TRANSPARENT APPLICATION DEPLOYMENT IN A SECURE, ACCELERATED AND COGNITIVE CLOUD CONTINUUM

Grant Agreement no. 101017168

Deliverable D3.4
Final release of SERRANO Secure Infrastructure Layer

Programme:	H2020-ICT-2020-2
Project number:	101017168
Project acronym:	SERRANO
Start/End date:	01/01/2021 – 31/12/2023

Deliverable type:	Report
Related WP:	WP3
Responsible Editor:	NBFC
Due date:	30/06/2023
Actual submission date:	14/07/2023

Dissemination level:	Public
Revision:	FINAL

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017168

Revision History

Date	Editor	Status	Version	Changes
31.05.2023	MLNX	Draft	0.1	Initial ToC and content
20.06.2023	CC	Draft	0.2	Section 3
21.06.2023	AUTH	Draft	0.3	Inputs in different sections
26.06.2023	NBFC	Draft	0.4	Section 4
28.06.2023	ICCS	Draft	0.5	Review
11.07.2023	MLNX	Final	1.0	Final integration, review, formatting and editing.

Author List

Organization	Author
MLNX	Yoray Zack, Juan Jose Vegas Olmos, Yonatan Maman
CC	Márton Sipos, Marcell Fehér, Daniel E Lucani Rötter
AUTH	Argyris Kokkinis, Dimitris Danopoulos, Dimosthenis Masouros, Aggelos Ferikoglou, Kostas Siozios
NBFC	Anastasios Nanos, Alexandros Karantzoulis, Charalampos Mainas, Christos Panagiotou, George Ntoutsos, Matias Vara Larsen, Ilias Apalodimas

Internal Reviewers

Aristotelis Kretsis, Panagiotis Kokkkinos (ICCS)

Abstract: This deliverable (D3.4) presents the outcomes of *Work Package 3 "**Hardware and Software Platforms for Enhanced Security**"* of the SERRANO project, which comprises the work by 5 partners and an investment of 65 PMs. The WP has ran for 23 months and comprises 3 concurrent tasks. Each task has developed different aspects of the SERRANO platform: the storage system in itself, a scalable secure storage system and workload isolation of processes, respectively. This deliverable owes to be read in conjunction with the previous three deliverables within this WP: D3.1 Accelerated encrypted storage architecture, D3.2 Secure cloud storage system and D3.3 Trust and isolated execution on untrusted physical tenders. This deliverable provides an overview of the final release of the SERRANO secure infrastructure layer, which is then used in the final demonstrators of the project.

Keywords: SERRANO platform, secure storage, distributed storage, secure boot, trusted execution, confidential computing

Disclaimer: *The information, documentation and figures available in this deliverable are written by the SERRANO Consortium partners under EC co-financing (project H2020-ICT-101017168) and do not necessarily reflect the view of the European Commission. The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The reader uses the information at his/her sole risk and liability.*

Copyright © 2021 the SERRANO Consortium. All rights reserved. This document may not be copied, reproduced or modified in whole or in part for any purpose without written permission from the SERRANO Consortium. In addition to such written permission to copy, reproduce or modify this document in whole or part, an acknowledgement of the authors of the document and all applicable portions of the copyright notice must be clearly referenced.

Table of Contents

1 Executive Summary 10
2 Introduction 11
2.1 Purpose of this document 11
2.2 Document structure 12
2.3 Audience 13
3 SERRANO-enhanced Storage Service 14
3.1 Overview 14
3.2 Architecture and Implementation 14
3.2.1 Architecture 14
3.2.2 The On-premises storage gateway 16
3.2.3 SERRANO edge devices 17
3.2.4 The Skyflok.com backend 17
3.2.5 Cloud Storage locations 18
3.2.6 Cloud Benchmarker Service 21
3.3 Public APIs 22
3.3.1 Secure Storage API 22
3.3.2 Storage Policy API 24
3.3.3 Cloud and Edge Telemetry API 25
3.4 Developer Portal 26
3.4.1 Managing S3 buckets 27
3.4.2 Managing storage policies 28
3.4.3 Managing API keys 29
3.5 Performance Considerations 30
3.5.1 Reducing the number of edge-cloud HTTP calls 30
3.5.2 File caching 31
3.6 Integration with SERRANO Platform Services 32
3.6.1 Automatic storage policy creation 32
3.6.2 Acceleration of data processing 33
3.6.3 Offloading of encryption for TLS connections 33
3.7 Privacy and Security 33
3.7.1 S3 authentication 34
3.7.2 Pre-signed URLs 35
3.7.3 Random Linear Network Coding 36
4 Trust and Isolation on Untrusted Physical Tenders 38
4.1 Hardware Trust 38
4.1.1 Secure Boot 39
4.1.2 Measure Boot 40
4.1.3 Trusted Platform Module as the Silicon Root of Trust 41
4.2 Confidential Computing and Trusted Execution 42
4.2.1 Workload attestation 43

4.2.2 Incorporating Sigstore in SERRANO 45
4.3 Isolation Between Tenants in Edge Computing 49
4.4 Integration of Security Mechanisms........ 54
5 Conclusions........ 56

List of Figures

Figure 1: The SERRANO platform, utilizing edge, cloud and HPC resources and empowering the Everything as a Service (EaaS) notion towards the cloud continuum 11

Figure 2: The core components of the SERRANO-enhanced Storage Service. 15

Figure 3: The components of the cloud monitoring features of the SERRANO-enhanced Storage Service. 15

Figure 4: A random selection of Cloud Storage Location performance characteristics, averaged over 7 days. Each cloud location was measured once a day. 21

Figure 5: Example storage policy definition that uses compression, encryption and erasure coding. Erasure coded fragments are distributed to both cloud and edge locations with a redundancy factor of 100%. 24

Figure 6: Developer portal - listing S3 buckets. 27

Figure 7: Developer portal- listing the contents of an S3 buckets. 27

Figure 8: Developer portal - listing Storage policies. 28

Figure 9: Developer portal - creating a new storage policy. 28

Figure 10: Developer portal - listing API keys. 29

Figure 11: Developer portal - creating a new API key. 30

Figure 12: Preliminary measurements for file upload and download of a single 10MB file. .. 31

Figure 13: Using pre-signed URLs to download a file 35

Figure 14: RLNC encoding of a 1000-byte file using 4 symbols that are combined into 6 erasure-coded packets 36

Figure 15: Secure boot execution flow 40

Figure 16: Measured boot execution flow 40

Figure 17: Image and signature creation 46

Figure 18: Signature Verification Process 49

Figure 19: A unikernel running as a VM on HEDGE 52

Abbreviations

ABI	Application Binary Interface
ACL	Access Control List
AES	Advanced Encryption Standard
AI	Artificial Intelligence
API	Application Programming Interface
ARDIA	A Resource reference model for Data-Intensive Applications
ASGI	Asynchronous Server Gateway Interface
CEO	Chief Executive Officer
CORS	Cross-Origin Resource Sharing
CoT	Chain of Trust
CT	Certificate Transparency
CTO	Chief Technology Officer
DM	Device Mapper
DPU	Data Processing Unit
FaaS	Function as a Service
FPGA	Field Programmable Gate Array
GCM	Galois/Counter ModE
GCP	Google Cloud Platform
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
HW	Hardware
IMA	Integrity Measurement Architecture
IO	Input/Output
JSON	JavaScript Object Notation
K8s	Kubernetes
KVM	Kernel-based Virtual Machine
LSU	Least Recently Used
ML	Machine Learning
NIC	Network Interface Card

OCI	Open Container Initiative
OIDC	OpenID Connect
OS	Operating System
PCR	Platform Configuration Register
PEF	Protected Execution Facility
PXE	Preboot Execution Environment
QoS	Quality-of-Service
REST	Representational State Transfer
RLNC	Random Linear Network Coding
ROM	Read-Only Memory
RoT	Root of Trust
RPC	Remote Procedure Call
SDK	Software Development Kit
SEV	Secure Encrypted Virtualization
SFTP	Secure File Transfer Protocol
SLA	Service Level Agreement
SoC	System on a Chip
SW	Software
TDX	Trust Domain Extensions
TEE	Trust Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module
UC	Use Case
UEFI	Unified Extended Firmware Interface
URL	Uniform Resource Locator
VM	Virtual Machine
VMM	virtual Machine Monitor
WP	Work Package
WSGI	Web Server Gateway Interface

1 Executive Summary

This deliverable entitled "D3.4 Final release of SERRANO secure infrastructure layer" presents the final results of Work Package 3 in SERRANO, entitled "Hardware and software platforms for enhanced security". The primary objective of this WP was to develop innovative mechanisms to enhance the security in disaggregated cloud and edge infrastructures. Specifically, this WP aimed to: (i) develop tools for acceleration of encrypted storage, (ii) develop a scalable secure storage system, enabling massive storage deletion by crypto-key handling, (iii) enable edge nodes to support heterogeneous data streams from diverse digital services, and (iv) provide strong isolation and security guarantees for workloads executing at edge nodes.

This document summarizes the efforts and working system solution developed within this work package, which includes a solution for distributed storage that employs acceleration engines in distributed infrastructures, as well as orchestration tools that rely on trusted execution environments and hardware security mechanisms.

The work presented in this deliverable, in conjunction with previous deliverables from this work package, will be transferred to the final test bed demonstrators for further benchmarking.

2 Introduction

SERRANO targets the efficient and transparent integration of heterogeneous resources, providing an infrastructure that goes beyond the scope of the "typical" cloud and realizes a true computing continuum. The project aims to define an intent-based paradigm of operating federated infrastructures consisting of edge, cloud, and HPC resources, which will be realized through the SERRANO platform (Figure 1). SERRANO automates the application deployment process across various computing technologies, translating applications' high-level requirements to infrastructure-aware configuration parameters. Next, the SERRANO platform will determine the most appropriate resources of the cloud continuum to be used by an application and then transparently deploy workloads and coordinate data movement. Also, SERRANO will continuously adapt the deployed applications, based on the observe, decide, act approach.

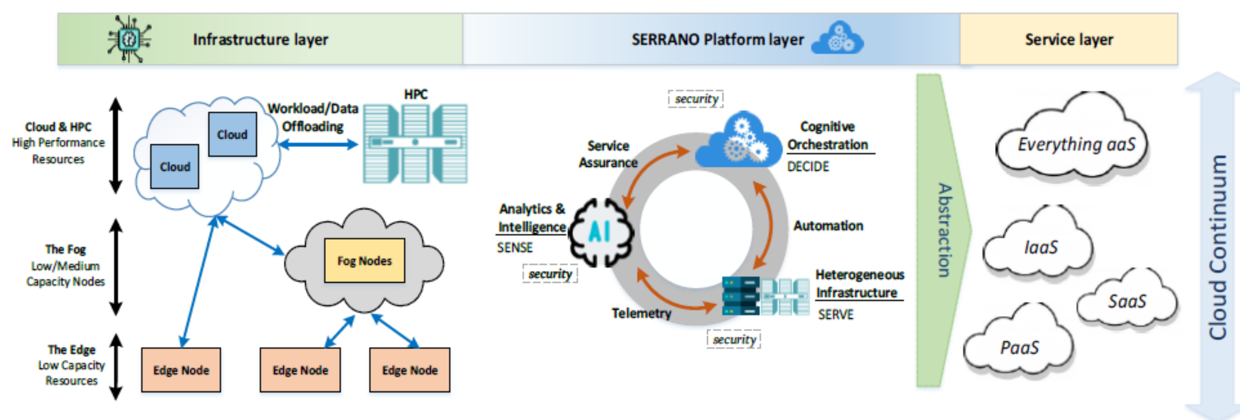


Figure 1: The SERRANO platform, utilizing edge, cloud and HPC resources and empowering the Everything as a Service (EaaS) notion towards the cloud continuum

2.1 Purpose of this document

The present deliverable (D3.4) presents the outcomes of WP3 and its tasks *"**Hardware and Software Platforms for Enhanced Security**"* of the SERRANO project, during the first iteration of the incremental implementation plan. WP3 is associated with designing and implementing the secure boot and trusted execution mechanisms enabling workload isolation in the SERRANO platform and the development of a secure storage system for an edge-cloud continuum.

