thesis produced the following results:

- An implementation of Omega, which leverages trusted execution environment, specifically from Intel SGX enclaves, to get guarantees about the generated events. The use of enclaves in fog nodes allows for the correct generation of events even in the case the node operating system is compromised. Thus, Omega can provide causally consistent events for edge applications.

- An implementation of a key-value store named OmegaKV, which leverages Omega to generate secure events. Thus, OmegaKV can store data in the untrusted part of the fog node in a secure way.

- A detailed assessment of the proposed solution, including the overhead introduced by the use of the enclave and that Omega is still within the latency limits required by latency-sensitive edge applications.
- An experimental evaluation of the Omega implementation performance when used by OmegaKV, our edge storage service .

1.4 Research History

This work was developed in the context of the Cosmos research project, that aims at finding to techniques to offer causal consistent storage for edge computing scenarios. Techniques to increase the security of edge devices are expected to be a key component in the final COSMOS architecture.

In my work I have benefited from the useful feedback from the team members of COSMOS, both from INESC-ID Lisboa and from NOVA LINCS.

A paper that presents parts of this work has been published as:

> C. Correia, L. Rodrigues and M. Correia. Ordenação Segura de Eventos na Periferia da Rede. In *Actas do décimo primeiro Simpósio de Informática (Inforum)*, Guimarães, Portugal, Setembro de 2019.

This work was supported by FCT – Fundação para a Ciência e a Tecnologia, as part of the projects with ref. UID/CEC/50021/2019 and COSMOS (financed by the OE with ref. PTDC/EEI-COM/29271/2017 and by Programa Operacional Regional de Lisboa in its FEDER component with ref. Lisboa-01-0145-FEDER-029271).

1.5 Structure of the Document

The rest of the document is organized as follows. In Section 2 we introduce the relevant key concepts and related work with some techniques that can help achieve security guarantees on the edge. Section 3 describes the implementation of our Omega prototype and its components that will be considered for evaluation. Section 4 presents the results of our experimental evaluation of Omega and its use case OmegaKV. Finally, Section 5 concludes this document by highlighting the main finding and discussing future work.

2

Background and Related Work

Contents

This thesis addresses the problem of securing a middleware service that runs on edge servers, with emphasis on storage services that can offer causal consistency, a key component of the architecture envisioned by the COSMOS project. In this chapter we cover the main techniques that have been proposed to secure application on edge devices, algorithms that can be used to order events according to causality, and previous storage services for the edge, as these are the areas that are most relevant for our work.

2.1 Edge Computing and Fog Nodes

The emergence of IoT and the stress it places on services that operate in the cloud motivates the use of computing resources close to the edge. Edge computing is a model of computation that aims at leveraging the capacity of edge nodes to save network bandwidth and provide results with low latency [6]. 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 [7–9]. Fog nodes provide computing and storage services to edge nodes with low latency, setting the ground deploying resource-eager latency-constrained applications, such as augmented reality. These fog nodes are computers or clusters of computers with a reliable internet connection and ample computing resources, also known as cloudlets [25].

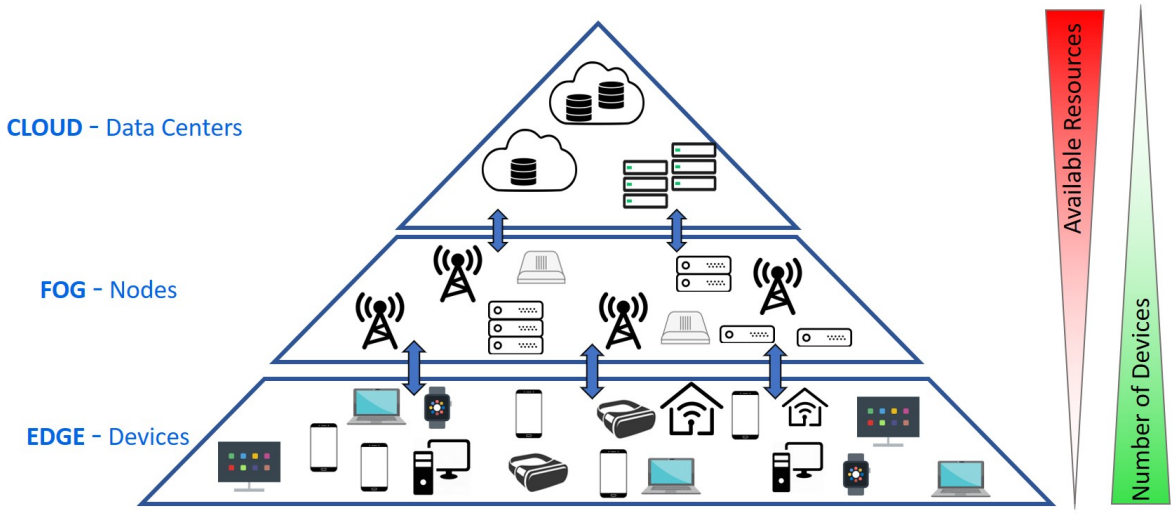

Figure 2.1: Three-layer fog computing architecture.

Figure 2.1 shows a general architecture of fog-edge computing, which consists of multiple infrastructures layers. The architecture has multiple layers. As we move down the stack, from the cloud to the edge, the number of nodes in each layer increases but the capacity of each of these nodes decreases. In the cloud layer, we may find data centers while in the fog layer we may find ISP servers, private data

centers, and 5G towers. Finally, in the edge, we may find all kinds of devices such as desktops, laptops, tablets, smartphones, sensors, and actuators.

The architecture suggests that, the devices located in the edge layer communicate mainly with those located in the fog layer. In addition, fog nodes also have to communicate with the cloud where critical data will be stored in a secure and persistent way.

2.1.1 Challenges and Requirements

One of the major motivations to deploy fog nodes close to the network edge is the ability to offer resources with low latency. Edge applications can use these services at a distance of a hop, emerging new applications such as augmented reality, real-time image processing, and collaborative applications, all latency-constrained. These applications require latencies below 30ms, leaving as a requirement for the fog computing model latencies between 5ms-30ms range [4].

A down side of placing fog nodes in close proximity to edge devices is their exposed location, that increases the risk of being attacked, raising the probability of such nodes to become malicious as discussed in the Zhang et al. [10], Mukherjee et al. [11] and Zhou et al. [26] surveys. If a fog node becomes compromised, it may attempt to corrupt the data it stores, discard stored information, serve faulty or stale data to the client. All these actions may leave edge applications in a faulty state. Therefore one of the major challenges in the fog computing model is how to deal with malicious fog nodes.

A fundamental objective is to provide an appropriate level of security without compromising the performance of the system, given that one of the main goals of supporting edge computing is to provide services with low latency to clients. Therefore in this document, we study security techniques to face the challenge of malicious fog nodes while maintaining the low latency requirements.

2.2 Securing Fog Services

In the fog computing model, fog nodes bring resources closer to the edge. However, the security guarantees offered by the cloud cannot be moved so easily. In the cloud model, data centers are isolated and well protected. On the other hand, fog nodes will be more exposed, and as a result, it is not possible to use the same techniques as in the previous model. The fact that fog nodes are dispersed among multiple geographic locations, close to the edge, increases the risk of being attacked and becoming malicious [10, 11, 26]. 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*:

1) Replication implies the use of multiple nodes, in the case a node becomes malicious a majority of nodes working correctly can mask the malicious node behavior.

2) Hardening implies the use of techniques to reduce the impact an attacker has on a single system, one such technique is TEE.

Following we have three sections, the first two sections discuss in more detail these two techniques, beginning with replication followed by hardening. The third section presents several systems that take advantage of hardening to offer security guarantees.

2.2.1 Replication

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 [27]) and, in some cases, to detect compromised nodes. Techniques such as Byzantine quorums [3, 28] 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. In the following paragraphs, we present a system that takes advantage of replication to mask malicious nodes.

2.2.1.A Minimal Byzantine Storage

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

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

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

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

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

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

A strong aspect of this work is that it relies on asynchronous communication and offers a protocol that maintains system availability in the case that *f* servers lose their correct behavior. A client has visibility when a server is misbehaving; he could inform the system if that is the case. However, this requires a few messages to be exchanged at every operation penalizing the system latency. Recall that a malicious fog node can be seen as a server with a Byzantine fault.

2.2.2 Hardening

Hardening [30] 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 in a way that passes unnoticed, effectively reducing the amount of damage an infected fog node can cause. A relevant mechanisms in this context is the use of a TEE, a secured execution environment with guarantees provided by the processor. The code that executes inside a TEE is logically isolated from the Operating System (OS) and other processes, providing integrity and confidentiality, even if the OS is compromised. Therefore, the use of a TEE is a natural choice to secure computation and sensitive data in fog nodes.

2.2.2.A Trusted Execution Environments

Edge and fog devices are a very compelling opportunity to use processors with Trusted Execution Environments (TEE) [31, 32] technology. With it, fog nodes can securely store private keys and offer clients a base of trust. If fog node providers choose to have this kind of technology, they will have additional security guarantees over the fog model.

A processor with this technology can operate in one of two modes, *normal mode* or *secure mode*. *Normal mode* is where the operating system and applications run without any special security guarantees, while in *secure mode*, application code has isolation, integrity, and confidentiality. This division between two different modes requires developers to build applications in two parts, one for each mode. One important goal of these architectures is enabling any application to take advantage of this secure environment. Therefore, device hardware configuration provides a TEE API, where applications in *normal mode* can request secure operations in *secure mode*. These requests require the processor to switch context between the two modes. When the processor switches from *normal mode* to *secure mode*, it

fetches the last state of the application that will run in *secure mode*, decrypts it and verifies its integrity. When it goes from *secure* to *normal mode*, it encrypts this state and stores it in memory. Between context switch, this state is encrypted and decrypted using a key that is secure in a chip element.

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

Examples of TEE technologies are: *AMD Secure Encrypted Virtualization (SEV)* that was recently introduced, it allows to protect system memory using encryption. However, it implies encrypting all the memory of an application occurring at a significant latency cost; *ARM TrustZone* that exists in processors focused on the mobile devices, offering low-power consumption. However, TrustZone does not offer confidentiality over application data; *Intel SGX* which is an architecture that already exists in many Intel processors. In the following paragraph we introduce in more detail Intel SGX. We chose the Intel SGX since it is a recent technology leading to significant new work in the field [33–36]. This increases its potential to be used in the real world by the industry, besides the fact that it is considered a practical solution to the fog (Intel Fog Reference Design [37]).

2.2.2.B Intel Software Guard Extensions

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* [38, 39]. The potential benefits of this technology for the fog have already been recognized by Intel [37] and it has already been used in practice [33, 40].

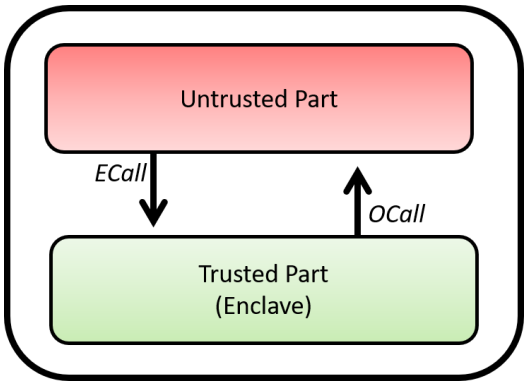

Figure 2.2: An Application in Intel SGX.

Applications designed to use SGX have two parts: an untrusted part (normal mode) and a trusted part (secure mode). Figure 2.2 shows a simplification of the Intel SGX design. 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 [41]. Such technology allows to store cryptographic keys in a secure way inside untrusted machines such as fog nodes and have guarantees about the code that is executed for example to generate metadata. A limitation of current SGX implementations is that the protected memory region, named enclave page cache, is typically limited to 128 MB [42]. 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 [43] and side channel attacks [44, 45] exist, Intel continues to investigate how to mitigate these issues [46]. Therefore, Intel SGX is still a good solution to be implemented inside fog nodes, offering a trust base for edge devices. In our work, such guarantees could help to secure cryptographic keys or execute secure functions.

2.2.3 Hardened Systems

In the following sections, we present several systems that take advantage of Intel SGX to provide security guarantees for both fog and cloud systems.

2.2.3.A Harpocrates

Content Distribution Networks (CDNs) are a geographically distributed network of proxy servers, which is located close to the users at the edge, and host static content. CDNs offer low latency and high scalability like fog computing and are already used for applications that provide static content, such as images and videos to users. However, providers of such CDNs are trying to evolve to fog computing and are implementing computation inside the CDN servers and one example of it is Cloudflare CDN.

Many CDNs work as a proxy for the origin web server, working as “man in the middle”. When a client does a DNS lookup for some server, the result may be the IP of a CDN server instead of the intended server. This technique hides the origin server IP and is reliable protection against DoS. However, for a CDN server to authenticate towards a user, it requires keys or other types of sensitive information, and the owners of applications may not fully trust the CDNs providers. To tackle this issue, Harpocrates [33] uses TEE, more specifically the Intel Software Guard Extensions (Intel SGX).

When a client intends to use the CDN service in a proxy for the first time, it requires mutual authentication between the CDN proxy and the client. For this authentication, some kind of secret is required on both the client side and the CDN proxy. The client is considered reliable, but CDN is not. Providers will have to store a secret within a CDN proxy they do not trust, which gives rise to a dilemma. This secret can be a private key to authenticate cookies or a symmetric key previously shared with the client.

Harpocrates solves this problem by taking advantage of the SGX enclave that guarantees the confidentiality of data stored within it. Thus, the proxy will have the application running in the untrusted part of the system and the secret will be stored inside the enclave.

In Harpocrates the main goal is to serve client requests at the edge while keeping secure the origin master secret. This secret is needed to authenticate the server towards clients. When the CDN proxy starts from a fresh boot, the enclave executes an initialization function that will contact the origin server. The origin server will use remote attestation to authenticate the code and the SGX if successful it will establish a secure channel with the enclave and send the master secret. When the application in the untrusted part needs to use the secret, it makes an ECALL to switch into the enclave and start the trusted execution. Inside the enclave, the secret is used to perform the requested function, then returns the output with an OCALL and returns to the untrusted part. This allows the CDN proxy to verify cookies and read messages content, without direct access to the master secret.

2.2.3.B ShieldStore

ShieldStore [34] is a secure key-value store that has been designed to operate in data centers at the cloud layer: it offers security guarantees for data stored in the key-value storage in scenarios where the cloud cannot be trusted. ShieldStore leverages Intel SGX to ensure the integrity and confidentiality of data stored in the cloud; the system works entirely within the enclave, never taking advantage of computation on the untrusted zone.

To access the storage service the client is responsible for performing attestation on the ShieldStore instance executing in a cloud node. If the attestation is successful, the client will establish a session key with the enclave that will be used to encrypt and authenticate the messages between the client and the enclave. All data on this storage system is stored outside the enclave due to its limited memory. However, ShieldStore must ensure the integrity and confidentiality of this data stored in the untrusted part of the cloud node.

As shown in the Figure 2.3, the data is stored outside the enclave. Whenever the enclave modifies some value, first the enclave must find the linked list corresponding to the value to be modified, for this it hashes the key. This hash is used to find a linked list that corresponds to this key, all key/values stored in the linked list correspond to the collisions that occur when the key is hashed. For each entry in a list, a MAC is created, and in each list these MACs are collected in set. Next a hash is calculated over all these MACs, and finally, a flat Merkle tree on these hashes is computed. The Merkle tree root is stored inside the enclave.

To persistently store data, Shildestore uses the current Intel SGX sealing mechanism, and additionally, the enclave is responsible to write a snapshot from the key-value store to disk. These operations have a strong impact on the latency. Also, for each operation, ShieldStore needs to access multiple

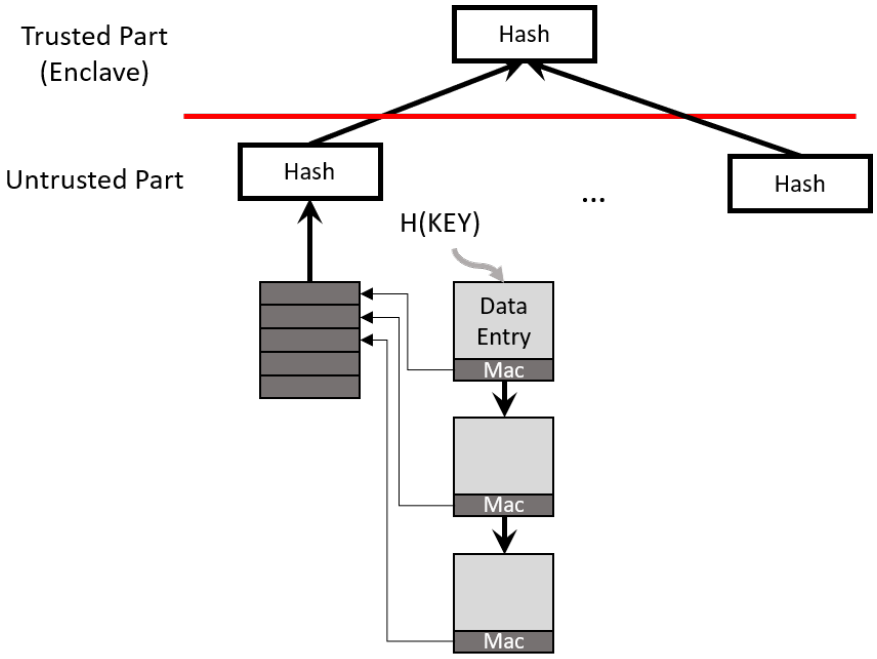

Figure 2.3: ShieldStore storage design.

MACs in order to compute a very large hash of these MACs. This hash can grow as large as the number of data entries in the linked list. In addition to this, ShieldStore is a key-value store that only offers *Read your Writes* consistency over the stored data.

2.2.3.C Speicher

Speicher [35] implements a Log-Structured Merge Tree (LSM)-based key-value store that takes advantage of Intel SGX to offer confidentiality, integrity, and freshness of data stored in a cloud node. Similarly to ShieldStore, Speicher stores data outside of the enclave. In the case of Speicher, the key/value entries are persistently stored on disk in SSTables. As shown in Figure 2.4, these SSTables are divided into blocks and a hash is calculated for each block. These ashes are stored at the end of the SSTable (SSTaple footer), in a special bock called the block of hashes. To protect the integrity of the block of hashes, an additional hash is calculated over this block, this new hash is then stored in a *manifest*. The manifest is a special data type that uses cryptographic operations and a trusted counter to ensure the integrity and prevent rollback attacks.

When a client requests a write operation, Speicher fetches the hash from the manifest, and checks if the block of hashes in the footer is correct, then it checks if the hash of the block corresponding to the input of the data is also correct, and if successful then writes the value into the SSTable, and recalculates the various hashes again. A reading operation only involves computing and checking these hashes.

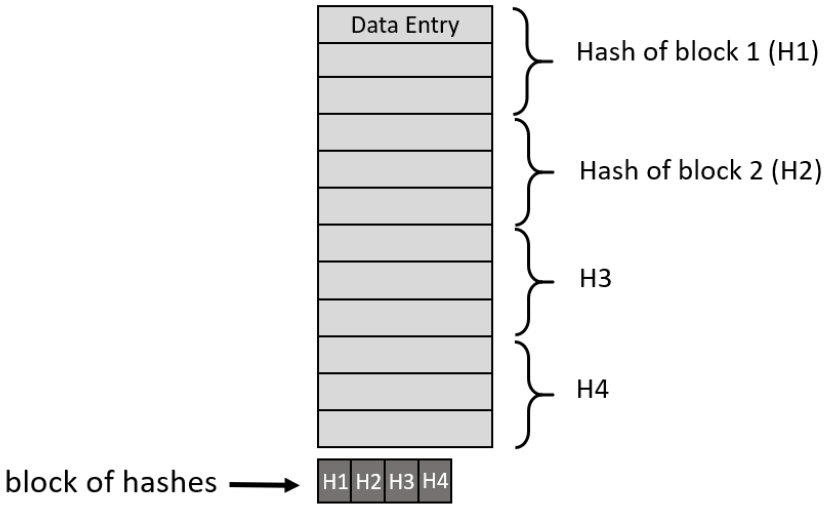

Figure 2.4: Speicher SSTable file format.

Speicher uses the Log-Structured Merge Tree to achieve a higher operation throughput, in level 0 of the LSM exists a table stored in memory with the values encrypted, and a hash of these values is stored in the enclave for integrity. For level 1 (last level in the case of Speicher) the SSTables are used. This implies that whenever there is a write operation that exceeds the in-memory table limit, the enclave must migrate data from memory to disk/SSTable. This suggests that some clients may experience significant latencies.

Finally, clients are responsible for attesting the service and establishing a direct channel to the enclave, once again not taking advantage of the computing that the untrusted party can offer. In [35], the authors report to have experimental evaluated a Merkle tree and that it was found to be an inefficient solution but no concrete numbers regarding this result are provided in the paper.

2.2.3.D Pesos

Pesos [36] is secure object store that also takes advantage of sIntel SGX. Pesos impose access policies on clients when they want to read or write objects stored in memory. The way Pesos ensures that these policies are enforced correctly is by taking advantage of the enclave. All access requests must go through the enclave, where the access policies are stored. Thus Pesos can restrict access to disk objects based on a declarative policy language.

A specific feature of Pesos is that it relies on a secure external component to persistently store objects; it uses Seagate Kinetic Disk [47] that allows it to securely store data in the disk. These disks provide an interface to establish a secure connection with authentication and confidentiality, using HTTP/S directly with the disk. This technique makes sense in the context of Pesos because it was designed with

the cloud in mind, where in the cloud is possible to tolerate the constant disks access.

When the enclave is started, an attestation service verifies that the enclave is running in the correct hardware and is running the correct code. If successful, this service provides the enclave with its encryption keys and credentials. This authentication data allows the enclave to connect to the Kinetic storage system and modify it in order to be the only entity with access to the storage disk. Thus the enclave is the only entity with disk access where the objects are located, forcing clients to always contact the enclave when they need to access an object. All objects stored on disk are encrypted with keys safely located in the enclave.

When a client intends to use the system, establishes an HTTPS connection with the enclave that takes advantage of the OpenSSL library, then the client can place requests to the enclave. All requests go through a Policy Interpreter that checks if the client in question can perform the requested operation, if so, then accesses the disk to fetch the object and returns it to the client.

The need to use Seagate Kinetic disks can be a limitation, and may prevent the broad adoption of Pesos. Additionally, Pesos needs to access the disk every time clients read or write, incurring in a significant latency burden, especially if the disks are not deployed on the same machine.

2.2.3.E Excalibur

Excalibur [48] is a service that allows to seal and unseal data based on predefined policies. This service is designed to operate on servers located in data centers and takes advantage of the Trusted Platform Module (TPM) [49], not SGX. TPM is a chip present in many mainboards, specified by Trusted Computing Group (TCG), providing security properties identical to the SGX, properties such as confidentiality and data integrity. It should be noted that although this section is dedicated to Intel SGX systems, we believe Excalibur should be discussed here, as this service implements an innovative technique to perform attestation that can also be used on Intel SGX systems.

In Excalibur, clients have two unique operations, seal and unseal data. To seal data, clients require two inputs: data and policies, and the output is a ciphertext. These policies are machine specific characteristics, such as: service = “EC2”; vmm = “CloudVisor”; version $\geq$ “1”; instance = ”large”. Besides the clients and the server exists a third entity, named *monitor*, responsible for distributing the keys to unseal the data among the servers, this entity first checks that the server machine has the necessary characteristics and then sends the cryptographic keys.

The way Excalibur ensures machines do not lie about their characteristics is by taking advantage of TPM. That is, the monitor performs attestation on all machines, with the attestation operation the monitor has access to a fingerprint of the hardware and software that the servers have, then the monitor can send the respective keys. The TPM guarantees the confidentiality and integrity of these keys, which then decrypts data upon client request.

In this paper clients can use servers resources with guarantees of their hardware and software without having to perform the attestation operation, this operation is responsibility of the monitor. This is an important factor in the fog since the attestation can have a big impact on the client's latency. However, it implies that clients attest to the monitor and that the monitor transfers cryptographic keys on the network.

2.3 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 [50, 51], Lamport's logical clocks [18], vector clocks [52, 53], hybrid clocks [54], 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 [55, 56].

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 [57, 58]. Second, it causes developers to maintain potentially complex code, that is duplicated in many slightly different variations, at different applications. The next paragraphs present two systems that offers an alternative approach.

2.3.1 Kronos

Kronos [17] offers event ordering as a service that can be used by multiple applications. It was designed for the cloud and does not implement any security measures. This system provides a simple abstraction for application developers to include event dependencies on their applications and later extract information from them. Kronos offers an API with 5 functions:

- *create_event():* the service stores the occurrence of a new event, and returns a unique identifier *e*.
- *acquire_ref(e):* given an event identifier *e*, increments the reference count on *e*.
- *release_ref(e):* decreases the reference counter for this event identifier *e*; when the counter reaches zero this event is deleted from the service.
- *query_order(e1, e2):* given two event identifiers, this function returns their relationship; either these events are dependent on each other (their causal relationship) or they are concurrent events.

- *assign_order(e1, order, e2, must/prefer):* Given two events the client can define what kind of relationship they have with each other, preferably (*prefer*) to maintain an order for the events if possible or if it is strictly (*must*) necessary for these events to have the proposed order.

In the paper [17], the authors provide three examples of how applications can use Kronos, namely: a Social Network, a Graph Store and a Transactional Key-Value Store. The Kronos implementation uses a directed acyclic graph, where the vertices represent the events created by the clients, and the edges represent the order in which the events are dependent on each other. Therefore Kronos allows clients to store and query information about the order of locally generated events. However, with time, some events are no longer needed, and with the creation of new events, the system can grow indefinitely, to solve this problem Kronos implements a garbage collector that eliminates unreferenced events in order to free space.

In the context of edge computing, a service such as Kronos can be used by many edge applications. If such a service can be built with security guarantees at the edge layer, edge applications using this service will inherit some of these security guarantees. Thus, it makes sense to focus efforts on securing a single service such as Kronos so that other applications can use it on the edge and obtain security guarantees.

2.3.2 Magpie

One area of research where ordering and generation of events are relevant is the analysis of distributed systems behavior. For this reason, in the following paragraphs we present a system that takes advantage of events to analyze systems behavior. Magpie [59] offers an approach where events are generated using instrumented code in kernel, middleware and application components. The objective of this work is to analyze the control flow and resource consumption of requests that applications perform. Analyzing the behavior of distributed applications makes debugging and optimizing easier for developers.

In this work, the application's code is instrumented to generate events along its critical path, allowing the analysis of the entire timeline of a request and to find where it wasted time or where an error occurred. In this work two applications were instrumented: Duwamish bookstore and Microsoft TPC-C Benchmark Kit. here are three major parts that were instrumented: the kernel to monitor when there is a thread exchange, I/O requests, and CPU consumption; the WinPcap packet capture library, to control network packets, so you can identify when a request is initiated and terminated; finally the application and middleware code, to know when resources are multiplexed or demultiplexed, and where the flow of control is transfered between components.

The paper considers also a problem that derives from the use of multiple threads where the OS has to share CPU time with multiple requests. This means that events in a given request are not continuously generated, and it is required to interpret all events and organize them. It is necessary to know which

events correspond to a specific request and in what order these events were generated, to create a timeline. For this, Magpie allows events to be created with a timestamp (processor cycle counter) and multiple attributes. Attributes can be thread id, Cpu Id, TcpIp and others. In the end, Magpie uses these attributes to merge multiple events that are part of a single request and uses the timestamp to order these events in causal order. The causal order is important because sometimes a request can have multiple process streams in parallel.