The deliverable provides an overview of the enhanced secure storage service developed within the SERRANO project. The service encompasses geographically diverse storage facilities accessed from a user interface. In addition, the system capitalizes on the availability of network interface cards (NICs) containing the SERRANO-developed encryption accelerators. Such accelerators improve the user experience by reducing end-to-end latency and freeing up local processing resources, which allows the network and storage fabric to scale up graciously.

Moreover, the implemented platform, along with the required backend mechanisms, provides the appropriate public secure and storage APIs and a developer portal. The main building blocks are described in this document as well as different performance tests. The results are fairly promising, as many optimizations, which are also described, are introduced, such as reducing the number of edge-cloud HTTP calls (edge-cloud continuum) and file caching. The deliverable also provides visibility on S3 authentication, pre-signer URLs policies, and random linear network coding (RLNC) techniques.

D3.4 also provides the final SERRANO developments on the trust and isolation execution on untrusted physical tenders. The current trend is to seek technologies that enable the isolation of processes, not only in the form of SW but also at the HW layer. This is particularly relevant with confidential AI approaches, where tens of thousands of AI processes (small and large) are distributed throughout the fabric for processing.

2.2 Document structure

This deliverable builds upon the previous three deliverables generated within WP3 in SERRANO (D3.1, D3.2, and D3.3), each defining, describing, and providing benchmark results on different building blocks. It has been an active and federated decision by the partners involved in these developments to avoid duplicating much content from these previous deliverables into the current one and limit ourselves to repeating content necessary to understand new developments. Hence, this document owes to be read and considered in conjunction with previous deliverables.

The present deliverable is split into the following major chapters:

- Section 3 provides an extensive overview of the SERRANO-enhanced storage service developed within the project. This section includes performance measurements on the storage solution under different conditions.
- Section 4 provides an extensive description of the SERRANO-enhanced mechanisms to provide trust and isolation execution on untrusted physical tenders.

Following the mid-term review outcome guidelines, we have reduced repeating documentation from previous deliverables. This means an overview of the accelerated encryption storage system and SERRANO platform is available in D3.1, D3.2, and D3.2 from M15. Essentially, the platform has been developed over MLNX hardware (i.e., a DPU). The hardware system for testing purposes is located in MLNX in Israel and consists of 2 servers with 2 DPUs for which CC has remote access and has tested the SERRANO secure storage service. In addition, Section 4 provides an overview that is described in deep in previous deliverables. The SW platform by MLNX is also described in previous deliverables – such SW APIs have been essentially co-designed MLNX-CC, and then CC have used them for testing the results discussed in Section 3.

2.3 Audience

This document is publicly available and helpful to anyone interested in accelerated storage, network interface cards, orchestration of storage resources, privacy and security in storage systems, edge-cloud continuum from a storage perspective, and confidential computing and trusted execution environments.

3 SERRANO-enhanced Storage Service

This Section includes a selection of results originally reported in Deliverable 3.2 and 3.3. The material is taken from these deliverables and updated to reflect progress made since their initial submission.

3.1 Overview

The SERRANO-enhanced Storage Service (Storage service in short and also referred to in some deliverables as 'Secure Storage Service') is the SERRANO project's file storage solution that uses object storage semantics. It exposes an S3-compatible API and, where possible, follows industry standards.

It is a distributed storage system that differentiates itself by providing support for both a large number of cloud locations as well as user-deployed edge locations. This makes it possible to meet a wide range of user requirements, bringing the benefits of both cloud and edge storage as part of a single solution. Before distribution, files are erasure coded for optimal cost-effectiveness and high reliability. A novel erasure code called Random Linear Network Coding (RLNC) with additional security benefits is employed. Files are encrypted at the edge and compressed for better storage efficiency. A file caching solution improves performance, especially when data is stored on the cloud storage locations. The Storage service is simple to configure and highly customizable. Each S3 bucket has a storage policy associated with it, specifying how and where the files are stored.

The Storage service is a core enabler of the project's first use case, UC1: Secure Storage. Its main features have been derived from the requirements of both the project and the use case, as has been described in Work Package 2 deliverables.

3.2 Architecture and Implementation

3.2.1 Architecture

The SERRANO-enhanced Storage Service (Storage service) has three main components, each with a well-defined role. The On-premises storage gateway (Gateway) is deployed on edge, close to the applications that use the service. It performs all the data processing tasks: compression, encryption, and erasure coding. It also acts as a gateway to the storage locations and a mediator towards the Skyflok.com backend (Backend). It exposes all the APIs that are described in Section 3.3 to the users of the SERRANO platform and its services. A fourth component, the Cloud Benchmarker Service, has been added to provide cloud telemetry.

The Skyflok.com backend has been developed by Chocolate Cloud over the last few years. It is in charge with maintaining the metadata related to the stored files and provides many of the complementary services needed by a storage solution such as authentication and

authorization, management of storage locations and so on. The users of the service do not access it directly.

The SERRANO-enhanced devices provide the service with edge storage locations. These are easy to deploy resources that provide performance that is not achievable through cloud-based services, especially in terms of latency. Like the Gateway, they are deployed on the customer's premises.

An overview of the core components and their connection can be seen in Figure 2.

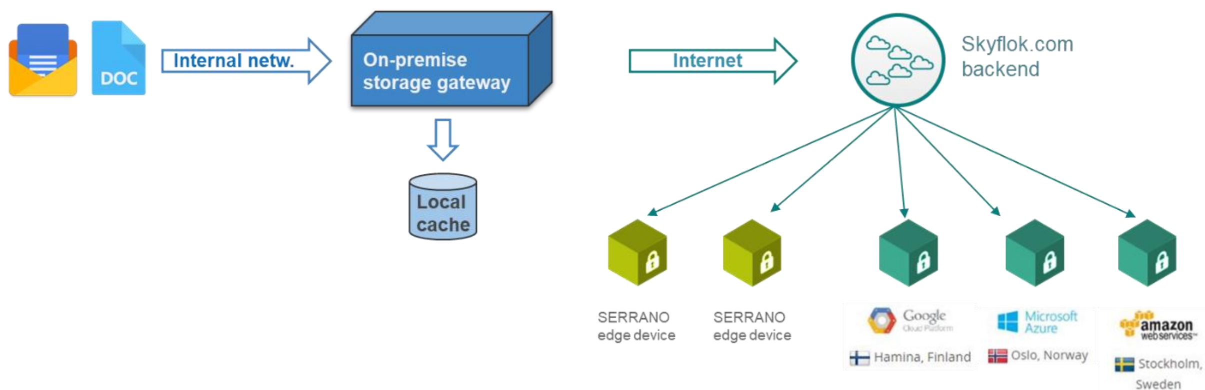


Figure 2: The core components of the SERRANO-enhanced Storage Service.

Monitoring cloud locations requires the ability to reliably schedule measurements that ascertain both the availability of each location and its performance characteristics regarding uploading and downloading files. The Skyflok.com backend provides the scheduling while the Cloud Benchmarker Service performs the measurements and stores the results. The Backend stores the results and are accessible to users of the Storage service through the Cloud Telemetry API. An overview of these separate responsibilities is shown below in Figure 3.

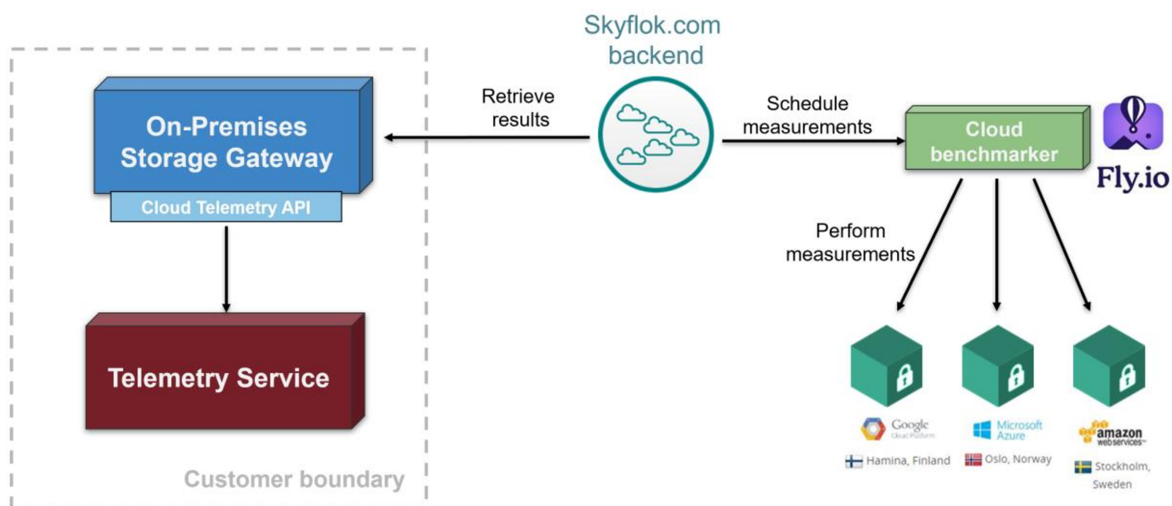


Figure 3: The components of the cloud monitoring features of the SERRANO-enhanced Storage Service.

3.2.2 The On-premises storage gateway

The On-premises storage gateway is the key new development of the SERRANO-enhanced Storage Service as well as its most important on-premises component. It is implemented in Python 3.8 using the FastAPI[1], a modern ASGI framework. Unlike traditional WSGI-based solutions like Flask, FastAPI has been developed with the goal of improving performance through asynchronous operations. The Gateway makes use of this by implementing all network and disk operations asynchronously, greatly improving the efficiency of the service and increasing the number of concurrent requests it can serve.

The Gateway runs as a containerized application deployed by the SERRANO orchestrator. Multiple instances can be deployed simultaneously since it features no state beyond caching some data for performance reasons. This makes it possible to tailor the performance of the SERRANO-enhanced Storage Service to the current workload by scaling horizontally. Its statelessness is a key design principle meant to ensure that the Gateway does not become a single point of failure or a performance bottleneck.

A key consideration related to performance is CPU use. Since all data processing operations are performed by the Gateway, acceleration techniques developed in Work Packages 3 and 4 are used to remove some of the burden from the CPU. These occur if specialised hardware (Nvidia DPU, GPU, FPGA) is available. The Gateway performs these tasks on the CPU if they are unavailable.

The second significant performance-related metric is memory usage. Given that the Gateway must be able to serve a large number of requests concerning large files, careful thought went into making this component memory-efficient. This design goal is achieved through a combination of techniques that make it possible to constrain the memory used when uploading and downloading files. First, larger files are divided into chunks called generations. Each generation is compressed, encrypted, erasure-coded, and distributed to the storage locations separately. The memory use of the component can be tuned by modifying the size of each generation. Second, file upload requests and file download responses are streamed. This is achieved using the TCP protocol's varying sized transmission window. This technique ensures that very little data beyond the generation being processed is kept in memory.

The Gateway features another performance-enhancing feature in the form of a local cache. Thus, files accessed recently or very popular are kept in their original, uncompressed, unencrypted, non-erasure-coded form on local, ephemeral storage. Like the other design decisions made when developing the service, the cache aims to improve performance beyond what a purely cloud-based solution can achieve.

[1] FastAPI framework: https://fastapi.tiangolo.com/

3.2.3 SERRANO edge devices

The SERRANO edge devices provide storage locations at the edge, on the customer's premises. Like the storage service itself, they provide an S3 interface. However, the client applications of the service do not access the devices directly. Instead, the Gateway is in charge of all file uploads and downloads to the storage locations. Details on how this is performed using pre-signed URLs is described in Section 3.7.2.

Each SERRANO edge device is a containerized application deployed by the SERRANO orchestrator. Each is a separate instance of MinIO[2], a high-performance, highly customizable object storage solution. It was selected because it is designed from the ground-up to be deployed in a container, it exposes the relevant S3 endpoints and includes a telemetry agent. The former is used to provide the platform's Telemetry Service with information regarding the status of the edge storage resource.

When deployed using Kubernetes (K8s), MinIO can use a wide range of available storage resources through K8s Persistent Volumes[3]. All information required to run MinIO, as well as all data that it stores, can be mounted using this technique. Thus, it is easy to tailor MinIO to the storage resources available on the customer's premises. For example, in most cases a local file system can be mounted to provide the necessary storage. However, through the use of plugins, a storage resource can be anything from an NFS share, an iSCSI drive/service, a Microsoft Azure disk, an Amazon Elastic Block Store, or even a local object storage solution such as Ceph.

Finally, while MinIO is a very competent multi-cloud solution with a wide range of features, we deploy it in its simplest, one instance configuration. Instead of relying on MinIO for security and reliability, we use the techniques offered by SkyFlok.

3.2.4 The Skyflok.com backend

SkyFlok is a next-generation file sharing and storage solution for users who care deeply about privacy and security. It is a multi-cloud platform that distributes data across a wide range of commercially available clouds. Beyond the big three of Amazon, Google, and Microsoft, SkyFlok supports most major EU cloud providers and can be configured to be GDPR compliant (42 out of a total of 59 cloud locations are GDPR-compliant). A key enabler of this is the ability provided to users to select the cloud providers that will store their data as well as the actual locations down to the city level. Internally, SkyFlok's secret sauce is RLNC, an erasure code that provides reliable service even if a cloud

[2]MinIO: https://min.io/

[3]Kubernetes Persistent Volumes: https://kubernetes.io/docs/concepts/storage/persistent-volumes/

provider becomes unavailable. It also offers protection from data loss and gives privacy benefits beyond those provided by conventional encryption.

SkyFlok was launched in February 2018. Since then, over 750 SME teams have used the service. Chocolate Cloud launched in 2020 its reseller portal for resellers in multiple countries (incl. Canada, Italy, Denmark, UK) to commercialise SkyFlok.

The SERRANO-enhanced storage service relies on the software infrastructure behind SkyFlok, the Skyflok.com backend, for a wide range of features. These can be grouped as follows:

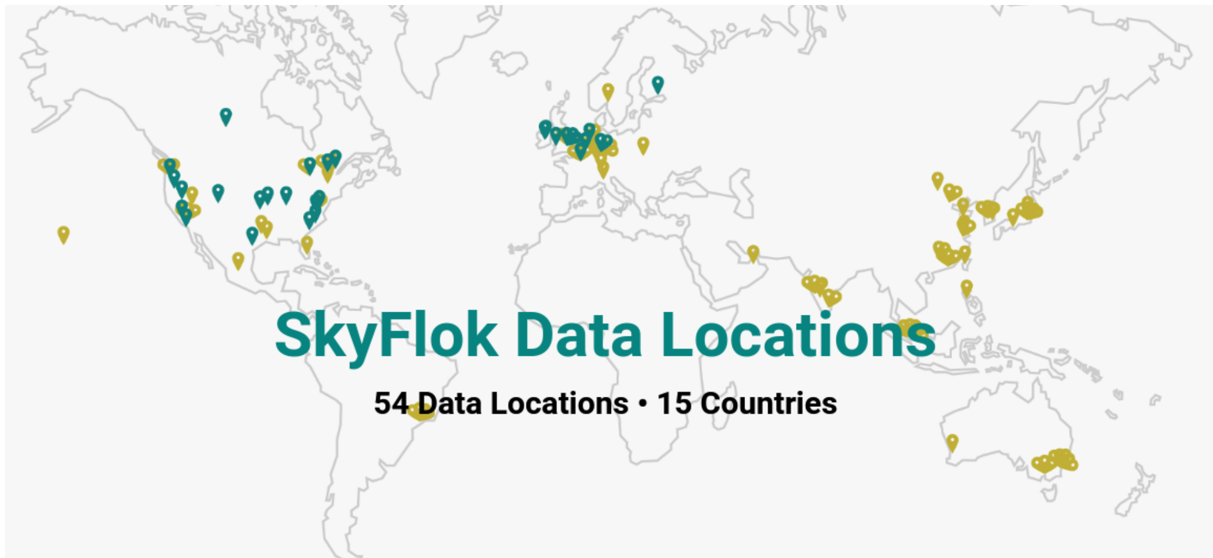
- File system management
- Storage location management
- Generating pre-signed upload and download links
- Storage policy management
- File and metadata consistency checking
- Authentication and authorization
- User and team management

The Skyflok.com backend uses a micro-service-based architecture. Each of the 17 services is in charge of a particular platform aspect. Most are also in charge of managing and persisting a set of business entities stored in either a traditional relational database or a NoSQL data store. Every service exposes a RESTful API and must perform authorization on the entities it manages itself. A separate service handles authentication. Almost all changes to business entities are recorded in an audit log and a consistency checking service periodically inspects that data and metadata are consistent with each other.

The Backend is written in Python and designed to be deployed to Google App Engine. This platform provides the scalability, security, and reliability necessary for the service to function. Business entities are persisted to either CloudSQL or Google Cloud Datastore, depending on their characteristics and access requirements.

3.2.5 Cloud Storage locations

The SERRANO-enhanced storage service provides access to all cloud locations supported by SkyFlok. This includes the locations offered by the three large US-based cloud operators: Amazon, Google, and Microsoft, as well as many smaller EU operators: CloudSigma, Deutsche Telekom, OVH, City Cloud, Exoscale, Scaleway, IONOS, Outscale, Leaseweb, Orange Business Cloud and Ventus Cloud. More are being added continuously in an effort to cover as much of Europe as possible. The 42 EU-based locations allow for fully-GDPR-compliant storage, exceeding the target value of 10 for **SIR.3** KPI, defined in Deliverable 6.6 (M27).



The following table shows the list of supported cloud locations as of June 2023.

Table 1: Supported cloud locations in SERRANO Secure Storage Service (June 2023)

Cloud provider	Location	GDPR-compliance
Google	Iowa	
Google	Oregon	
Google	Belgium	YES
Google	South Carolina	
Google	Northern Virginia	
Google	London	YES
Amazon	Ohio	
Amazon	Canada	YES
Amazon	Oregon	
Amazon	North Virginia	
Amazon	North California	
Amazon	Ireland	YES
Amazon	Frankfurt	YES
Amazon	London	YES
Microsoft	Virginia	
Microsoft	Iowa	
Microsoft	Chicago	
Microsoft	San Antonio	

Microsoft	Wyoming	
Microsoft	California	
Microsoft	Seattle	
Microsoft	Quebec City	YES
Microsoft	Toronto	YES
Microsoft	Dublin	YES
Microsoft	Amsterdam	YES
Microsoft	Cardiff	YES
Microsoft	London	YES
OVH	Beauharnois	YES
OVH	Gravelines	YES
OVH	Strasbourg	YES
Google	Frankfurt	YES
Amazon	Paris	YES
Google	Los Angeles	
Google	Hamina	YES
Google	Montreal	YES
Microsoft	Paris	YES
Amazon	Stockholm	YES
Microsoft	Oslo	YES
Microsoft	Frankfurt	YES
Microsoft	Zurich	YES
OVH	Frankfurt	YES
OVH	London	YES
OVH	Warsaw	YES
City Cloud	Frankfurt	YES
City Cloud	Karlskrona	YES
City Cloud	Buffalo, NY	
Exoscale	Frankfurt	YES
Exoscale	Geneva	YES
Exoscale	Zürich	YES
Exoscale	Munich	YES
Exoscale	Vienna	YES

Exoscale	Sofia	YES
Scaleway	Amsterdam	YES
Scaleway	Paris	YES
Scaleway	Warsaw	YES
IONOS	Frankfurt	YES
Deutsche Telekom	Magdeburg	YES
Ventus Cloud	Vienna	YES
Ventus Cloud	St. Gallen	
Ventus Cloud	Marchtrenk	YES

3.2.6 Cloud Benchmarker Service

The Cloud Benchmarker Service was introduced to measure the performance characteristics of cloud storage locations in a repeatable way. While QoS metrics are collected anonymously for SkyFlok users, these are quite heterogeneous. Users' physical location and internet connection characteristics vary greatly and significantly affect the perceived performance of storage locations. To remove this variability, the Cloud Benchmarker Service performs measurements from a single location in Romania, using fly.io infrastructure. This removes any variability resulting from different geographic locations and, to some degree, from the connection's characteristics. Measurements are scheduled so that all locations are covered. A random selection of results is shown in Figure 4.