The technique of instrumenting code allows applications to generate events transparently, but it becomes more difficult for the application to use these events. Another downside is that its transparency makes it difficult to create events with application-specific parameters, such as an identifier that only makes sense in the application logic.

2.4 Edge Storage

To unleash their full potential, fog nodes should not only provide processing capacity, but also cache data that may be frequently used [60]; otherwise, the advantages of processing on the edge may be impaired by frequent remote data accesses [61]. By using cached data, requests rarely need to be served by data centers. Consequently, a fundamental service of edge-assisted cloud computing is a storage service that extends the one offered by the cloud in a way that relevant data is replicated closer to the edge. We are interested in systems that can offer causal consistency, because it has been shown that this is the strongest consistency model that can be offered without compromising availability [62].

Causal Consistency Data consistency assures developers about the state of geographically replicated data after undergoing multiple operations. With this guarantee, the development of applications that uses replicated data becomes easier and more efficient. In the context of replicated data access, we can divide data consistency into two broad categories: *strong consistency* and *weak consistency*.

In *strong consistency*, all replicas observe the same state. In other words, replicas apply all the updates in the same order, serializing the updates. Such a strict model comes at a heavy price. Replicas must coordinate to synchronize each update, severely penalizing the system availability. In fact, it has been proved that it is impossible offer simultaneously *strong consistency*, full availability, and tolerance to network partition, a fact known as the CAP theorem [63].

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

Causal consistency [18] is one of the most relevant *weak consistency* models. This consistency model captures the causal order between operations and enforces the visibility of this order in the entire

system. In other words, given two updates, *a* and *b*, if *a* potentially causes *b*, then in all replicates in the system it is only possible to observe the update *b* after update *a* is applied. The notion that *b* is causally dependent on *a* is denoted $a \rightsquigarrow b$.

Typical data stores located in the cloud offer *weak consistency*. Usually, these systems such as Saturn [19] and COPS [12] offer causal consistency. Gesto [64] also offers causal consistency but on the edge. Since the edge is still a current subject of research, we expect to see in the near future additional systems that support causal consistency in this setting. In the following paragraphs, we present in more details these three systems.

2.4.1 Saturn

Saturn [19] is a metadata distribution system at the data center level that ensures the causality of updates in a geo-replicated storage system. In Saturn, a client is required to connect to a data center before it can perform read and write operations. To ensure causal order of the updates, Saturn generates metadata for each update. The metadata associated with each update is called *label*. Data centers are responsible for generating labels for each update and propagating them to remote replicas and clients. Saturn assumes that replicas are organized in a tree for efficient metadata propagation.

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

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

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

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

and finally stores the label.

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

2.4.2 COPS

COPS [12] is another geo-replicated system at the data center level that offers causal consistency for a distributed key-value store.

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

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

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

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

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

2.4.3 Gesto

Gesto [64] is also a system that offers causal consistency on a data store, however its architecture was design for the edge layer. This system takes advantage of nodes close to the edge to build an architecture that can offer causality from the cloud to the edge. These nodes, which provide resources

and are outside the data centers, closer to the edge devices, are called cloudlets, conceptually similar to fog nodes.

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

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

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

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

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

Summary

In this chapter, we introduce some security challenges that exist for edge applications. Moreover, we present work that offers solutions that prevent a system from failing in the presence of a malicious node. We presented Kronos that can be a secure middleware for edge applications. We conclude the chapter by discussing work that provides causal consistency with various techniques, where using a scalar to order messages in a system is the most efficient solution.

In the next chapter, we introduce our Omega system, and discuss how some of the related work helped to converge on our solution. Our solution leverages Intel SGX for a trust base on fog nodes and implements an event-ordering service that uses timestamps to ensure causal consistency.

3

Omega

Contents

In this chapter, we introduce our Omega system that offers secure event ordering for edge applications. In Section 3.1 we present the goals that our system design has to achieve. Section 3.2 presents the threats to which fog nodes are exposed. Section 3.3 presents the design of the Omega service and Section 3.4 its implementation. Finally Section 3.5 presents an application named OmegaKV which uses Omega extensively to achieve security at the edge.

3.1 Design Goals

The goal of Omega is to function as middleware and offer secure data and computation for applications operating on the network edge. Therefore Omega aims to offer the following three security properties for edge applications:

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.

Data Consistency: A fog node cannot modify the causal order of events without being detected.

3.2 Violations of the Event Ordering

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

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

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

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

3.3 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 [65] of all timestamp requests it receives, defining a total order consistent with causality for all events occurring in the fog node.

3.3.1 Omega API

Table 3.1: The Omega API.

void *registerTag* (EventTag tag)	*Register a tag with Omega*
Event *createEvent* (EventId id, EventTag tag)	*Create a timestamped event with a given identifier and a given tag*
Event *orderEvents* (Event e_1, Event e_2)	*Order two events and return the first*
Event *lastEvent* ()	*Return the last event timestamped by Omega*
Event *lastEventWithTag* (EventTag tag)	*Return the last timestamped event with a given tag*
Event *predecessorEvent* (Event e)	*Return immediate predecessor of a given event*
Event *predecessorWithTag* (Event e)	*Return the most recent predecessor with the same tag*
EventId *getId* (Event e)	*Return the application level identifier of an event*
EventTag *getTag* (Event e)	*Return the tag associated with an event*
void *newEventNotification* (App app)	*Is callback to notify an application of a new event*

The interface of the Omega service is depicted in Table 3.1. Omega assigns, upon request, logical timestamps to application level events. Below we address the 3 main components that an event holds:

- *event identifier*: 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.
- *event tag*: 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 3.3.2, we provide examples that illustrate how tags can be used by different applications.
- *logical timestamp*: 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.

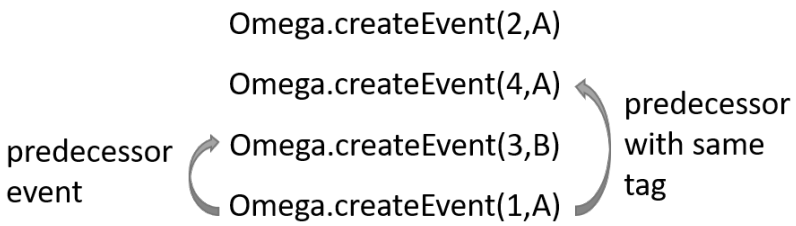

Figure 3.1: Example of *predecessorEvent* and *predecessorWithTag* parameters of an event.

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 3.1. The *getId* and *getTag* extract the application level event identifier and tag that have been securely bound with the target event. Finally the *newEventNotification* primitive allows an application running locally on the fog node to be notified whenever a new event is created, it only requires the application to subscribe using the primitive. Note that this primitive is only for applications running locally on the fog node, unlike the other primitives that serve remote clients.

Note that, although Omega is inspired by services such as Kronos [17], describe in Section 2.3.1, it offers an interface that makes different tradeoffs. First, it allows clients to associate events with specific objects / tags and to fetch all previous events that have updated that specific object; Kronos instead requires clients to crawl the event history to get the previous version of a particular object. Second, Kronos requires the application to explicitly declare the cause-effect relations among objects. This is more

versatile but more complex to use than Omega, that automatically defines a causal dependency among the last operation of a client and all operations that this client has performed or observed in its past. Finally, unlike Kronos, Omega automatically establishes a linearization of all operations, which simplifies the design of applications that need to fully order concurrent operations consistent with causality.

3.3.2 Example Use Cases

We now discuss how different applications, namely a messaging application, an online augmented-reality multiplayer game, and a distributed key-value store, can leverage an event ordering service such as Omega. These examples also serve as an illustration of how the API exported by Omega can be used for different purposes.

3.3.2.A Messaging Application

Messaging applications are widely used today, as standalone tools (e.g. WhatsApp, WeChat) or embedded in other applications (e.g. Facebook, Skype, Overleaf). Messaging applications can benefit from services provided by fog nodes, offering reduced latency to users located in the same region and reduce the amount of data to be propagated on the network by batching in a fog node.

We now briefly describe how a messaging application could be implemented to leverage fog nodes and how Omega could be used in this context. We assume the messaging application supports multiple chat groups and that all messages posted to a group are stored in encrypted form using cryptographic mechanisms that allow only group members to read their content (this can be a achieved by having all members share a secret key or by using more sophisticated techniques such as tree-based Diffie-Hellman [66]). Since posts to a group are encrypted, they can be safely cached in a fog node for quicker access by group members that access the chat from the same region. All posts are eventually propagated to the cloud. Also, clients locally cache their posts so they can replay them to another fog node (or directly to the cloud) in case the fog node they are using fails and loses its content. Although a compromised fog node can neither leak the content of the posts nor erase posts permanently (given that clients also cache their content), it might serve other clients with incomplete or truncated sequences of posts.

Omega prevents a fog node from serving incomplete sequences of posts to clients. The main idea is that clients use Omega to obtain ordered events that are associated with each post. In order to do this, when making a post, the client would create an *EventId* for the post as follows: i) the client would locally create an unique *postId* for the post; ii) the client would cypher the post content obtaining *cypheredContent*; iii) the client would compute *postHash* by applying a cryptographic hash function (e.g., SHA-256) to the tuple *(postId, cypheredContent)*; iv) the *eventId* for the post would be the tuple *(postId, postHash)*. The client would subsequently use the *createEvent* method of the Omega API, using the

Algorithm 1 Messaging application client algorithm

```
1:  function POST_MESSAGE(postId, cypheredContent, chatId)
2:      postHash ← hash(postId ⊕ cypheredContent)
3:      eventId ← postId ⊕ postHash
        ▷ Generates a new event cryptographically bind to the message id
4:      e ← omega.createEvent(eventId, chatId)
        ▷ Propagates message content or stores it in the untrusted zone of the fog node
5:      share_message(e, postId, cypheredContent)
6:  function RENDER_TIMELINE(messages, chatId)
        ▷ Receives the most recent event of the group chat
7:      e ← omega.lastEventWithTag(chatId)
8:      eventId ← omega.getId(e)
9:      m ← messages.get(eventId)
10:     sortedMessages.insert(m)
11:     while e ← omega.predecessorWithTag(e) do
12:         eventId ← omega.getId(e)
13:         m ← messages.get(eventId)
14:         sortedMessages.insert(m)
15:     return sortedMessages
```

chat group identifier as the *EventTag*. Finally, it would request the non-secured portion of the fog node to persistently store the tuple *(event, postId, chypheredContent)*.

When reading a chat group, the client would use the method *lastEventWithTag* to get the last post published to the chat group. It would subsequently use iteratively the method *predecessorWithTag* to get the preceding posts until it finds a post already cached in the client. The operation of the client is summarized in Algorithm 1.

3.3.2.B 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 and less propagated data. 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 are required

to 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 the 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).

3.3.2.C Key-Value Stores

Key-value stores are widely used in cloud computing today, and a large number of designs have been implemented [67–69]. Many 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, Saturn, or Gesto (presented in Section 2.4), 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 3.5, where we present OmegaKV.

3.4 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 the system face. Then, we describe in detail the most important aspects of the implementation, namely: how clients perform attestation (required to ensure that they interact with a secure Omega implementation), and the techniques used by Omega to preserve the integrity of its internal state (as we will discuss, the state can be large and needs to be preserved persistently).

3.4.1 System Architecture

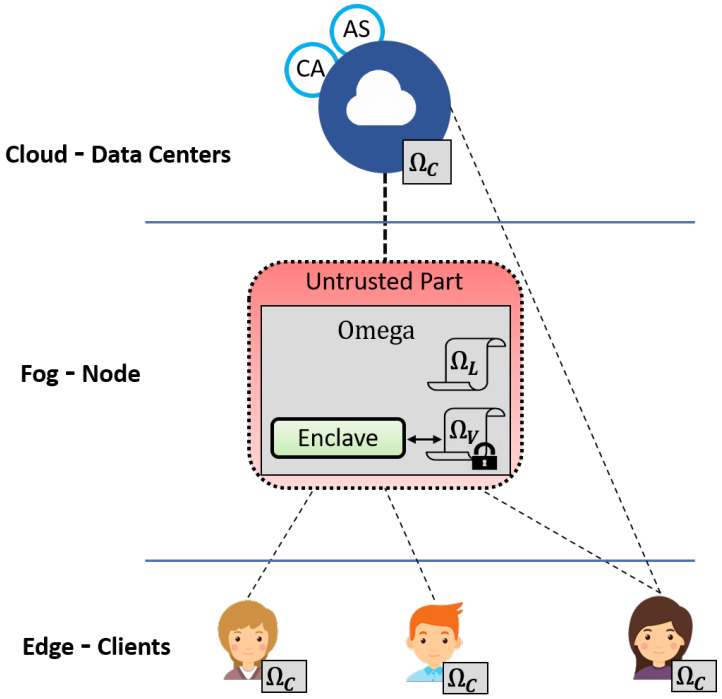

Figure 3.2: Omega architecture. CA is certificate authority, AS is attestation server, Ω_C is Omega client, Ω_V is Omega Vault and Ω_L is Omega Event Log.

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 3.2. 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 document. We can simply assume that cloud nodes are aware of all fog nodes (via some registration procedure) and the edge devices can find fog nodes using a request to the Domain Name System (DNS), e.g., using a name associated with the application, or to the cloud, e.g., using an URL associated with the application.

The implementation of Omega assumes the existence of two external components, that are executed in the cloud and are assumed to be secure. These components are a *Certificate Authority (CA)*, that is used to generate public key certificates (we assume the existence of a Public Key Infrastructure (PKI)), and an *Attestation Server (AS)*, which is used when a fog node binds to the Omega implementation via a

binding procedure (described in Section 3.4.4). The techniques used to ensure the correctness of these two external components are orthogonal to this work (e.g. Byzantine fault-tolerant replication might be used [70, 71]).