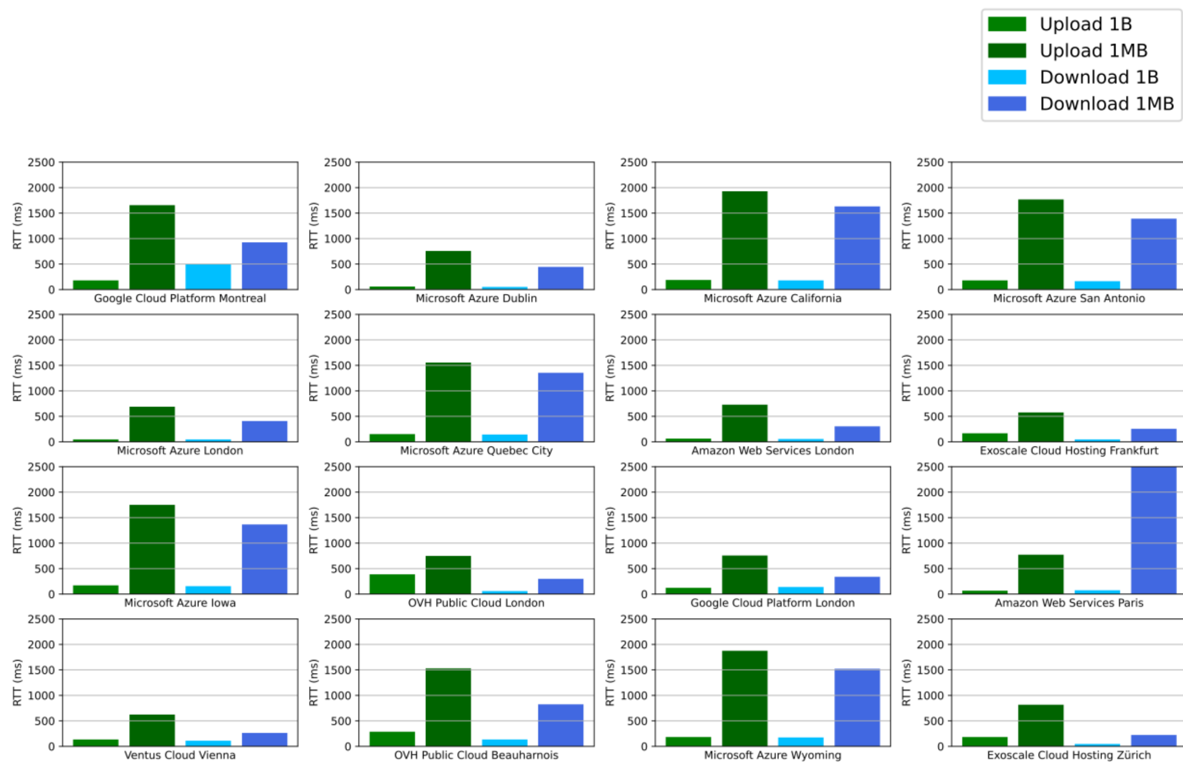


Figure 4: A random selection of Cloud Storage Location performance characteristics, averaged over 7 days. Each cloud location was measured once a day.

The decision to deploy to Romania was made in an effort to estimate cloud resource characteristics better, as observed from the K8s cluster of UVT. We plan to extend the deployment of the Cloud Benchmarker Service to different locations within Europe. This will make it possible to provide a more exact estimation of cloud location performance for a broader geographic user pool. For example, by also measuring in Western (e.g., Paris) and Northern Europe (e.g., Copenhagen), we could use interpolation to estimate expected performance results for users in Central Europe (e.g., Prague). Moreover, we are exploring the possibility of disseminating these results to the general public through a simple website as part of Work Package 7.

The Skyflok.com backend provides the scheduling for performing the above measurements. To this end, using a Google App Engine cron job that runs once every 24 hours, the Backend creates a list of measurements that should be performed in the subsequent 24 hours, distributing them evenly over time. Measurements are bundled in small batches for practical and cost reasons and are defined as Google Cloud Tasks. The tasks provide a callback mechanism through which the Skyflok.com backend issues the measurements to the Cloud Benchmarker Service.

The Cloud Benchmarker Service authenticates that the request comes from the Skyflok.com backend using a pre-shared key and then executes it. The results are returned to the Backend, which stores them for future queries. Whenever an application like the SERRANO Telemetry Service calls the Storage Service's Cloud Telemetry API, the Gateway fetches these results from the Backend.

3.3 Public APIs

The SERRANO-enhanced storage service exposes three distinct APIs to the users of the SERRANO platform as well as the other platform services. The Secure Storage API exposes the core storage-related features. The Storage Policy API allows for the management of storage policies. The Telemetry API provides information about the storage locations. All three are documented using OpenAPI 3.0. The JSON-formatted file[4] is uploaded to the Gateway's SERRANO Github.com repository.

3.3.1 Secure Storage API

The Secure Storage API provides SERRANO users with a way to store and retrieve files. It is based on what can be considered the industry standard for object storage: Amazon Web Services S3. The decision to use a well-known API brings significant benefits, as it allows users to integrate their existing software solutions with the SERRANO platform seamlessly. There are S3 client libraries for most programming languages, along with countless tools for all common operating systems.

[4] OpenAPI 3.0 definition of services offered by the SERRANO-enhanced Storage Service https://github.com/ict-serrano/On-Premise-Storage-Gateway/blob/master/openapi.json

Amazon's S3 service offers object storage. Objects are immutable, versioned entities with a key as a unique identifier and may have other associated metadata. Objects are organised into buckets, which have a name that is unique across the system and may also have metadata associated with them. There are several distinctions between file systems and object storage solutions. However, cloud storage solutions like Dropbox and Google Drive have shown that object storage semantics and characteristics, while being somewhat less permissive than those of file systems, are suitable for file storage. Indeed, SkyFlok also builds on this observation to offer its users a file system that is built on top of object storage.

The Secure Storage API supports all major Create, Read, Update, Delete (CRUD) features of both objects and buckets in their simplest form. Thus, the project achieved the target value of 100% for **SIR.2** KPI, defined in Deliverable 6.6 (M27).

Buckets

- CreateBucket
- DeleteBucket
- ListBuckets

Objects

- GetObject
- PutObject
- ListObjectsV2
- DeleteObject

An API reference can be found on Amazon's website:

https://docs.aws.amazon.com/AmazonS3/latest/API/API_Operations_Amazon_Simple_Storage_Service.html

Amazon Web Services S3 provides two URL schemas to access buckets and their contents. The Secure Storage API adopts the first one. The second scheme was created by Amazon to address the bottlenecks related to routing requests through DNS on a global scale. Given that the SERRANO-enhanced Storage Service operates at a much more modest scale, the second schema does not bring any benefits.

1 *http://s3.amazonaws.com/[bucket_name]/*

2 *http://[bucket_name].s3.amazonaws.com/*

The Secure Storage API uses the same parameters for each endpoint and maintains the error handling of AWS S3, both in terms of the format of error messages as well as the different codes that identify the cause.

Development of the SERRANO-enhanced Storage Service will continue based on the three use cases' requirements. Some S3 features, such as Access Control Lists (ACL) and multi-part uploads, are planned and will be executed if needed. In all cases, compatibility with the S3 API as used by Amazon will be maintained.

3.3.2 Storage Policy API

The Storage Policy API allows the platform's users as well as the SERRANO Resource Orchestrator to create, update and retrieve storage policies. These are the recipes used to translate an application's storage task's requirements to a storage resource allocation. The ARDIA framework developed in Work Package 5 contains both the Application and the Unified Resource Model definitions.

Figure 5 shows an example of a JSON configuration file for a storage policy. It contains the list of cloud storage locations (*backends*) and SERRANO edge device locations (*edge_devices*). It also defines the *compression* and *encryption* policy in use as well as the method used to add *redundancy* to the data. The mechanism can be tailored using parameters.

```
"name": "S3_test_storage_policy_hybrid",

"description": "Used to run integration tests in INTRAsoft K8s
cluster and locally using Docker.",

"edge_devices": [1,2,3],

"backends": [144],

"redundancy":
            {
                "scheme": "RLNC",
                "redundant_packets": 1
            },

"compression":
            {
                "scheme": " DEFLATE",
                "wbits": -15,
                "level": 9
            },

"encryption":
            {
                "scheme": "AES",
                "key_size": 256,
```

Figure 5: Example storage policy definition that uses compression, encryption and erasure coding. Erasure coded fragments are distributed to both cloud and edge locations with a redundancy factor of 100%.

The storage policy API is RESTful and exposes the following endpoints:

Storage policy

- CreateStoragePolicy
- ListStoragePolicies
- GetStoragePolicyOfBucket
- EditStoragePolicy
- DeleteStoragePolicy

3.3.3 Cloud and Edge Telemetry API

The Telemetry API is used to expose information about cloud storage locations. This information is used by the SERRANO Resource Orchestrator as input when automatically creating storage. It is also used by the SERRANO Telemetry Service to monitor the state of the cloud storage locations as resources of the SERRANO platform.

One endpoint (*/cloud_locations*) exposes the following static characteristics for each cloud location:

- unique identifier used in storage policies
- provider name: Google, Amazon, ….
- url
- geographic location – longitude and latitude coordinates
- country
- city/region
- jurisdiction
- GDPR compliance
- storage cost in $ / GB / month
- ingress cost in $ / GB
- egress cost in $ / GB

as well as several dynamically measured parameters:

- Round-trip times of uploading 1B of data
- Round-trip times of uploading 1MB of data
- Round-trip times of downloading 1B of data
- Round-trip times of downloading 1MB of data
- list of cases when the cloud location was unavailable.

The dynamic parameters are provided for the last 30 days, except for reports on unavailability. These are included for the entire duration of measurements and can be used to estimate the expected availability of each cloud. The measurements performed for 1B of data can be used to estimate the latency of accessing the storage. The measurements performed for 1MB of data, together with estimations on access latency, can be used to calculate an estimation for the upload and download throughput offered by each cloud location. Details on how the measurements are carried out can be found in Section 3.2.6.

The characteristics of edge locations are exposed using a separate mechanism. A second endpoint (*/edge_locations*) provides a list of SERRANO devices along with their static characteristics:

- unique identifier used in storage policies
- name
- description
- cluster identifier
- storage URL of S3 endpoints
- team identifier
- S3 region

Dynamic characteristics are collected through a standard, Prometheus-compatible interface exposed by MinIO[5]. This provides a way to both monitor performance characteristics as well as configure alert rules for certain types of abnormal events.

The SERRANO orchestration uses the unique and the cluster identifier to establish the connection between the two sets of information. The identifiers can then be used to create storage policies.

3.4 Developer Portal

To enhance the usability (non-functional requirement NF_GR4 defined in Deliverable 6.6 (M27) of the SERRANO-enhanced Storage Service, we created a web portal to cater to the needs of the developers who will use the service. The features have been selected by studying the online interfaces of object storage providers and based on the project's requirements with particular emphasis on the use cases. The developer portal will also play an essential role in the exploitation of the project's outcomes.

[5] Monitoring and Alerting using Prometheus: https://min.io/docs/minio/linux/operations/monitoring/collect-minio-metrics-using-prometheus.html

3.4.1 Managing S3 buckets

The first category of features on the developer portal relates to the management of S3 buckets and their contained files. These features offer a graphical interface for some of the Secure Storage API's endpoints. This makes it possible for team members without technical training to use some of the service's main features. Buckets can be listed as shown in Figure 6.