An important aspect of Omega is maintaining 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 3.2; 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 components:

- The event object, which has a digital signature generated within the enclave to provide authentication and integrity. (Section 3.4.3).
- A protocol that defines how Omega is launched and the required steps to offer its services with security guarantees (Section 3.4.4).
- 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 (Section 3.4.5).
- Two sub-components named “Omega Vault” and “Omega Event Log” that are used to maintain the Omega state (Section 3.4.6).
- An implementation of each method in the API (Section 3.4.7).

In the following sections, we begin by discussing the threat model, and then we introduce the Omega components in more detail.

3.4.2 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 [12, 17, 19–21]). 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 3.2. Both clients and fog nodes have asymmetric key pairs (K_u, K_r). The private key of the fog node K_r^E never leaves the enclave. For public key distribution, we consider the existence of a 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 [72] with 256-bit keys, which is recommended by NIST [73]. We assume the existence of a collision-resistant hash function. In practice we use SHA-256 [74], also recommended by NIST [73]. 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).

3.4.3 The Event Object

An important component of our system is the event object, which is used to keep track of the relative order of events. In the following paragraphs, we describe the content of the event object. Note that Omega users do not need to be aware of the internals of the object; they just need to use the object interface.

The event object is implemented as a tuple that contains the following fields:

1. A unique timestamp, that is associated to the event by the server (in the current implementation, this timestamp is a sequence number)
2. the *EventId* a unique identifier
3. The associated *EventTag* (that must be a tag previously registered via the *registerTag* event)
4. The *predecessorEvent* that is the event identifier of the last event generated by Omega
5. The *predecessorWithTag*, the event identifier of the most recent event generated by Omega with the same tag
6. A nonce
7. A digital signature signed with the privated key of the Omega server

The timestamp is a positive integer that allows to define an order in which events were created. This technique is similar to what Saturn and Gesto use. The predecessorEvent and the predecessorWithTag fields allow the client to quickly access events that may have some relationship depending on the application. These identifiers are maintained in the Omega vault, as described later in the text. The nonce

is only used when a client asks for a recent event. This ensures freshness for the request. The digital signature ensures that the event was generated inside the enclave and did not suffer any illegal changes by untrusted components of the architecture.

3.4.4 Service Initialization

Omega is a service that executes in a fog node. In order to be initialized, the service provider would typically configure the fog node to launch the Omega service on bootstrap. Omega runs as a daemon, where part of the code runs in unprotected mode and another part runs in the enclave. In the next paragraphs when we refer to the enclave, we are referring to the part of Omega that runs inside the enclave.

When the Omega code is started it is responsible for initializing the enclave inside the fog node. The Omega system performs the following steps at initialization (see also Algorithm 2):

1. The untrusted part of Omega calls the *sgx_create_enclave()* function to launch the enclave;
2. The untrusted part of Omega makes an ECALL that starts the attestation protocol within the enclave;
3. The untrusted part of Omega connects to the AS in the cloud and instructs the AS to perform the standard Intel attestation protocol of the enclave [38];
4. When the attestation completes, the AS generates a certificate, that includes the public key of the enclave. This certificate, is used as a proof that the AS has successfully attested the code running on the enclave.
5. The AS sends the certificate to the untrusted part of Omega. This certificate is later provided to clients as evidence that the Omega code that runs in the enclave as been attested,
6. The untrusted part of Omega opens a socket and starts listening for client's requests;

This certificate will then allow clients to have a guarantee that the enclave was successfully attested bye the AS. This scheme was inspired in Excalibur [48]. Using this certificate the clients delegate the attestation to the AS; this is possible because the cloud is considered trusted.

The AS runs Intel's attestation protocol with each fog node. If the fog node passes the attestation, the AS obtains from the CA a certificate with an expiration date, 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, the clients of

Algorithm 2 Omega initialization

```
1:  function INITIALIZEOMEGA
2:      #OUTSIDE ENCLAVE
    ▷ initializes empty structures in memory such as Mekle tree nodes and the tables
3:      (tree_pointer, table_pointer) ← untrustedStorage.init()
4:      OmegaLog.init();
5:      sgx_create_enclave()
6:      E_init(tree_pointer, table_pointer)
7:      INITIATE_INTEL_ATTESTATION()
8:
9:      certificate ← receiveCertificateFromAs()
10:     listener ← new ServerSocket()
11:     socket ← listener.accept()
12:     manageClient(socket)
13:     #OUTSIDE ENCLAVE
14:
15: function E_INIT(table_pointer, tree_pointer)
16:     #INSIDE ENCLAVE
    ▷ generates a fresh cryptographic key pair
17:     crypto.init()
    ▷ initializes with empty values the table and calculate a Merkle tree
18:     OmegaVault.init(table_pointer, tree_pointer)
19:     lastEvent ← new empty_event
20:     #INSIDE ENCLAVE
```

the Omega service just ask the Omega implementation to return the certificate that has been issued by the AS.

In this protocol, we assume that the cloud services AS and CA are correct and trusted, we also assume that the Intel attestation protocol is correct and trusted. Therefore, when the AS successfully completes the attestation we are assured that AS was able to verify that it is communicating with a real and legitimate Intel SGX enclave and that it is running the correct version of Omega. Additionally, when the attestation successfully completes a secret is established between the enclave and the AS, this key/secret is established via Diffie-Hellman Key Exchange. Using this secret a secure channel can be established between the enclave and AS. Through this secure channel, the enclave sends to the AS its public key (which was freshly generated after enclave's creation). This public key arrives at the AS encrypted with the secret, this ensures the authentication from the enclave that successfully passed the attestation. The CA then generates a certificate on this public key and returns it to the fog node (this certificate is visible in the Figure 3.3 as the last message from the CA).

This certificate is then sent to clients that use the service, a client with this certificate is able to verify the signatures from the enclave. These steps ensure the client that the enclave with the corresponding private key has been successfully attested and the client can trust all events signed with this private key.

At the end of the protocol described in the Figure 3.3 the system is the following state:

- The enclave has been launched;

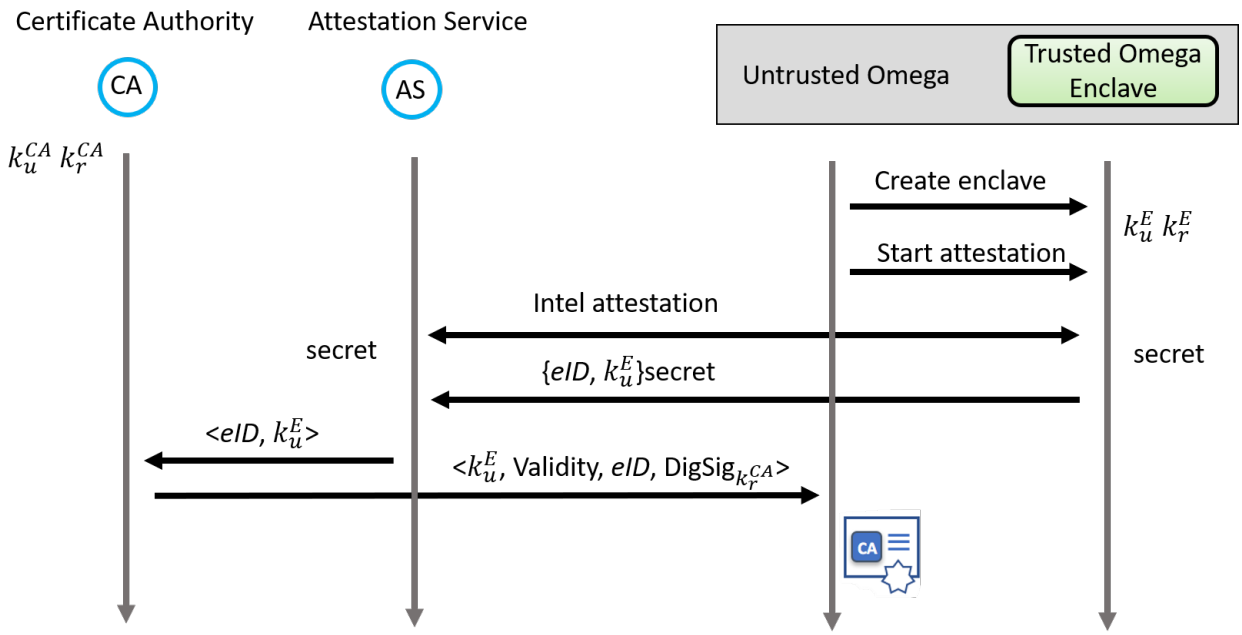

Figure 3.3: Omega attestation protocol. The secret is a symmetric key, eID is the enclave identifier, DigSig$_{K_r^{CA}}$ is the digital signature calculated over the entire message and signed with the key K_r^{CA}

- This enclave has been successfully attested;
- There is a certificate that proves that the AS successfully attested the enclave, and that includes the public key that matches the private key of the enclave;
- A daemon running in the untrusted zone waiting for client requests;

The certificate generated by CA, shown in Figure 3.3, has an expiration date. Expiration dates on certificates is part of PKI policies. This certificate can be renewed by the CA following PKI policies. The certificate renewal happens whenever the expiration date expires or if CA renews the certificate periodically.

Data stored inside the enclave is periodically dumped to disk using the sealing mechanism supported by SGX [38, 42], so that this data can be recovered when the system is rebooted (the period is configurable). The sealing mechanism allows encrypting data with a symmetric key inside the enclave, then storing it outside of the enclave, e.g., in disk for persistence, which is the reason why Omega uses it. The data that is dumped is essential for the correct functioning of Omega. This data is the private key of the fog node, the Merkle tree root hash (explained later in Section 3.4.6), and the last generated event. The sealing mechanism is used with the MRSIGNER (identity of the enclave author) key policy [42] to encrypts the data. This policy allows other instances of an enclave with the same software to unseal data stored on disk, meaning that when the enclave is launched again it would be able to recover this data. The symmetric key in the case of the MRSIGNER policy is generated from CPU key material and

vendor software information. This mechanism is not yet implemented in the current prototype.

The communication between the enclave and the client is based on the client sending requests messages with parameters, these messages are always accompanied by digital signatures signed with the client's private key. The enclave can then authenticate these request messages. Enclave responses are always in the form of an event, and events are always accompanied by digital signatures signed with the enclave's private key allowing clients to verify the integrity and authentication of events from the enclave. Note that all requests must go through the untrusted zone, the untrusted zone is responsible for maintaining the socket with the client and for forwarding the requests to the enclave and back. If the untrusted zone fails in this job, the client will detect that it is not receiving responses to its requests.

3.4.5 Client Binding

Before a client invokes any method of the Omega API, it has to execute a *client binding* procedure. For this, the client needs to find a fog node to connect to. The client can make a DNS request to acquire the node Internet Protocol (IP) address or by directly asking the cloud which is the nearest fog node. After acquiring the IP address of the fog node the client opens a socket and establishes a connection with the untrusted component of the Omega service.

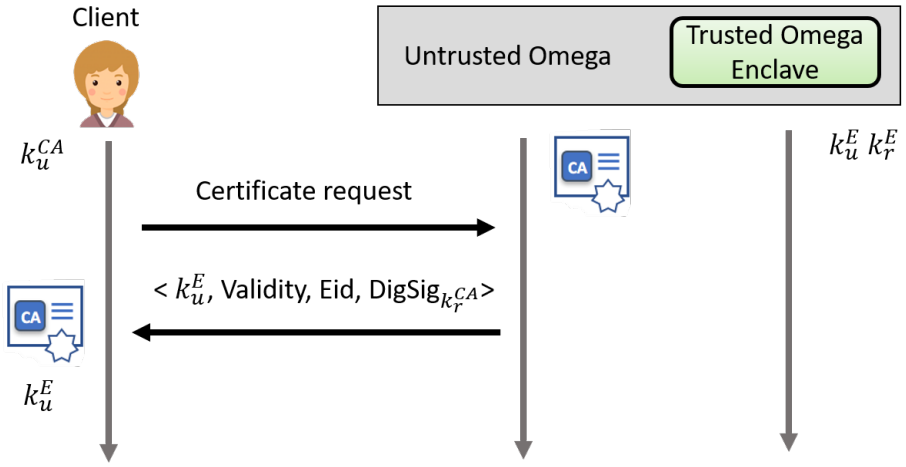

Figure 3.4: Client certificate request. Th eID is the enclave identifier, DigSig$_{K_r^{CA}}$ is the digital signature calculated over the entire message and signed with the key K_r^{CA}

After opening a socket with the fog node, the client initiates the binding procedure. The purpose of the binding procedure is to allow the client to obtain 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). The steps that are required to achieve these goals are know as the attestation procedure [38]. One limitation of the attestation proce-

dure defined by Intel is that it involves multiple communication steps, it may even include 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 also referred in the Section 3.4.4. In this procedure, the client delegates the attestation to the AS.

Instead of running the Intel's attestation procedure, the clients of the Omega service just ask the Omega implementation to return the certificate that has been issued by the AS. Note that clients could also obtain the certificate by contacting the cloud services, in particular by contacting the AS directly. However, by letting Omega to store a copy of the certificate in the fog node, the client can obtain the certificate with a lower latency. Figure 3.4 illustrate this step of the binding procedure.

The client can then use the Omega API, all client requests are accompanied by a digital signature created with the client's private key. The client digital signature is then verified within the enclave, and if it is correct the request is accepted and executed. The enclave always responds with an event object signed with the private key stored inside the enclave, when the client receives the response from request it can verify that it has been accepted and executed by the enclave because only the enclave could generate such a signature. Please note that events are the response from the enclave and that the client can be certain that these events are indeed from the enclave by verifying the digital signature that follows these events.

3.4.6 Storage on Untrusted Memory

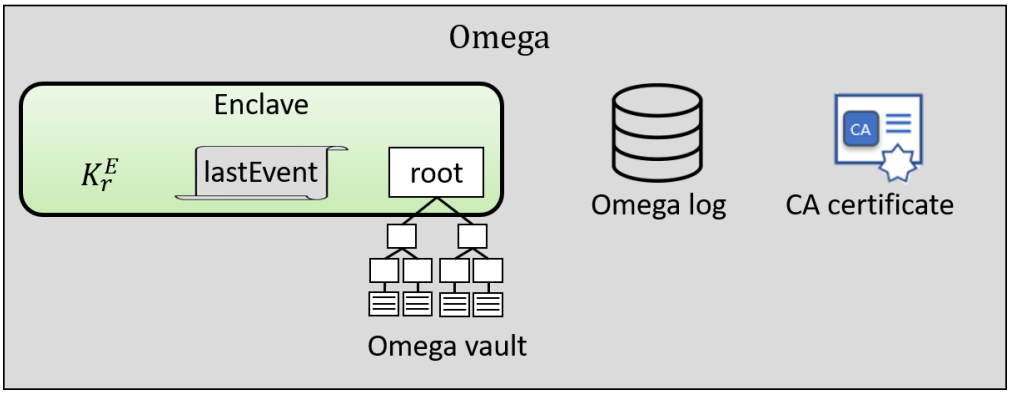

Figure 3.5: Omega storage components.

Omega is required to safely store different pieces of information, such as its private and public keys K_r^E and $K_r u^E$ (the public key is included in the certificate signed by the CA), 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 needs a way to securely store the above information (in particular the last event for an arbitrary number of tags).

Also, Omega must have access to events it has generated in the past, given that clients can use the *predecessorEvent* method to crawl the event history.

To satisfy these requirements, Omega uses two storage services with different properties, the *vault* and the *Event Log*. The location of these components is illustrated in Figure 3.5. 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. That is, Event Log stores all events that are generated and Omega Vault stores the most recent events of each tag. We now describe the implementation of these two storage services.

3.4.6.A Event Log

The *Event Log* is a record of all events generated. We decided to implement this component on top of a a key-value store, where events are stored using the unique identifier assigned by the application as key. The Event Log is stored outside the enclave, on the untrusted part of the fog node. Since all events have a digital signature, the integrity of a single event can be verified.

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.

As will be shown in the Section 3.4.7, the interaction with the Event Log is performed using the Omega components running in the untrusted zone.

3.4.6.B Omega Vault

When the client uses the *lastEventWithTag* function, the Omega should return the most recent event of a given tag. This function could be implemented using the Event Log, by parsing the log, checking the validity of the predecessor event at each step, until the last event associated with the target label is found. Unfortunately, this procedure is very inefficient. Therefore we have implemented another mechanism that helps in finding the target event quickly. The purpose of the Omega Vault is to store the last event associated with each tag.

The *vault* is somehow harder to implement than the Event Log, because it 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 were not tampered. At the logical level, this is achieved by requiring the enclave to hash the vault every time it updates its content and to store the hash in the enclave itself. However, a naive implementation that

would actually keep a single hash for the entire vault would not perform well because, as we have noted, the application may use a large number of tags and computing a hash of all these tags may take a long time. Also, it is not straightforward to ensure that the hash function yields the intended value if the values being hashed are to many to fit inside the enclave and may be changed by the adversary while the hash is being computed.

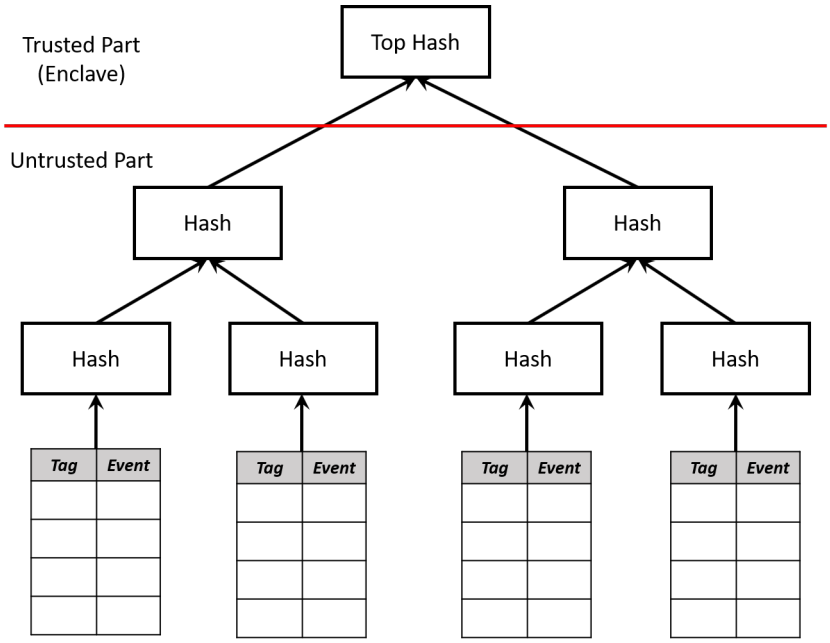

Figure 3.6: Merkle tree used in the Omega vault that is stored in the untrusted zone of the fog node.(with $N = 4$).

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* [75]. 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 3.6); 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. Using a pointer to untrusted memory the enclave can manipulate data outside the enclave's memory, it can verify and generate the Merkle tree hashes when needed requiring only to store the top hash in enclave's memory.

When the enclave has to modify one part of the table it needs to: compute the Merkle tree to verify the data, then change the data and, finally, recalculate a few of the Merkle tree hashes (as many as the depth of the tree). These operations must be performed in an atomic manner, otherwise an attacker could change the table between the two Merkle tree calculations and the enclave would not be able to detect

it. To ensure the atomicity of the combined operations, the enclave maintains two different contexts that allows to calculate two different hashes, this is possible by using the variable sgx_sha_state_handle_t from the SGX SDK [42]. These two contexts are feed with the same input data throughout the table iteration except for the entry that has been modified. Thus the enclave is assured that the old and new hashes were calculated on the same data.

Besides, it should be noted that both the Merkle tree and the leaves are in memory. The enclave access this data in memory and modifies it in memory, the untrusted part is responsible for asynchronously write to disk all changes made by the enclave. This, unlike the techniques proposed in papers such as [34–36], prevents the enclave from having to access the disk, which has a latency penalty.

If a fog nodes becomes compromised and becomes malicious, the untrusted part can modify these values in memory and/or block the communication between the enclave and the clients and/or the cloud. A client that fails to establish communication with the enclave on a given fog node can report this fact to the cloud. These accusations can be used by the cloud to detect the failure of the fog node and initiate corrective measures.

3.4.7 Implementation of the Omega API

Clients invoke the Omega API via a client library. In this way, the clients do not need to be aware of the specifics for communication with the Omega server. Also, some of the methods can be executed locally 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. Note that in the following algorithms there is a division of Omega Vault into two parts: *storage* and *merkleTree*. This division is just for ease of programming, where the *storage* contains the logic of the tables, i.e. the content that belongs to the Merkle tree leaves. The *merkleTree* part contains all the Merkle tree logic.

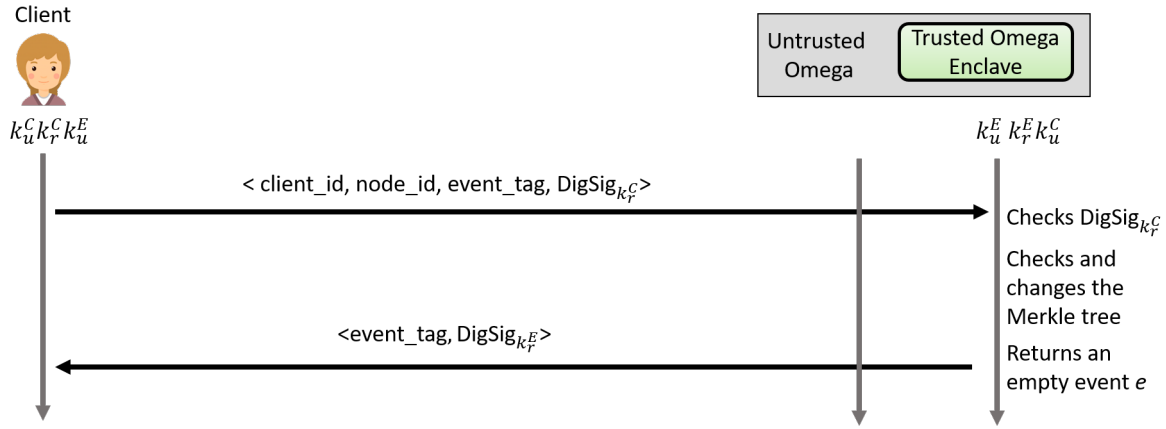

Figure 3.7: Communication pattern for the function *registerTag*.

Algorithm 3 Implementation of the function *registerTag* in Omega that executes in the fog node

```
1: function REGISTERTAG(cliente_id, node_id, event_tag, signature)
2:     #OUTSIDE ENCLAVE
   ▷ allocate space in memory for the new table
3:     table_pointer ← untrustedStorage.addNewTables(event_tag)
   ▷ Ecall into the enclave
4:     empty_event ← E_registerTag(cliente_id, node_id, event_tag, signature, table_pointer,
   tree_pointer)
5:     return empty_event
6:     #OUTSIDE ENCLAVE
7:
8: function E_REGISTERTAG(cliente_id, node_id, event_tag, signature, table_pointer, tree_pointer)
9:     #INSIDE ENCLAVE
   ▷ verifies the signature and that is not a repeated event_tag
10:    status ← crypto.verifySignature(cliente_id, node_id, event_tag, signature)
   ▷ if signature is incorrect return error
11:    if status != true then
12:        return error
13:
14:    new empty_event
15:    empty_event.event_tag ← event_tag
16:    omegaVault.registerTag(empty_event, table_pointer, tree_pointer)
17:
18:    empty_event.sig ← crypto.getSignature(empty_event)
19:    return empty_event
20:    #INSIDE ENCLAVE
21:
22: function OMEGAVAULT.REGISTERTAG(empty_event, table_pointer, tree_pointer)
23:    #INSIDE ENCLAVE
   ▷ generates new leaf hash fill with empty events
24:    (newEmptyLeafHash) ← storage.newTag(table_pointer, empty_event)
   ▷ doubles the tree size if needed
25:    correct ← merkleTree.newTag(newEmptyLeafHash, tree_pointer, empty_event)
   ▷ if the data was modified illegally in the untrusted zone return error
26:    if correct != true then
27:        return error
28:    #INSIDE ENCLAVE
```

The methods *registerTag* and *createEvent* are the only methods that modify the state of the Omega server in the fog node. We describe the implementation of these two methods next.

The method *registerTag* registers tags, so it has to adjust the space allocated to the vault when the space available for storing past events is exhausted (recall that Omega keeps track of the last event associated with each tag it observed); this mechanism is explained in the Section 3.4.6.B. When new tags are created, it is necessary to create new tables in memory and increase the Merkle tree so that it is possible to save the most recent event of the new tags, this is summarized in Algorithm 3. An empty event is used to respond to the client as a guarantee that the enclave received and execute the request. The communication for the *registerTag* function is illustrated in Figure 3.7.

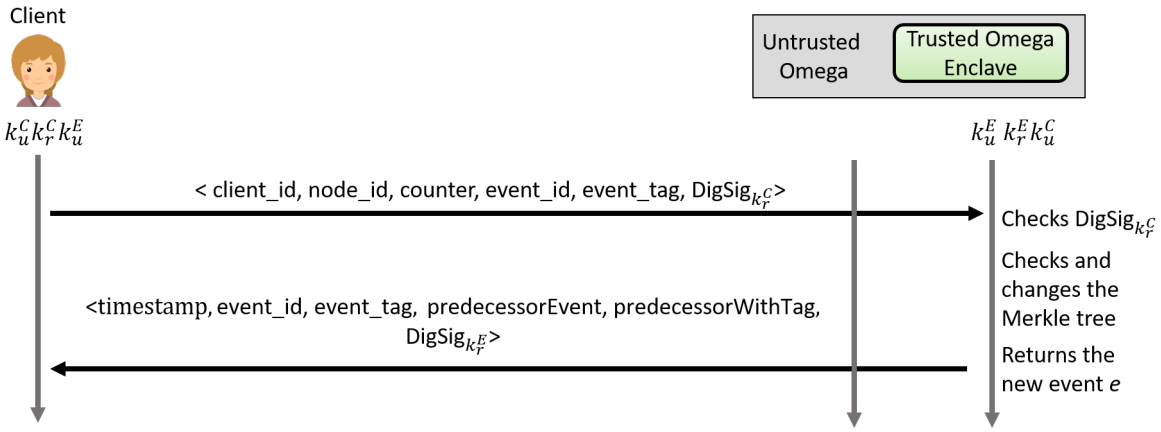

Figure 3.8: Communication pattern for the function *createEvent*.

The method *createEvent*, summarized in Algorithm 4, is used to create a new event in the server. When the server receives this request it creates a new event and fills ite fields of the event structure as follows (the fields for an event object are described in Section 3.4.3). The enclave accesses the Omega Vault to fetch the predecessorWithTag value. The value of the predecessorEvent field is taken from the *lastEvent* variable in enclave memory. Subsequently, the Omega server replaces the old identifier for the new one in the Omega vault (for predecessorWithTag) and the *lastEvent* in memory for this new event. The new tuple is signed with the private key of the Omega server. 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 communication steps required to execute the *createEvent* function are depicted in Figure 3.8.

The four methods *lastEvent*, *lastEventWithTag*, *predecessorEvent*, and *predecessorWithTag* do not change the state of the Omega.