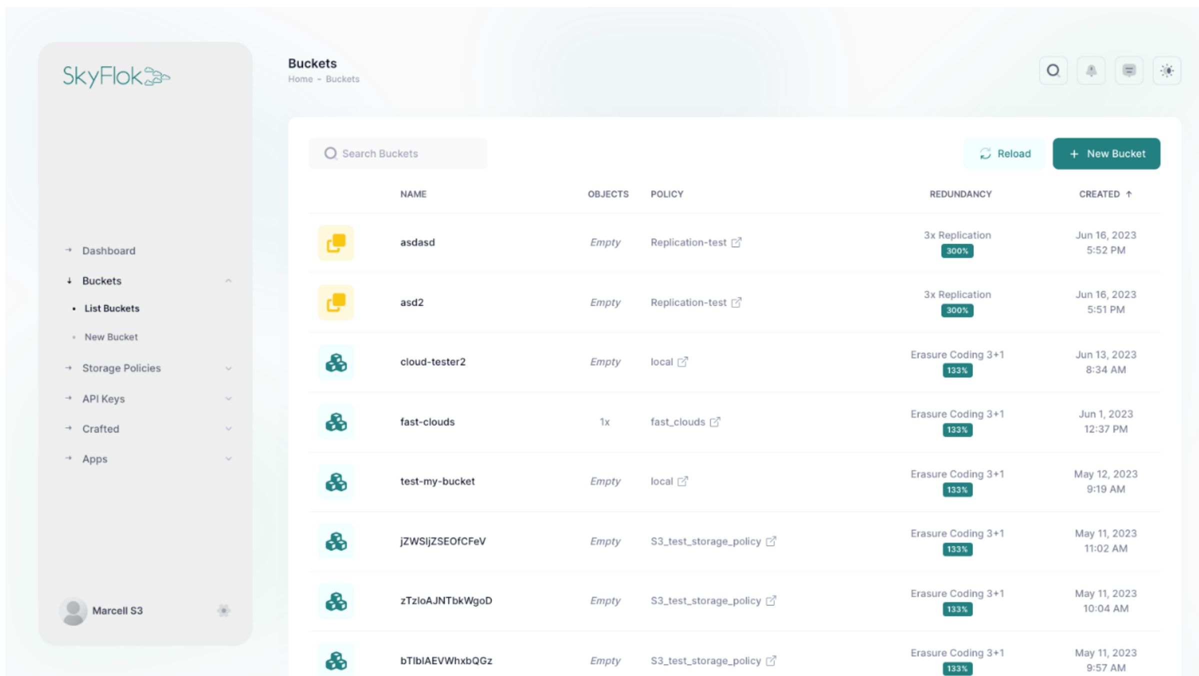


Figure 6: Developer portal - listing S3 buckets.

The details of a bucket can also be examined (e.g., CORS configuration, selected storage policy, ACL, etc.) and the list of objects (files) visualized as show in Figure 7.

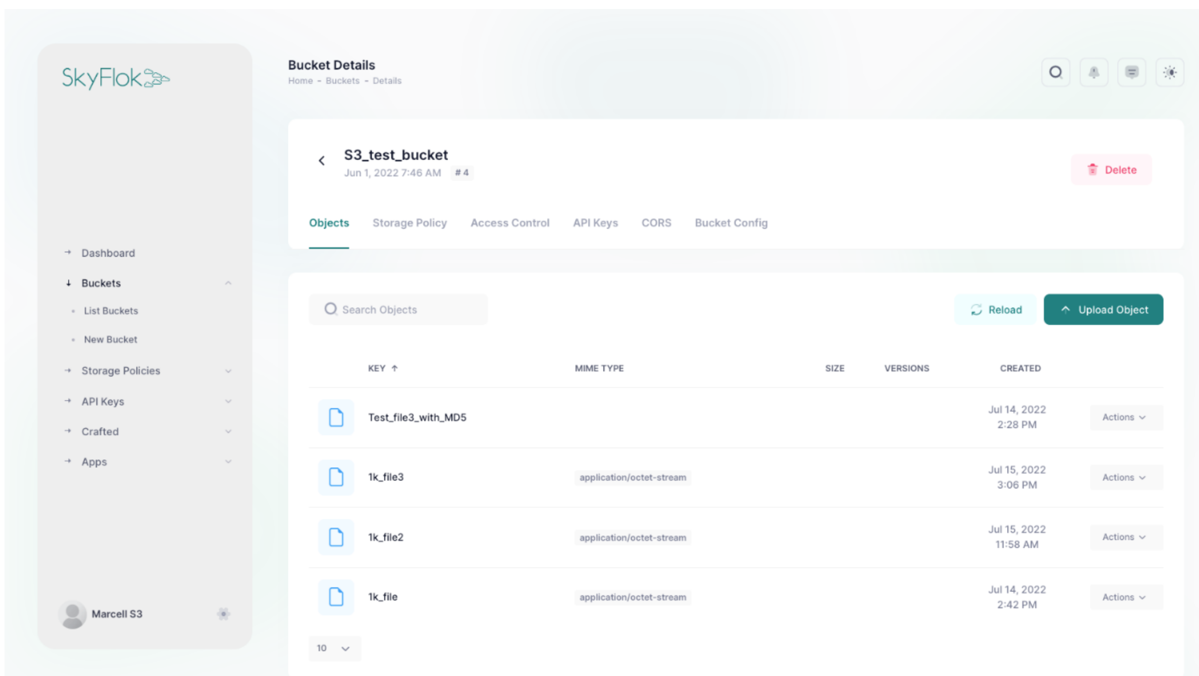


Figure 7: Developer portal- listing the contents of an S3 buckets.

New buckets can also be created using a simple wizard.

3.4.2 Managing storage policies

Storage policies are not a part of the S3 interfaces. As such, a graphical UI greatly enhances the experience of first-time SERRANO platform users. Policies can be listed (shown in Figure 8) and created using a simple wizard (shown in Figure 9).

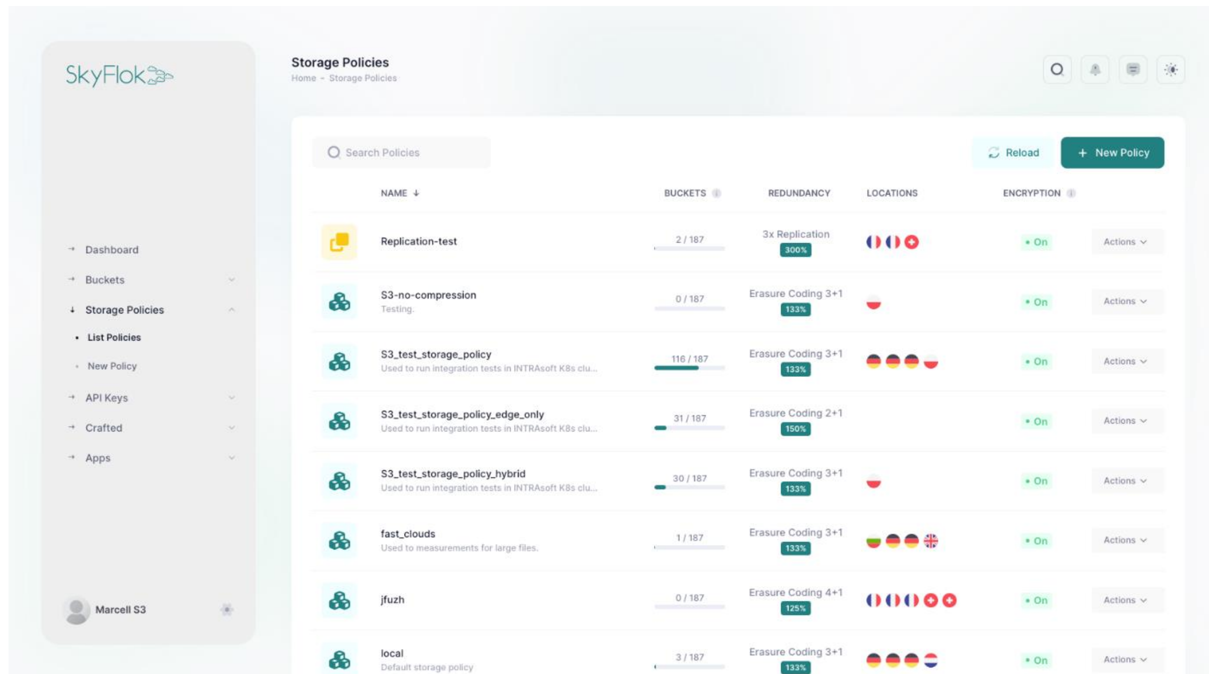


Figure 8: Developer portal - listing Storage policies.

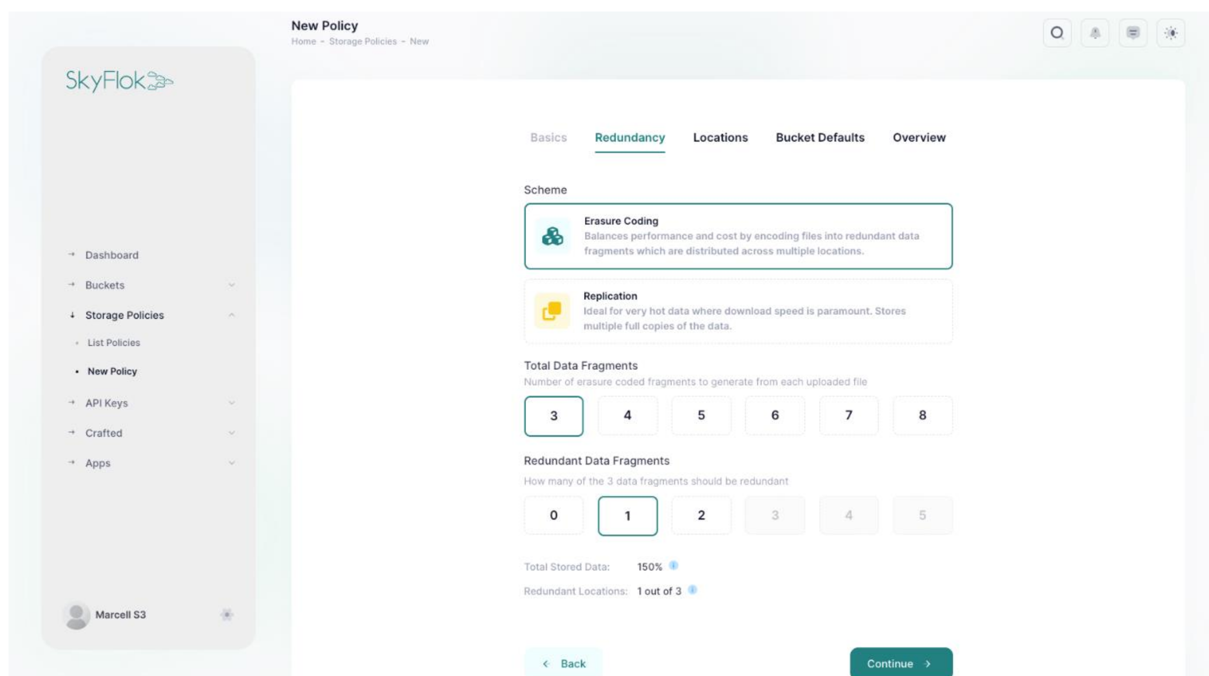


Figure 9: Developer portal - creating a new storage policy.

3.4.3 Managing API keys

API keys are used similarly to Amazon AWS keys for authenticating requests. In fact, they are a natural part/extension of the S3 interface and are adopted in some form by most, if not all, S3-compatible object storage solutions.