When the server receives a *lastEventWithTag* request, it extracts the last event it has processed form the vault with the respective tag, concatenates a nonce and generates a digital signature before returning it to the client; these steps are listed in Algorithm 5. The interaction between the client and the

Algorithm 4 Implementation of the function *createEvent* in Omega that executes in the fog node

```
1: function CREATEEVENT(cliente_id, node_id, counter, event_id, event_tag, signature)
2:     #OUTSIDE ENCLAVE
3:     table_pointer ← untrustedStorage.getTablePointer(event_tag)
   ▷ Ecall into the enclave
4:     e ← E_createEvent(cliente_id, node_id, counter, event_id, event_tag, signature, table_pointer,
   tree_pointer)
   ▷ store the new event in the Omega Log
5:     omegaLog.store(e)
6:     notifyAppsNewEvent(e)
7:     return e
8:     #OUTSIDE ENCLAVE
9:
10: function E_CREATEEVENT(cliente_id, node_id, counter, event_id, event_tag, signature,
   table_pointer, tree_pointer)
11:     #INSIDE ENCLAVE
   ▷ verifies the signature and that is not a repeated counter for this client
12:     status ← crypto.verifySignature(cliente_id, node_id, counter, event_id, event_tag, signature)
   ▷ if signature is incorrect return error
13:     if status != true then
14:         return error
15:
16:     new e
17:     e.timestamp ← getNewTimestamp()
   ▷ event_id is unique and therefore a nonce is not required for freshness
18:     e.eventID ← event_id
19:     e.eventTag ← event_tag
20:     e.predecessorEvent ← lastEvent.eventID
21:     e ← omegaVault.createEvent(event, table_pointer, tree_pointer)
22:
23:     lasEvent = e
24:     e.sig ← crypto.getSignature(e)
25:     return e
26:     #INSIDE ENCLAVE
27:
28: function OMEGAVAULT.CREATEEVENT(event, table_pointer, tree_pointer)
29:     #INSIDE ENCLAVE
   ▷ change value and simultaneously calculate old and new table hash
30:     (oldLeafHash, newLeafHash, predecessorWithTagID) ← storage.changeTable(table_pointer,
   event)
31:     event.predecessorWithTag ← predecessorWithTagID
   ▷ simultaneously verify if the tree is correct and generate the new tree
32:     correct ← merkleTree.verifiyAndChange(tree_pointer, oldLeafHash, newLeafHash, event)
   ▷ if the data was modified illegally in the untrusted zone return error
33:     if correct != true then
34:         return error
35:     return event
36:     #INSIDE ENCLAVE
```

Algorithm 5 Implementation of the function *lastEventWithTag* in Omega that executes in the fog node

```
1: function LASTEVENTWITHTAG(cliente_id, node_id, nonce, event_tag, signature)
2:     #OUTSIDE ENCLAVE
3:     table_pointer ← untrustedStorage.getTablePointer(event_tag)
   ▷ Ecall into the enclave
4:     e ← E_lastEventWithTag (cliente_id, node_id, nonce, event_tag, signature, table_pointer,
   tree_pointer)
5:     return e
6:     #OUTSIDE ENCLAVE
7:
8: function E_LASTEVENTWITHTAG(cliente_id, node_id, nonce, event_tag, signature, table_pointer,
   tree_pointer)
9:     #INSIDE ENCLAVE
10:    status ← crypto.verifySignature(cliente_id, node_id, nonce, event_tag, signature)
   ▷ if signature is incorrect return error
11:    if status != true then
12:        return error
13:
14:    e ← omegaVault.lastEventWithTag(event_tag, table_pointer, tree_pointer)
15:    e.nonce ← nonce
16:    e.sig ← crypto.getSignature(e)
17:    return e
18:    #INSIDE ENCLAVE
19:
20: function OMEGAVAULT.LASTEVENTWITHTAG(event_tag, table_pointer, tree_pointer)
21:    #INSIDE ENCLAVE
   ▷ get event from table and calculate leaf hash
22:    (currentLeafHash, e) ← storage.getEvent(table_pointer, event_tag)
   ▷ verify if the tree is correct
23:    correct ← merkleTree.verifiy(tree_pointer, currentLeafHash, e)
   ▷ if the data was modified illegally in the untrusted zone return error
24:    if correct != true then
25:        return error
26:    return e
27:    #INSIDE ENCLAVE
```

Algorithm 6 Implementation of the function *lastEvent* in Omega that executes in the fog node

```
1:  function LASTEVENT(cliente_id, node_id, nonce, event_tag, signature)
2:      #OUTSIDE ENCLAVE
    ▷ Ecall into the enclave
3:      e ← E_lastEvent(cliente_id, node_id, nonce, event_tag, signature)
4:      return e
5:      #OUTSIDE ENCLAVE
6:
7:  function E_LASTEVENT(cliente_id, node_id, nonce, event_tag, signature)
8:      #INSIDE ENCLAVE
9:      status ← crypto.verifySignature(cliente_id, node_id, nonce, event_tag, signature)
    ▷ if signature is incorrect return error
10:     if status != true then
11:         return error
12:
    ▷ lastEvent is always in enclave's memory
13:     e ← lasEvent
14:     e.nonce ← nonce
15:     e.sig ← crypto.getSignature(e)
16:     return e
17:     #INSIDE ENCLAVE
```

omega server is depicted in Figure 3.9.

When a *lastEvent* request is received, the server simply concatenates a nonce to the event and generates a digital signature (the last event generated in the enclave is always within the enclave's memory) and returns it to the client, these steps are listed in Algorithm 6. The communication for the *lastEvent* function is depicted in Figure 3.10.

The requests *predecessorEvent* and *predecessorWithTag* are executed collaboratively by the client library and by the server. The client library, that is aware of the internal structure of the *Event* tuple, extracts the identifier of the desired 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. Both functions are listed in Algorithm 7 and the their communication pattern is depicted in Figure 3.11. These functions verify signatures in the untrusted zone, this ensures that as long as the fog node is not compromised only authorized clients can access events.

For ease of programming, we have used a Java layer in Omega, that runs outside of the enclave; this allows to implement the *newEventNotification* primitive in Java. The *newEventNotification* primitive takes an object as input and adds it to a list, as this function is implemented in Java an application can simply pass itself as an object using the parameter *this* in Java. Each time an event is created, Omega iterates through all objects in this list and calls the *newEventNotification* method; this is illustrated in Figure 8.

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

Algorithm 7 Implementation of the functions predecessorEvent and predecessorWithTag in Omega that executes in the fog node

```
1:  function PREDECESSOREVENT(cliente_id, node_id, event_id, signature)
2:      #OUTSIDE ENCLAVE
3:      status ← crypto.verifySignature(cliente_id, node_id, event_id, signature)
        ▷ if signature is incorrect return error
4:      if status != true then
5:          return error
6:      e ← omegaLog.getEvent(event_id)
7:      return e
8:      #OUTSIDE ENCLAVE
9:
10: function PREDECESSORWITHTAG(cliente_id, node_id, event_id, signature)
11:     #OUTSIDE ENCLAVE
12:     status ← crypto.verifySignature(cliente_id, node_id, event_id, signature)
        ▷ if signature is incorrect return error
13:     if status != true then
14:         return error
15:     e ← omegaLog.getEvent(event_id)
16:     return e
17:     #OUTSIDE ENCLAVE
```

Algorithm 8 Implementation of the function *newEventNotification* in Omega that executes in the fog node

```
1:  function NEWEVENTNOTIFICATION(app)
2:      #OUTSIDE ENCLAVE
3:      allSubscriberList.add(app)
4:      #OUTSIDE ENCLAVE
5:
6:  function NOTIFYAPPSNEWEVENT(new_event)
7:      #OUTSIDE ENCLAVE
8:      for app in allSubscriberList do
9:          app.newEventNotification(new_event)
10:     #OUTSIDE ENCLAVE
```

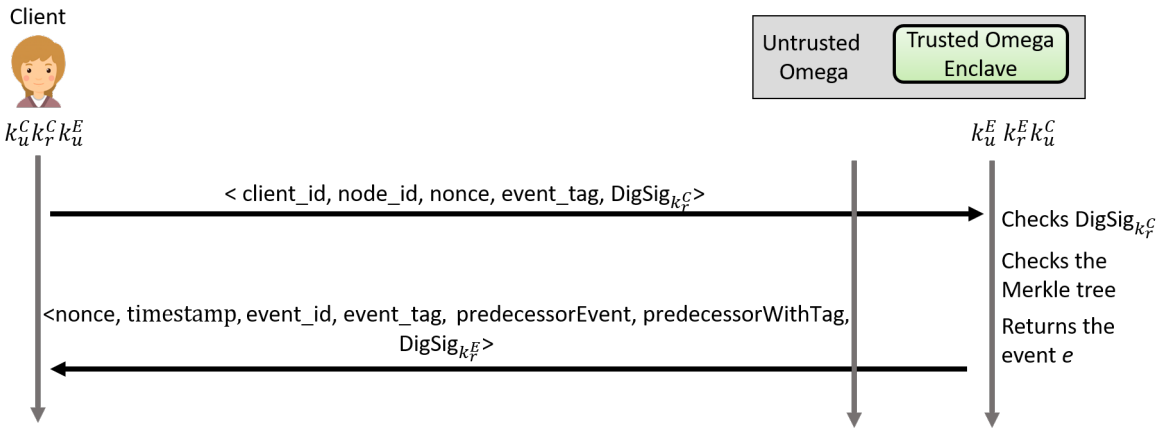

Figure 3.9: Communication pattern for the function *lastEventWithTag*.

Figure 3.10: Communication pattern for the function *lastEvent*.

tuple, compares their value, and return 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. As we describe before, 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 event is signed with the private key of the server that 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.

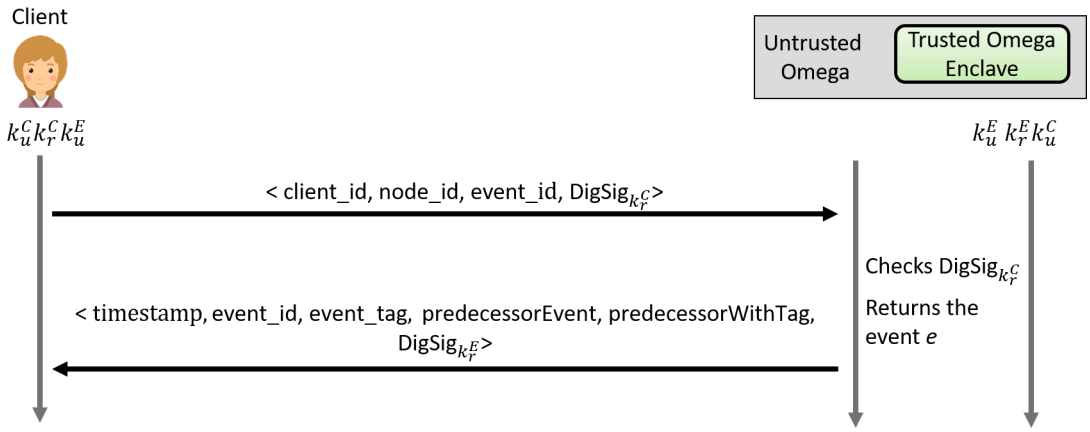

Figure 3.11: Communication pattern for the functions *predecessorEvent* and *predecessorWithTag*.

3.5 Omega Key-Value Store

As mentioned in Section 2.4, we chose a key-value store to evaluate our system. We now describe in more detail how we implemented this application on top of the Omega service. Omega is a service that can be use a module for other applications, meaning that any application can easily be designed to use our service.

OmegaKV is an extension to key-values stores that have been designed for the cloud. It makes it possible to maintain a cache of some key-value pairs in the untrusted space of a fog node while still ensuring that clients observe up-to-date values of the cached objects, in an order that respects causality. This is achieved by resorting to the services of Omega. OmegaKV also ensures that all updates performed by edge clients on the fog node, if they are propagated to the cloud, they are propagated to the cloud in an order that respects causality. As discussed in Section 3.3, 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 should note that the later mechanisms have not been implemented in the current OmegaKV prototype, given that we have developed OmegaKV mainly to illustrate the use of and to asses the performance of Omega; improving the OmegaKV prototype to make it fully functional is outside the scope of this thesis).

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 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 3.12 illustrates the architecture of OmegaKV. As with the Omega service, the implementation of OmegaKV has components that run on a client library and components that run of the fog node.