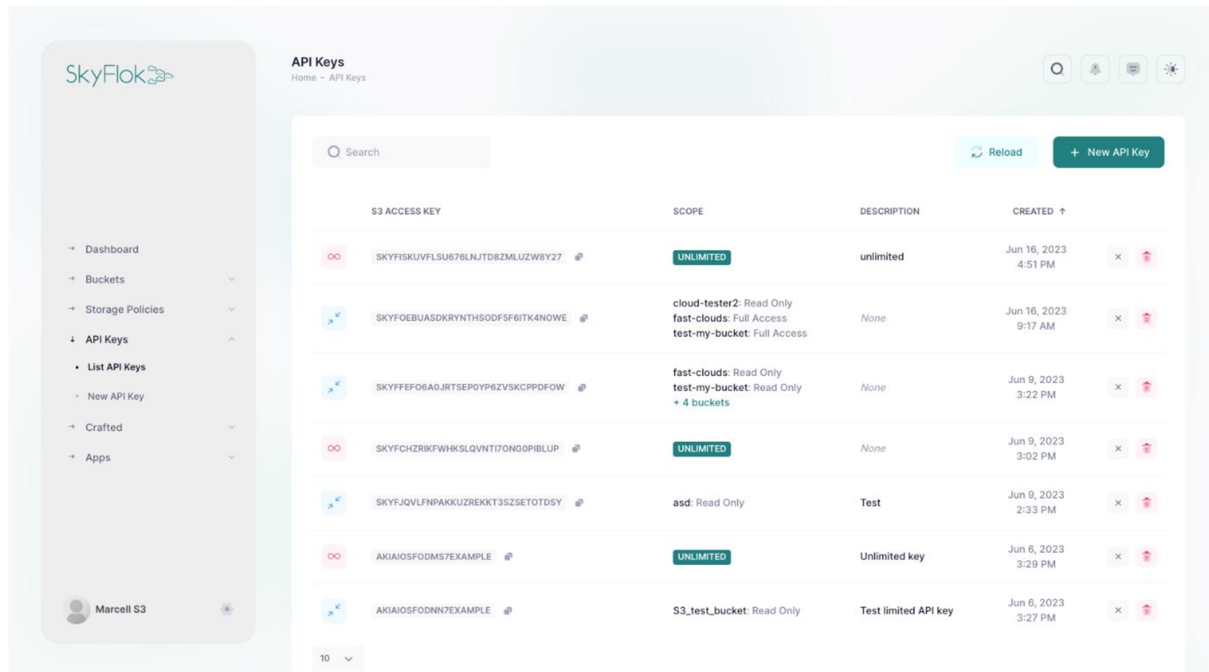


Figure 10: Developer portal - listing API keys.

The portal provides a way to list keys, as shown in Figure 10. New keys can also be added (Figure 11). We provide a way to limit the keys' capabilities by scoping them per-bucket to have full, read-only, or no access to a bucket. This feature closely resembles AWS S3's canned ACLs.

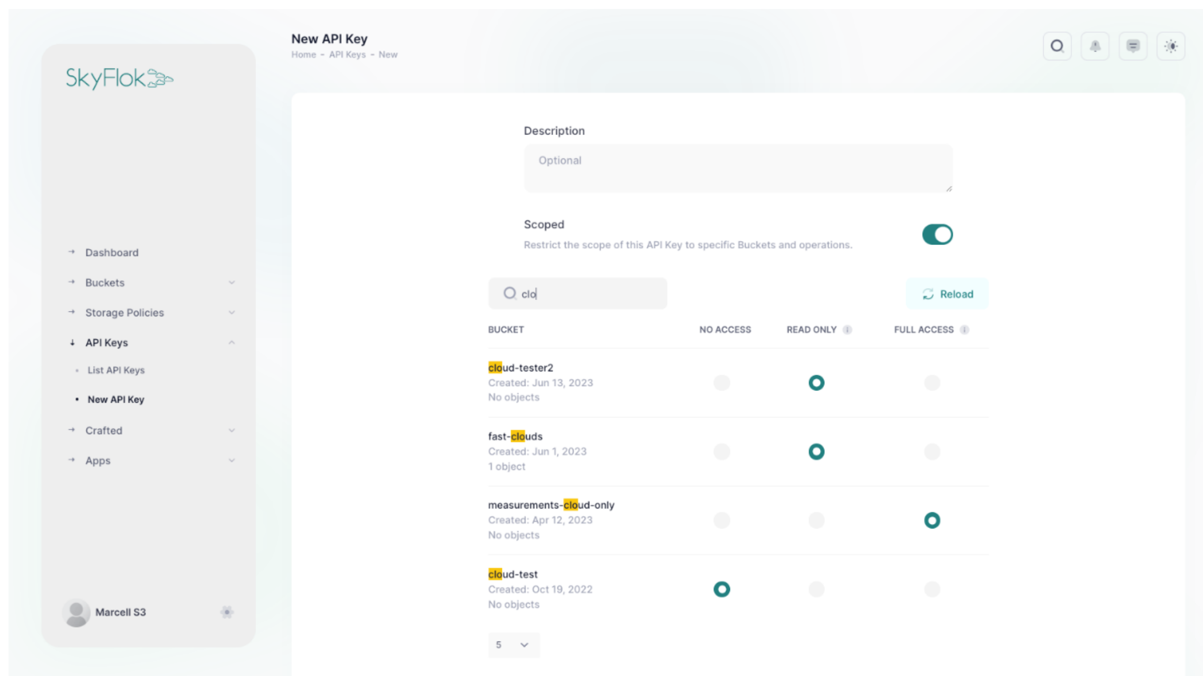


Figure 11: Developer portal - creating a new API key.

3.5 Performance Considerations

3.5.1 Reducing the number of edge-cloud HTTP calls

Several steps have been taken to increase the performance of downloading and uploading files. Firstly, the number of HTTP requests made from the Gateway to the Skyflok.com backend has been reduced as much as possible. While requests between services of the Backend have low latency as they are deployed on the same infrastructure, the Gateway is situated on the edge, and thus every call made to the Gateway suffers from significant latency. Round-trip times for even very simple calls can range between 200-500ms.

When uploading a file, the Gateway first authenticates with the Skyflok.com backend and requests download links as well as information such as the storage policy of the bucket. The Backend performs several checks for each file upload (e.g., bucket ownership) as part of this process. However, since the beginning of the project, these have been consolidated into a single edge-cloud HTTP call. When it finishes uploading the fragments, the Gateway signals this to the Backend, which can then store the necessary metadata. Since storage policies are immutable, they are cached by the Gateway and only requested from the Backend the first time a bucket is accessed.

File downloads have been streamlined to an even greater extent. The Gateway requests all metadata required to download fragments from the Backend when the S3 call is received. Following this, it completes the process autonomously. We provide an example of the time taken to upload and download a single file of 10MB in Figure 12. Three different storage policies have been used. The cloud-only policy is slowest as the time required to upload and

download fragments is significant. The edge-only policy almost eliminates this delay entirely, while the hybrid policy is somewhere in between.

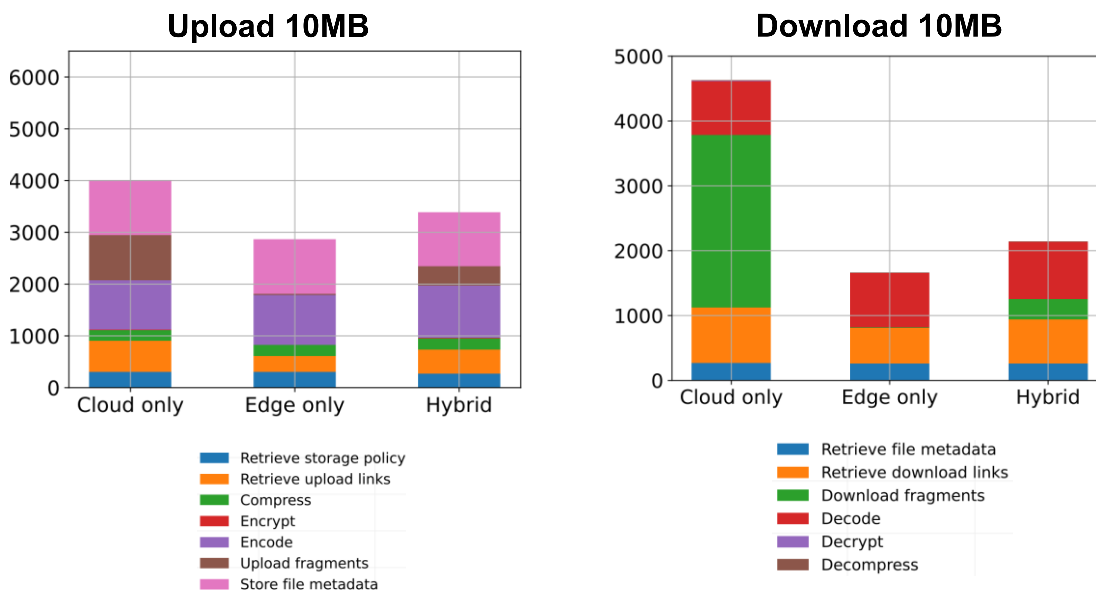


Figure 12: Preliminary measurements for file upload and download of a single 10MB file.

3.5.2 File caching

To further improve file download performance, we introduced a file caching solution. Since the Gateway has been designed to be stateless and a file cache affects this highly desirable property, caching is an opt-in service. If a user decides to utilize it, they must have mechanisms in place to guarantee data consistency. A straightforward approach is only to deploy a single instance of the Gateway, thus ensuring that the cache is always consistent with the distributed data. A more complex approach requires sharding of the data, where each instance of the Gateway is used to access a subset of a user files. For example, sharding can be performed by taking the S3 bucket as a separator. Other mechanisms can also be put in place.

The Gateway employs write-through caching. As files are uploaded through the S3 interface, the Gateway stores a copy of the original data on a local, typically ephemeral, file system. Internally, it records cached files along with their size, bucket, and time of caching. Downloads that result in a cache hit can be served from the file system. Cache misses are served by downloading the fragments, decoding, decrypting, and decompressing the data. When a file is deleted using the S3 interface, it is also removed from the cache. A Least Recently Used (LRU) approach is employed that favours recent files. If the cache fills up, older entries are deleted to make room for new entries. This is done following the assertion that more recent files are more likely to be accessed than older files. Users have the option to turn on read-through caching as well. In case of cache misses, the file is added to the cache as it is being downloaded.

Beyond simply turning the file caching feature on and off, the Gateway offers the ability to configure the strategy using the following parameters:

- Maximum size of the file cache
- Minimum disk space kept empty
- Maximum file size to be cached
- A blacklist of file types (MIME) that shouldn't be cached
- Enable or disable read-through caching

Currently, we offer a simple REST interface to configure these parameters. However, we might integrate this feature into storage policies, making caching a per-bucket setting based on future discussions with the UC providers and technical partners. Given that this is a recently implemented feature with some details subject to change, it will be documented in greater detail in WP6 deliverables as part of the Secure Storage Use Case at the end of the project.

3.6 Integration with SERRANO Platform Services

The SERRANO-enhanced Storage Service is a core component of the SERRANO platform and is integrated with other platform services to offer users an accelerated experience regarding file storage across the edge-cloud continuum. An AI-enhanced orchestrator automates the creation of storage policies, matching user intents with the optimal storage configuration. Here we limit the presentation to technical details on how three integrated workflows have been implemented as part of the Storage Service. More details on the benefits of the integrations have been and will be presented in WP6 deliverables in the context of UC1: Secure Storage. Furthermore, the implementation details of the other named platform services can be found in various deliverables from Work Packages 3, 4, and 5.

3.6.1 Automatic storage policy creation

Users declare their applications' intents using the ARDIA Framework. Each separate storage task is associated with a separate intent (applications may have multiple storage tasks and multiple applications may share a storage task), which is translated by the SERRANO platform to a newly created storage policy. User applications can assign the automatically created policy to newly created S3 buckets.

The process is designed to be simple and seamless for the user. Internally, several mechanisms are in effect. The Storage service acts as a source of information for storage resources. Information about cloud locations is exposed through the Cloud Telemetry API, whereas edge locations are simply listed in the Edge Telemetry API. The SERRANO Telemetry Service consumes and aggregates this information and also retrieves telemetry data from the SERRANO edge devices directly using a Prometheus-compatible interface. The AI-enhanced

Service Orchestrator maps the intents to an internal representation that it supplies to the Resource Orchestrator. The Resource Orchestrator consumes the aggregated information related to storage resources and uses the algorithms defined in the Resource Optimization Toolkit to create an optimal data distribution. This is the core of a storage policy and includes the redundancy scheme as well. The creation of the storage policy is done through the Storage Policy API.

3.6.2 Acceleration of data processing

The Gateway's file upload and download workflows include three data processing step pairs: compression, encryption, and erasure coding. The Gateway uses the accelerated algorithms developed as part of WP for encryption and erasure coding to increase performance. Encryption is accelerated using GPUs, while FPGAs are employed for erasure coding. The final integration work is in progress and will use a set of libraries developed by AUTH. These will be called directly from the Gateway's Python code whenever the appropriate hardware is available.

3.6.3 Offloading of encryption for TLS connections

To reduce CPU load, the Gateway offloads the task of encrypting TLS connections to DPUs, leveraging NVidia Bluefield cards when available. The Gateway is left unchanged, and the offloading is performed seamlessly. Instead of relying on the Gateway's web server, Uvicorn, to provide TLS encryption, a nginx TLS termination proxy is deployed. This forwards all HTTPS connections to the Gateway through HTTP. We use a relatively new version of nginx which supports a feature called kernel TLS. The switching between CPU and DPU-based encryption is done by a custom OpenSSL implementation developed by Nvidia. More details can be found in Deliverable 3.2 (M15).

3.7 Privacy and Security

Data privacy and security are critical concerns for almost any IT system's users. This is especially true for enterprises, where the nature of the data means its protection is critical to complying with laws and regulations. For example, GDPR regulations explicitly and precisely stipulate how private data must be handled. While cloud storage has steadily increased its adoption rate, concerns over its security remain. Cloud providers or user accounts of cloud services may be hacked. Furthermore, some cloud providers explicitly state that they scan files uploaded to their service[6]. Unfortunately, local storage solutions have their own security issues. Most small to medium enterprises do not have the capability or know-how to match the level of security afforded to cloud services by having dedicated teams to manage software patching, monitoring of network traffic and other security-related operations.

[6] Google, "Is Google Drive secure?," [Online].
Available: https://support.google.com/drive/answer/141702?hl=en&ref_topic=2428743.

The Secure Storage use case presents a hybrid system that includes both cloud-based and on-premises storage. As such, it has to deal with a larger attack surface than purely local or purely cloud-based solutions. To counter this increased threat, the SERRANO-enhanced Storage Service includes a wide range of platform-level measures adopted to ensure a high level of data security and privacy.

Data in transit can be intercepted by either a man in the middle or a snooping attack. To limit the danger posed by such attacks, all communication between the user and the service, as well as the different components of the service, must be encrypted. In practice, this is ensured through HTTPS where the TLS protocol provides protection through AES encryption. Furthermore, the storage service employs two additional design principles to limit the attack surface. First, data does not leave the company's infrastructure without being encrypted. To this end, the AES-GCM encryption used to protect data at the storage locations is applied before the data reaches the public internet. As such, even if an attacker somehow decrypts the TLS traffic in any place outside the company's premises, it will still not be able to access the original data. Second, data is always transferred directly between the Gateway and the storage locations. No data transits the Backend, thus lowering the attack surface.

3.7.1 S3 authentication

All requests on the S3 Secure Storage API must be signed using the most recent version of Amazon's authentication system: AWS Signature Version 4[7]. The Gateway calculates the requests' signature in the manner described in Amazon's documentation and compares it to the signature supplied by the client application. If the signatures do not match, the request is rejected. The signature is calculated based on a large number of different parameters. Beyond its core role of authenticating requests, the signature process is designed to protect against tampering as part of a man-in-the-middle attack and replay-based attacks, as it has a time component. Access is provided with a pair of credentials in the form of an access key ID and a secret key. The secret key is one of the inputs to the signature verification process. The credentials are provided to teams, where a team is a collection of users. There is no possibility to access files from a different team than your own.

Given that the Gateway is deployed to the customers' premises and applications that access it are likely running on the same infrastructure, on a protected internal network, this level of request signing may be considered somewhat unnecessary. Furthermore, large requests such as file uploads carry a performance penalty as the SHA-256 hash of the HTTP request payload must be calculated on the Gateway. Nevertheless, this mechanism makes it possible to deploy the Gateway to a more challenging environment in terms of security. Furthermore, it is a reasonable assumption that the operators of S3 client applications expect such security mechanisms to be in place.

[7] Amazon - Authenticating Requests (AWS Signature Version 4)
https://docs.aws.amazon.com/AmazonS3/latest/API/sig-v4-authenticating-requests.html

3.7.2 Pre-signed URLs

A key design consideration that benefits both scaling and security is that the owners of the data upload/download fragments of their files directly to/from storage locations. As such, the Skyflok.com backend does not handle data directly. This is achieved by generating pre-signed URLs (temporary URLs in OpenStack Swift parlance) on the Backend to connect users directly with the data stored in the storage locations.

Whenever a file is to be uploaded or downloaded, the Gateway authenticates with the Skyflok.com backend and requests a set of upload/download links. The links point to protected resources on the cloud and edge locations. This poses a security challenge; how can the Gateway be trusted with the credentials necessary to access these resources? This component is outside the control of the service's creators, as it is deployed on customer infrastructure. If these credentials are stored on site, they will provide unauthorised access to data.

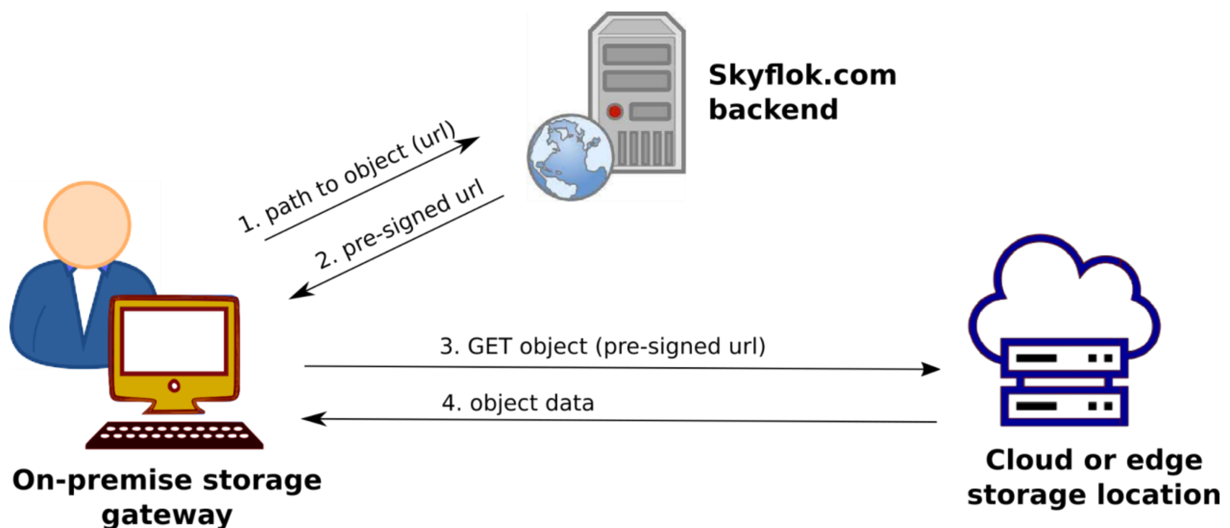


Figure 13: Using pre-signed URLs to download a file

The solution is to create a special set of cryptographically signed links – pre-signed URLs – which do not require additional credentials. The links have a short lifespan and can generally only be used once. The signature is calculated using asymmetric-key cryptography and is guaranteed to be different every time. Given the elegance of this solution, CC has also integrated the SERRANO edge devices with the help of this feature. Figure 13 illustrates the process of downloading a file using a pre-signed URL.

Closely tied to this technique is the design principle that data never leaves the customer's premises unencrypted. Therefore, data privacy will be maintained even if cloud locations outside the user's control are compromised. The SERRANO-enhanced Storage Service relies on industry-standard AES encryption using the GCM block scheme. It is included in the NSA Suite B Cryptography and is used in many communication protocols such as TLS 1.3, SSH, and IEEE 802.1AE (MACsec) Ethernet security. 256-bit keys are used along with randomly generated initialisation vectors.

3.7.3 Random Linear Network Coding

After a file is encrypted, it is also erasure-coded using Random Linear Network Coding (RLNC)[8]. RLNC is a novel erasure coding technique that creates a set of coded fragments or packets. Figure 14 illustrates the encoding process through the example of a single-generation 1000-byte file. First, the example file is sliced into four (4) separate symbols, each 250 bytes long. These are combined using a set of randomly generated coefficients into six (6) coded packets. The coefficients are selected from a mathematical structure called a finite field. Different sized fields have different trade-offs in terms of performance and reliability, with the Galois Field of size 2^8 being a relatively common choice. This is partly due to practical reasons, as each field element can be represented using a single byte. In this particular example, two packets are redundant and have the purpose of protecting against temporary or permanent storage location issues. In other words, any 4 of the 6 packets are enough to reconstruct the original file. For a similar level of protection using conventional replicated storage, 3 copies of the original file would need to be stored. This would constitute a 200% overhead compared to the 50% overhead achieved by RLNC in this example.