OmegaKV uses Omega as follows. Every update performed on the local replica is associated with

Algorithm 9 OmegaKV server implementation

```
1:  function PUT(k, v)
2:      #OUTSIDE ENCLAVE
3:      event_id ← hash(k ⊕ v)
    ▷ wait for the event creation in enclave/Omega
4:      while (tmpEventHolder.contains(event_id) = false)
5:      e ← tmpEventHolder.get(event_id)
6:
7:      BEGIN ATOMIC
8:      (old_v, old_e) ← local_kv.get (k)
9:      if old_e = omega.orderEvents (old_e, e) then
10:         local_kv.put (k, (v, e))
11:     END ATOMIC
12:
13:     tmpEventHolder.remove(event_id)
14:     #OUTSIDE ENCLAVE
15:
16: function GET(k)
17:     #OUTSIDE ENCLAVE
18:     return local_kv.get (k);
19:     #OUTSIDE ENCLAVE
20:
    ▷ every time a new event is created
21: function NEWEVENTNOTIFICATION(e)
22:     #OUTSIDE ENCLAVE
23:     tmpEventHolder.insert(omega.getId(e), e)
24:     #OUTSIDE ENCLAVE
```

Algorithm 10 OmegaKV client implementation

```
1:  function PUT(k, v)
2:      #OUTSIDE ENCLAVE
3:      asyncCall(omegaKV.put(k,v))
4:      event_id ← hash(k ⊕ v)
5:      omega.createEvent(event_id, k)
6:      #OUTSIDE ENCLAVE
7:
8:  function GET(k)
9:      #OUTSIDE ENCLAVE
10:     v ← asyncCall(omegaKV.get(k))
11:     event_id ← omega.getId(omega.lastEventWithTag(k))
    ▷ wait for the asynchronous call
12:     while (v = null)
13:     hash_val ← hash(k ⊕ v).
14:     if event_id = hash_val then
15:         return v
16:     else
17:         return error;
18:     #OUTSIDE ENCLAVE
```

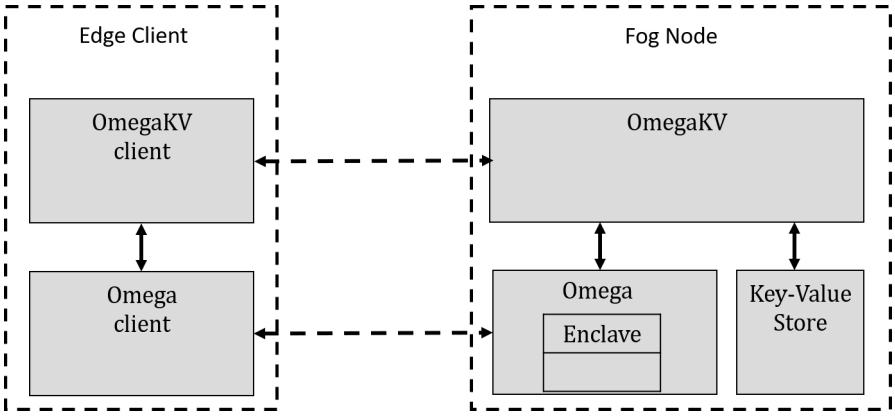

Figure 3.12: OmegaKV service components.

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 of the OmegaKV client and of the OmegaKV server are summarized in Algorithm 9 and 10 respectively. Note that all OmegaKV implementation is outside of the enclave this is possible thanks to the abstraction level that Omega offers allowing OmegaKV to abstract itself from using the enclave and only use the events that are generated.

To *put* a value on the OmegaKV, the client sends two messages in parallel: one to OmegaKV untrusted daemon and one to Omega daemon. The first is a normal write operation with the key and value. The second is the creation of an event in Omega where the event identifier is obtained hashing the concatenation of the key and the value. After receiving the request, OmegeKV server waits for the corresponding event to be created in Omega and then applies the update. After the event is generated in Omega, OmegaKV can apply the update locally as long as it still is the most recent update. Since all updates depend on the generation of events in Omega, update operations are serialized respecting causality.

To perform the *get* operation, the client also sends two messages in parallel: one to OmegaKV and one to Omega. The first is a normal request to read the value associated with the key. The second reads the latest Omega event associated with the target key. The client then uses the hash of the value concatenate with the key that has been safely stored in the event by Omega and compares it with the hash of the value returned by the OmegaKV server running on the untrusted zone of 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. If another write update has occurred between the two parallel messages, the client can just ask OmegaKV for the data correspondent to the event it has receive from Omega.

The necessity for two messages in parallel derives from the fact that the client cannot use the Omega

API to send very large data and therefore has to communicate directly with OmegaKV to send the updates values. In addition to this, it is necessary that the client is the one generating and sending the nonce to Omega to guarantee the freshness in the request's response, the nonce parameter is inside the event object which only makes sense at Omega abstraction level, so it is required to send two messages in parallel for the put and get operations.

To make OmegaKV informed of all events that are created in Omega we have added a callback to OmegaKV that is not part of the API. This callback is just an optimization, alternatively, a client could create an event and then send the event along with the update to OmegaKV having the same effect but with higher latency.

Finally, if the fog node were to ship 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). This operation can be executed by the cloud that can function as a client of the OmegaKV.

3.6 Discussion

Table 3.2: Techniques used by surveyed secure systems. Optimize attestation means that attestation is performed by a non-client entity, and optimize disk access means that the untrusted zone is responsible for transferring data to the disk.

Systems	Quorums	TEE	Integrity Technique	Optimize Attestation	Optimize Disk Access	App Abstraction Layer
MinByz	yes	—	—	—	yes	yes
Harpocrates	—	yes	—	—	yes	—
ShieldStore	—	yes	Flat Merkle tree	—	—	—
Speicher	—	yes	Flat Merkle tree	—	—	—
Pesos	—	yes	Seagate Kinetic Disk	yes	—	yes
Excalibur	—	yes	—	yes	yes	—
Omega	—	*yes*	*Normal Merkle tree*	*yes*	*yes*	*yes*

Table 3.3: Systems that offer causal consistency. K is the number of all data objects in the system.

Systems	Application Abstraction Layer	Automatically define Causal Dependencies	Efficient for the Client	Edge Support	Metadata
Kronos	yes	—	yes	—	Graph O(K)
Magpie	—	yes	—	—	Scalar O(1)
Saturn	—	yes	yes	—	Scalar O(1)
COPS	—	yes	—	—	Graph O(K)
Gesto	—	yes	yes	yes	two Scalars O(1)
Omega	*yes*	*yes*	*yes*	*yes*	*Scalar O(1)*

In this section, we compare and discuss how the concepts and ideas presented in the related work have inspired us in the design of Omega, a secure middleware service for edge applications. We support

our discussion with the information presented in Table 3.2 and Table 3.3, that summarize the most important system characteristics of the surveyed solutions.

As stated earlier, the execution of applications in edge servers increases the risk of these applications being attacked and failing. Several of these risks derive from the fact that fog nodes, that offer fog layer resources, are deployed in more vulnerable locations, which increases the probability of being compromised, and potentially becoming malicious. Thus, it is important to create solutions that offer security guarantees to applications that run on the fog. In Table 3.2 we can find several systems with technology to help with this challenge.

Some systems, such as MinByz [29], rely on replication and on the use of quorums to achieve dependability. This approach requires the client to make multiple connections in order to tolerate Byzantine faults. 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. Thus we have considered the use a TEE, specifically of Intel SGX encalves, as a better alternative. However, enclaves possess small memory size and therefore we needed to design a solution that could circumvent this limitation. We have opted to store data outside the enclave, although this requires the implementation of additional mechanisms to ensure data integrity.

The ShieldStore [34] system ensures data integrity outside the enclave by maintaining a Merkle tree. Similarly Speicher [35] stores data on disk and ensures its integrity with a Merkle tree. However, both systems only implement a flat Merkel tree, a tree with a single level. This implies that they do not take advantage of the logarithmic growth property it offers. Moreover, in both cases, the leaves of the Merkel trees have considerable size, which is not efficient as shown in Section 4.2.1. Pesos [36] takes advantage of special disks that offer security guarantees. The use of these disks imposes an additional requirement for specific hardware. It also implies a large latency in all operations. These considerations suggested that the Merkle tree is a good approach to store data in memory outside of the enclave. However, we need to expand the depth of the Merkle tree to take advantage of its properties. For this reason we chose to use a normal Merkle in our Omega system. Note that although Speicher also stores data in memory, it stores in the enclave a hash for each entry of the data storage, which is not scalable given the enclave's limited memory.

Any solution based on Intel SGX requires some form of enclave attestation operation. From the systems covered in the related work, only Pesos does not require the client to perform the attestation. In Pesos an external service attests the system and transfers cryptographic keys on the network into the enclave. Additionally Excalibur [48] proposes a different attestation technique, however the client has to attest the monitor at least once. This attestation operation has a high latency, so it is important that the clients can avoid it. In Omega, we have proposed an efficient technique for the client that, unlike Pesos, avoids sending cryptographic keys on the network and does not require the client to execute attestation

(see Section 3.4.5).

Another important factor to offer low latency is to avoid disk usage as much as possible. Harpocrates [33] offers a solution where the enclave does not need to access the disk, however, it only verifies cookies, hence it requires a small state in the enclave. In Excalibur, the data is sent by the clients to decrypt and the system is not responsible for storing the data. Therefore, unlike ShieldStore, Speicher, and Pesos, our system Omega does not require the enclave to accesses the disk, because all the events are copied to disk by the untrusted part.

Additionally, Omega can offer its services with high abstraction for applications, i.e. applications do not need to be aware that there is a TEE within Omega. Applications just need to use the API (see Section 3.3.1) to create and read events with security guarantees, such as integrity, freshness and data consistency. Data confidentiality is application-level responsibility, applications can choose the best way to obtain confidentiality. As for the confidentiality of the events themselves, as it is not part of our goals we leave it to future work, where the enclave may store a symmetric key to encrypt all events before storing them in untrusted memory.

Finally, we wanted to provide the developer a general and abstract layer to make application development easier. We aimed at a general solution, that could be used to implement not only a key-value store but also other services at the fog layer. In this front, we found inspiration in the Kronos system [17]. Kronos offers an easy-to-understand and easy-to-use abstraction for edge applications, using events allows many distributed applications to order operations they need and can later extract information about them. This work has inspired us to develop a secure event ordering middleware for the edge. However, we have decided to offer an alternative interface to the application. Kronos allows clients to define arbitrary dependencies among events and uses a graph to keep track of these relations. However, a graph can take considerable space in memory, has a high cost to manage (requires a garbage collector), and requires clients to explicitly declare the dependencies among events. Other systems such as Saturn [19], COPS [12], and Gesto [64], define the operations dependencies automatically. COPS needs to store a graph on the client side, which is not practical, as many clients may have memory restrictions. Saturn and Gesto use only one scalar to order the operations they execute, which is very efficient for both the client and the amount of information required. Thus, we decided to use positive integer within events to order them, thus whenever a fog node generates an event it assigns a timestamp to it. Magpie [59] offers a solution that generates application events transparently, however, this transparency prevents the application from creating events with a high level of abstraction. This means that clients must manipulate events with a white box abstraction, and work with machine-specific parameters such as thread id, Omega events are generated as black box giving a high level of abstraction service to clients. Concluding, using events with only one scalar allows generating small size events facilitating its use at the edge of the network.

Summary

This chapter addresses the design and implementation details of Omega, a secure event ordering service for the edge. Our Omega system generates events with the guarantee of freshness, integrity and causal consistency by leveraging Intel SGX. For this it requires to store data outside the enclave, guaranteeing its integrity with a Merkle tree also stored in untrusted memory. Because Omega is an ordering service, edge applications can use the Omega service through its simple API having an abstraction over the Intel SGX technical details. In the next chapter, we evaluate our Omega system, and a use case called OmegaKV.

4

Evaluation

Contents

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

4.1 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. Both the clients and the fog node are deployed in our laboratory, in the same network, emulating a 5G station communicating with a terminal (i.e., a 1-hop communication). Cloud services are executed on Amazon Elastic Compute Cloud (Amazon EC2), using the London data center, in t2.micro virtual machines.

The Intel SGX SDK and the code for the enclave are in C/C++. For simplifying the application implementation on Omega, we created a layer in Java over the part of Omega that operates in the enclave. For this we used Java 11 and the Java Native Interface (JNI) was used as a bridge between Java and C++. For persistent storage we used the key-value store Redis [69]. Redis is a typical key-value storage system, providing value storage indexed by keys. Redis is an in-memory system but offers a snapshot mechanism for data persistence. As a result of Omega abstraction, any key-value system could be used, we chose Redis because it is simple and lightweight to use. In the experiments we executed 5000 operations and discarded the first 500 and the last 500 for warming up and to remove potential outliers

4.2 Omega Configuration and Performance

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

4.2.1 Merkle Tree Configuration

The Merkle tree used to store events is used on most of the Omega operations. Therefore, its correct configuration is key to the performance of the service. To understand how to configure the Merkle tree it

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

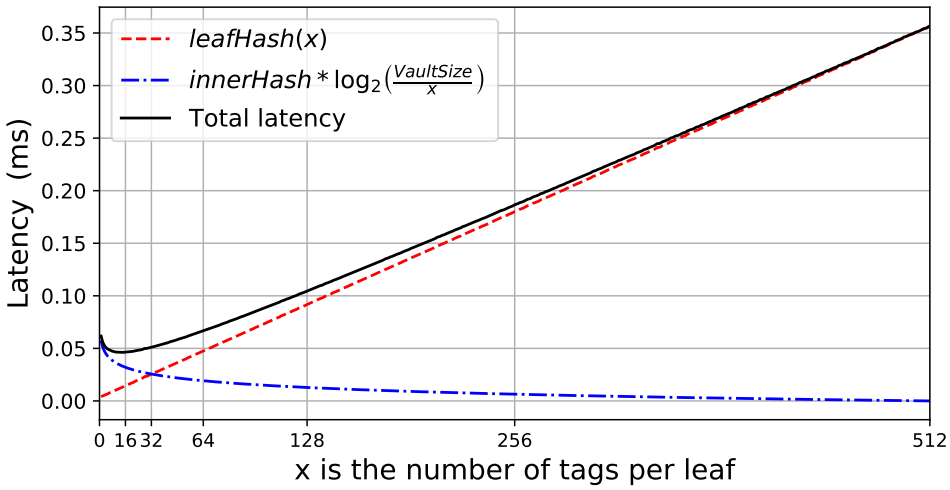

Figure 4.1: Estimated optimal leaf size of the Merkle tree for a vault of size 512.

The formula above suggest that the optimal size of the leaf nodes of the Merkle should be very close to 1, given that the cost of hashing the leaf node grows linearly, while the cost of hashing the inner nodes grows logarithmically. However, it should be noted that there is a different cost of calculating a very large hash (*leafHash*) versus calculating many small hashes (*innerHash*). Figure 4.1 depicts the estimated cost of vault operation, on a vault of size 512 when the size of leaf nodes is varied from 1 to 512 entries. Note that when the leaf size is 1, the height of the Merkle tree is 9 and when the size of the leaf is 512 the entire vault is stored in a single leaf. The values in this figure were obtaining using the formula above, that was fed with results obtained experimentally for the parameters $leafHash(x)$ and *innerHash*. The values suggest that leafs should not be large; in this case, for a vault size of 512, the formula suggests that 8 is the best leaf size.

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

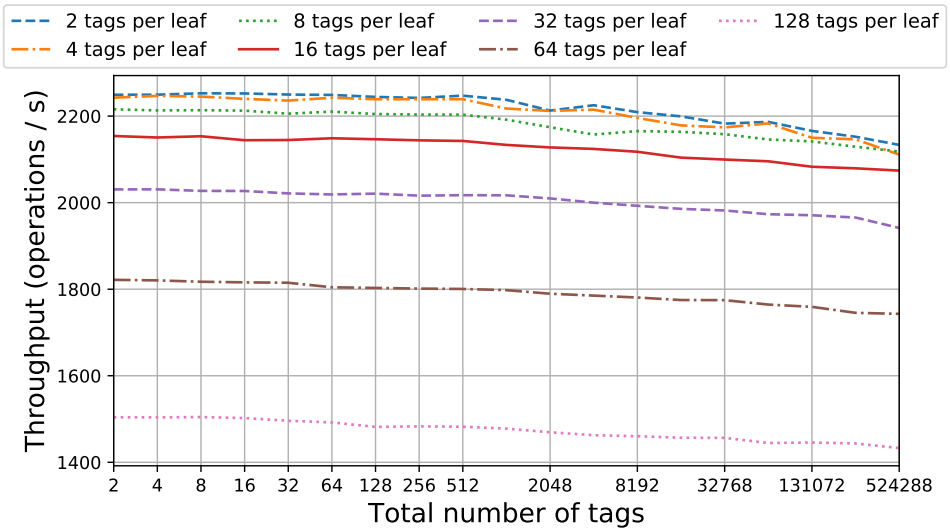

Figure 4.2: Actual performance of the Merkle tree as a function of the vault size and leaf size.

measured the performance of the Omega vault implementation with different leaf sizes and different vault sizes. The results are depicted in Figure 4.2. 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.

4.2.2 Executing Omega Operations

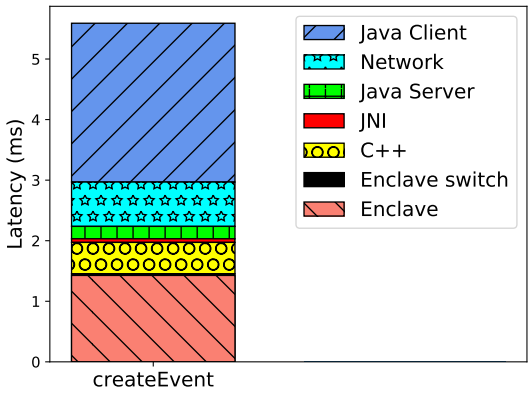

Figure 4.3: Operation latency for *createEvent*.

We now present the results of a set of experiments that are aimed at assessing the performance of Omega when used in isolation. For this experiment we have measured the latency observed by a client when performing the most costing requests of the Omega interface. The *createEvent* operations

involves modifying the Omega vault, which has a major latency penalty. We have measured how each software component that is executed in the client critical path contributes to the latency. The results are depicted in Figure 4.3.

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 3.4.7). Thus, the main contributor to the latency are the cryptographic functions executed in the client and in the enclave. In the client, 2ms–2.5 ms are required to compute and verify digital signatures. On the server side, most of the time is also spent in the process of computing and verifying digital signatures.

4.3 Performance of the OmegaKV

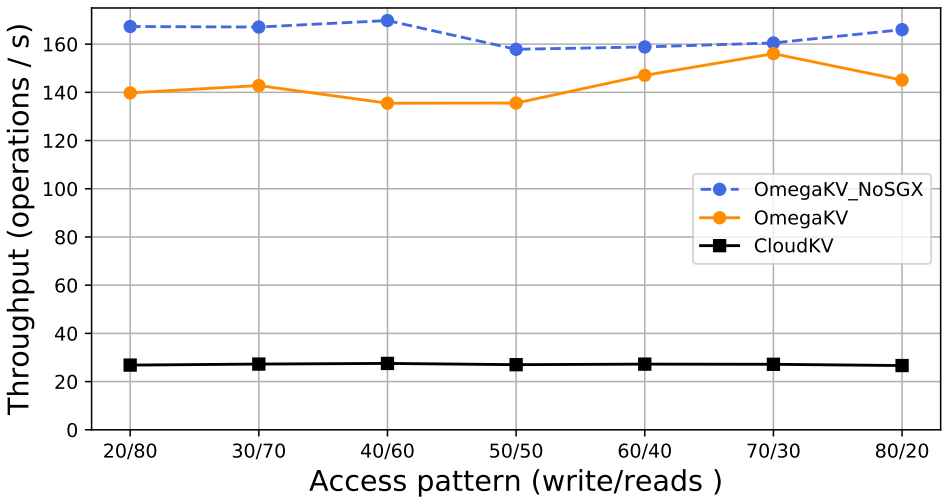

Figure 4.4: Access pattern throughput (writes/reads).

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 [69] 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 date integrity, and they do not use

JNI for interaction between Java layer and C++.

Figure 4.4 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 4.2), 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. There is an insignificant variation of throughput relative to the variation of the access pattern. The justification for this small variation is that the major overhead in all operations is a result of the cryptographic operations that are present in both reads and writes, as seen in Section 4.2.2.

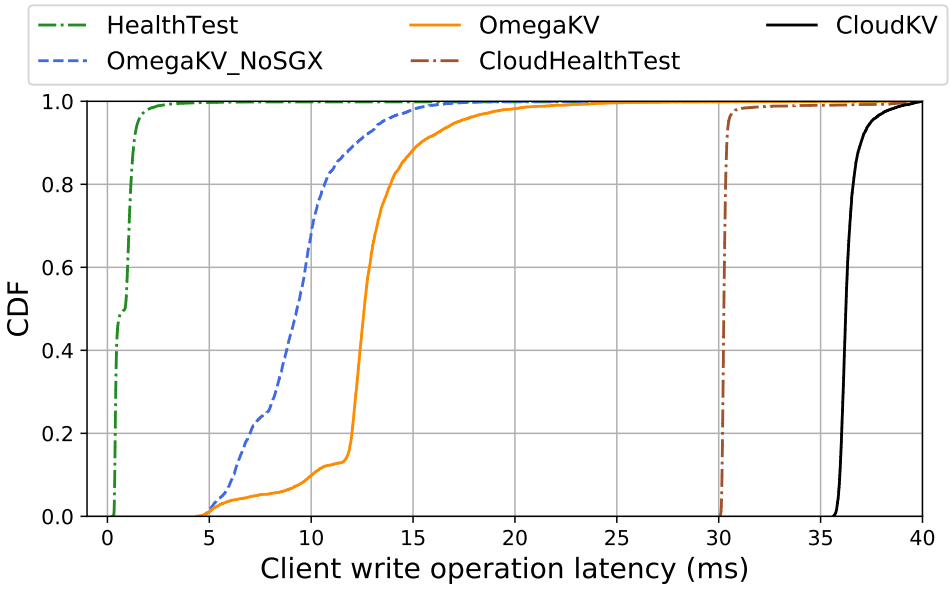

Figure 4.5: Write operation latency of a fog node and cloud.

Figure 4.5 shows the Cumulative Distribution Function (CDF), it compares the latency that a client experiences when using the services OmegaKV, OmegaKV_NoSGX, and CloudKV. For a better understanding of the graph, we use the ping operation to measure the round-trip time from the client to the fog node and to the cloud. This is shown as the HealthTest line for the fog node and the CloudHealthTest for the cloud. As expected the client can perform operations with much lower latency by using the fog node rather than using the CloudKV services that are in a data center, a reduction from 36ms to 12ms, close to 67%. OmegaKV has higher latency than OmegaKV_NoSGX, due to the use of the enclave.

In absolute value we observe an increase in latency in the order of 4ms, which is non-negligible but still significantly smaller than the latency introduced by wide-area links. This allows OmegaKV to offer latency values in the 5ms–30ms range required by time-sensitive edge applications [4].

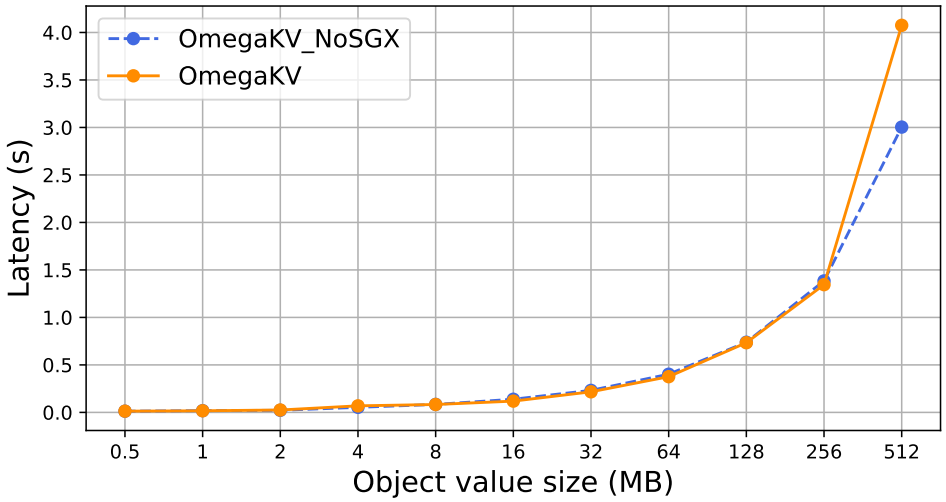

Figure 4.6: Write operation latencies w/ and w/o SGX.

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 4.6. 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.

Summary

This chapter presented the experimental evaluation of Omega, detailing the tests that were conducted in order to assess its performance. We study the best configuration for our Merkle tree and the cost of using the enclave on our system. We finish by evaluating an application using Omega, a key-value store. The results demonstrate that despite the use of enclave and the Merkle tree our system allows latency values in the 5ms-30ms range, as required by time-sensitive edge applications.

5

Conclusion

Contents

5.1 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 thesis makes a step in this direction by describing the design and implementation of a middleware service that can be executed on fog nodes in a secure manner leveraging on the properties of trusted executions environments such as Intel SGX. In particular, we have proposed Omega, an event ordering service that can be used as a building block to build higher level abstractions. With the dual purpose of illustrating the use of Omega and of assessing its performance when used in practice, we have also designed and implemented OmegaKV, a causally consistent key-value store for the edge. Our evaluation shows that, despite the costs incurred with the use of the enclave mode to secure the Omega implementation, the use of Omega based applications can still provide much smaller latency and higher throughput than current cloud based solutions.

5.2 Future Work

For future work, we plan to explore approaches that allow increasing the performance of our system. For this it would be appealing to develop a cache inside the enclave, to take advantage of all the memory it offers. Another approach is to develop a multithreaded version of Omega. However, a multithread version implies synchronizing data structures like Merkle tree or the timestamp used to order the events.

A major goal for future work is to deploy a version of Omega that takes advantage of multiple fog nodes enabling Omega to efficiently scale at the fog layer and produce experimental results closer to the real-world scenario.

Bibliography

[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] G. Idex, "Cisco global cloud index: Forecast and methodology, 2016–2021," *White Paper*, 2016.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] M. Satyanarayanan, V. Bahl, R. Caceres, and N. Davies, “The case for VM-based cloudlets in mobile computing,” *IEEE Pervasive Computing*, vol. 8, no. 4, 2009.

[26] 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.

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

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

[29] J.-P. Martin, L. Alvisi, and M. Dahlin, “Minimal Byzantine storage,” in *Proceedings of the International Conference on Distributed Computing*, Berlin, Heidelberg, Jun. 2002.

[30] 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, pp. 335–344.

[31] J.-E. Ekberg, K. Kostiainen, and N. Asokan, “The untapped potential of trusted execution environments on mobile devices,” *IEEE Security & Privacy*, vol. 12, no. 4, 2014.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, Jul. 2017.

[45] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, San Jose, California, Jul. 2015.

[46] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX enclaves from practical side-channel attacks," in *Proceedings of the USENIX Annual Technical Conference*, Boston, MA, USA, Jul. 2018.

[47] S. K. E. Storage, https://www.seagate.com/support/enterprise-servers-storage/nearline-storage/kinetic-hdd/, accessed: 2019-10-04.

[48] 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.

[49] T. Group *et al.*, "TPM main specification level 2 version 1.2," 2006.

[50] 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.

[51] 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.

[52] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, 1991.

[53] 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.

[54] 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.

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

[56] M. Reiter and L. Gong, "Securing causal relationships in distributed systems," *Computer Journal*, vol. 38, no. 8, 1995.

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

[58] 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.

[59] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, “Using Magpie for request extraction and workload modelling.” in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, Dec. 2004.

[60] E. Ahmed and M. H. Rehmani, “Mobile edge computing: Opportunities, solutions, and challenges,” *Pervasive Computing*, vol. 70, 2017.

[61] 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.

[62] 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.

[63] E. A. Brewer, “Towards robust distributed systems (abstract),” in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Portland, Oregon, USA, Jul. 2000.

[64] N. Afonso, “Mechanisms for providing causal consistency on edge computing,” Master’s thesis, Instituto Superior Técnico, Universidade de Lisboa, Nov. 2018.

[65] 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.

[66] 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 *Proceedings of the ACM Conference on Computer and Communications Security*, Toronto, Canada, Oct. 2018.

[67] 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.

[68] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, 2010.

[69] Redis, “Key-value store,” http://redis.io, accessed: 2019-10-04.

[70] 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.

[71] 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.

[72] ANSI, “X9.62-1998 public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ECDSA),” Sep. 1998.

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

[74] NIST, “FIPS 180-4, Secure Hash Standard,” Aug. 2015.

[75] 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.