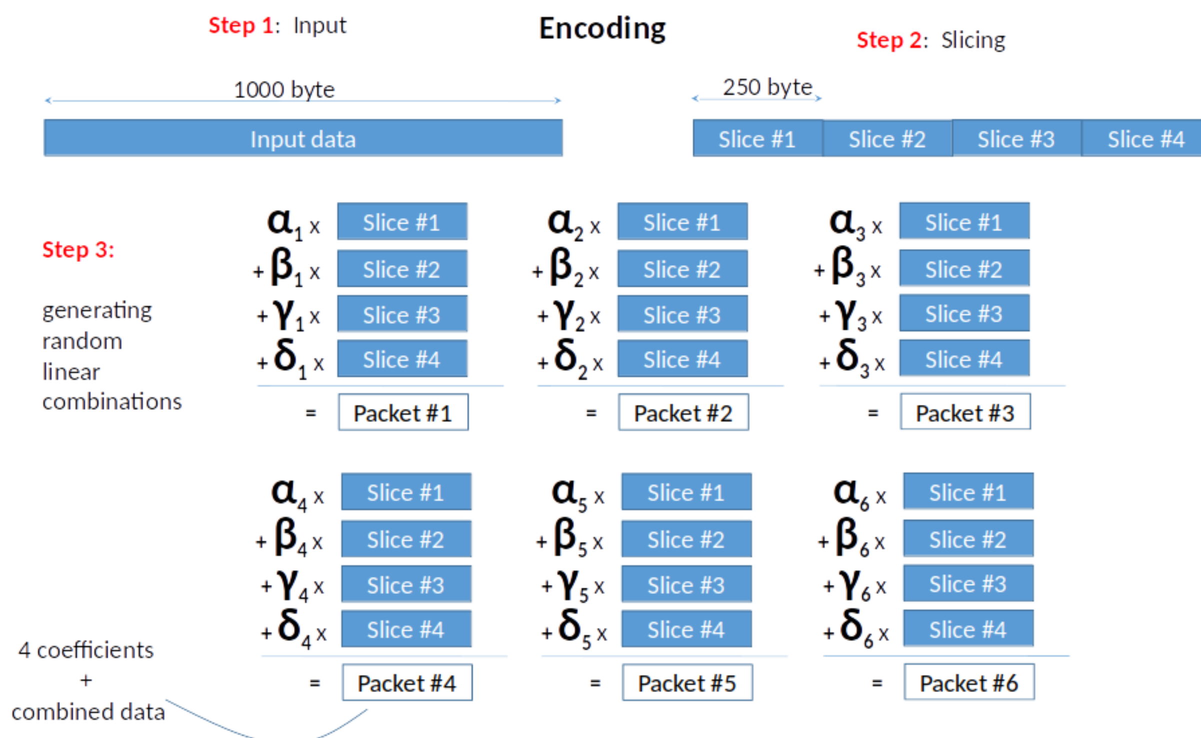


Figure 14: RLNC encoding of a 1000-byte file using 4 symbols that are combined into 6 erasure-coded packets.

[8] "Random Linear Network Coding (RLNC)-Based Symbol Representation," https://tools.ietf.org/id/draft-heide-nwcrg-rlnc-00.html.

In most applications, the packets (coded fragments) also include the coefficients used to create them. This makes it possible to decode without needing to look them up separately. However, they can be stored separately, further enhancing the system's security. In this case, beyond the conventional roles of an erasure code, RLNC also acts as a simple cipher. Because the coded fragments are created as linear combinations of the original data using random coefficients, an attacker must guess the coefficients correctly to decode the data. Furthermore, the data distribution to physically separate cloud and edge locations also has a privacy-enhancing effect. A malicious actor would need to break into several locations and retrieve the data before attempting to solve the system of linear equations and then decrypt the data. In other words, access to a subset of coded fragments with a combined size smaller than the original data provides the attacker with no information about the original data.

Beyond defending against attacks directed against cloud storage locations and even snoopy cloud providers, these measures are also beneficial if the edge storage locations are less secure. This is the case when an edge location is located in an environment that does not have physical access controls of the calibre of those found in a modern datacentre or an enterprise-grade server room.

The metadata and the encryption keys for protecting data at rest are stored in the Skyflok.com backend, hosted using Google Cloud Services. To access them, an attacker would either need to compromise Google's security infrastructure or a user account with access to this information. To limit the attack surface, only the CEO and CTO of Chocolate Cloud have access and must use a complex password and 2-factor authentication.

4 Trust and Isolation on Untrusted Physical Tenders

A software supply chain attack occurs when malicious code is purposefully added to a component that is sent to target users. The code may be introduced to the component in several ways, such as via compromise of the source code repository, theft of signing keys, or penetration of distribution sites and channels. Customers unknowingly acquire and deploy these compromised components onto their systems and networks as part of an authorized and normal distribution channel. Advanced malicious code typically does not disrupt normal operations and may not activate for several days or weeks, thereby remaining hidden from typical application and software testing practices.

For instance, a telecommunications company buys core network systems management software from a trusted provider; however, unbeknownst to the trusted provider, one of the components it uses in the product has been compromised and now contains malicious code. The deeper into the supply chain it occurs, the more difficult it is to identify in advance. The malicious actor may use this inserted vulnerability as part of a larger attack chain that uses the malicious code to gain access within the telecom core network and pivot towards other attack vectors.

The malicious actor may use this inserted vulnerability as part of a larger attack chain that uses the malicious code to gain access within the telecom core network and pivot towards other attack vectors. For instance, a potential attack vector stems from ‘persistent threats’, where malware is inserted into a system so that the platform always boots in a compromised state, even after legitimate software is re-installed. To combat this attack, system vendors are turning to Secure Boot and Measured Boot to assure that when a platform boots, it is running code that has not been compromise.

4.1 Hardware Trust

An edge device is susceptible to several attacks by malicious users, physical or otherwise. For instance, one could gain access to the storage backend of an edge server (e.g., an SD card, eMMC memory etc.) and tweak the operating systems binary that drives the device to relay data to a malicious third party. This type of attack is relevant to all three use cases of the SERRANO platform. To prevent this, mechanisms such as Secure boot and Measured boot are introduced.

The terms Secure Boot and Measured Boot are often seen together, and they can be complementary, but they are not at all the same. Both technologies rely on a “Root of Trust”, that is, some piece of code or hardware that has been hardened well enough that it is not likely to be compromised and either cannot be modified at all or else cannot be modified without cryptographic credentials.

For many systems, that "Root of Trust" is provided by the Unified Extended Firmware Interface (UEFI) BIOS[9] code that takes the place of the ad-hoc “legacy” BIOS that has been in use for years. The UEFI BIOS works with platform hardware to ensure that the flash memory that contains the BIOS cannot be modified without cryptographic authority, thus forming the "Root of Trust".

A UEFI BIOS depends on several elements to ensure the Root of Trust is not compromised:

- The BIOS contains a public key that is controlled by the equipment manufacturer. Any authorized change to the BIOS must be signed with the corresponding private key.
- The BIOS itself is required to check the validity of the signature on a proposed update using the public key stored in a protected part of the BIOS flash.
- The BIOS must configure processor hardware features to block any unauthorized writes to the flash. In an x86 design, Protected Range Registers are one line of defence, with other mechanisms also available.

Both Secure and Measured Boot start with the Root of Trust and extend a "chain of trust", starting in the root, through each component, to the Operating System (and in embedded systems, often to the application itself). However, once a Root of Trust is established, Secure Boot and Measured Boot do different things.

Modern platforms of all sorts often use a multi-stage boot, where firmware in flash launches an OS Loader (such as Grub2 or u-boot), which then loads and launches a series of OS components.

4.1.1 Secure Boot

In a Secure Boot chain (Figure 15), each step in the process checks a cryptographic signature on the executable of the next step before it is launched. Thus, the BIOS will check a signature on the loader, and the loader will check signatures on all the kernel objects it loads. The objects in the chain are usually signed by the software manufacturer, using private keys that match up with public keys already in the BIOS. If any of the software modules in the boot chain have been hacked, then the signatures will not match, and the device will not boot the image. Because the images must be signed by the manufacturer, it is generally impractical to sign any files generated by the platform user (such as config files).

Secure Boot is relatively self-contained. If the handful of signed objects has not been tampered with, the platform boots, and the Secure Boot process is done. If objects have been changed so the signature is no longer valid, the platform does not boot, and a re-installation is indicated.

[9] Unified Extensible Firmware Interface Forum: http://www.uefi.org/

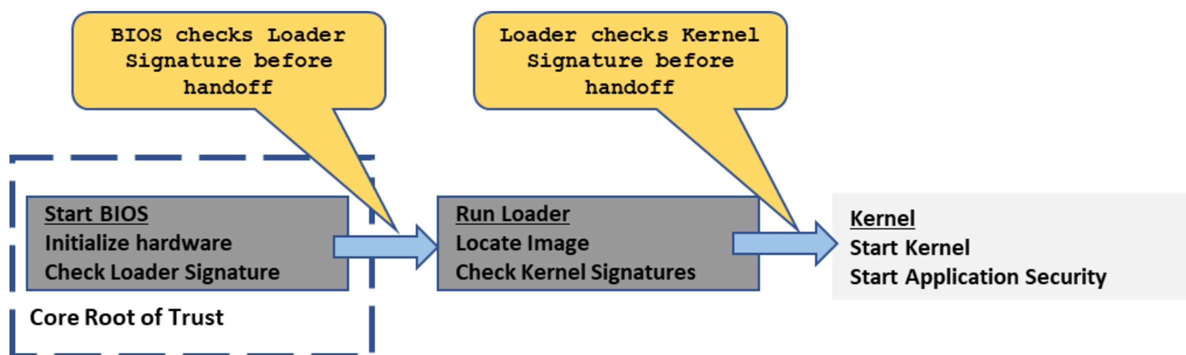


Figure 15: Secure boot execution flow

4.1.2 Measure Boot

In a Measured Boot chain (Figure 16), we still depend on a Root of Trust as the starting point for a chain of trust. However, in this case, before launching the next object, the currently-running object "measures" or computes the hash of the next object(s) in the chain and stores the hashes so that they can be securely retrieved later to find out what objects were encountered. Measured Boot does not make an implicit value judgement as to good or bad, and it does not stop the platform from running, so Measured Boot can be much more liberal about what it checks. This can include all kinds of platform configuration information, such as which was the boot device, what was in the loader config file, or anything else that might be interesting.

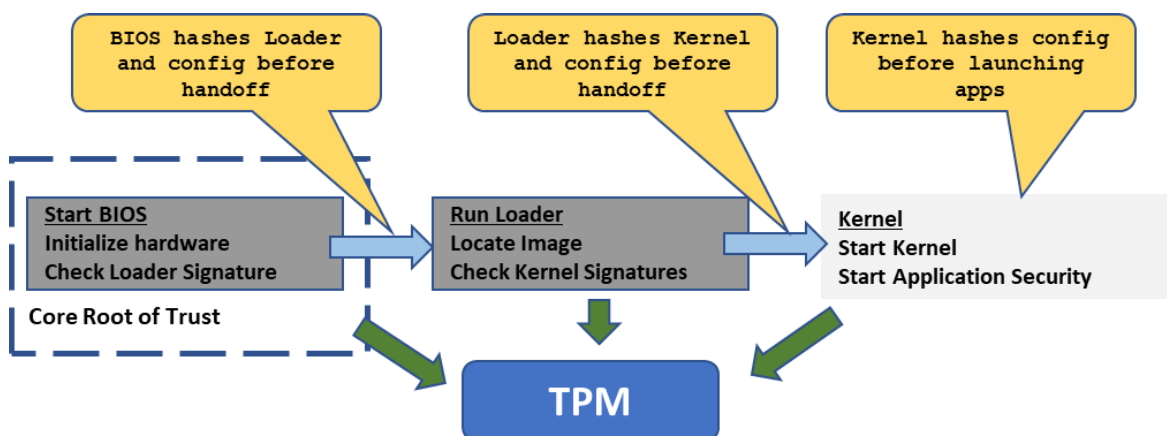


Figure 16: Measured boot execution flow

Measured Boot is more flexible but also requires an important step. All those hashes have to be stored so that there is very little chance that they can be manipulated and a very high likelihood that they can be reliably reported to a management station using a process called Attestation. Since Measured Boot does not stop the platform from booting, the host OS cannot be relied upon to report the hashes.

In the case of Measure Boot, the Trusted Platform Module (TPM) is used to record these hashes. The TPM is a small self-contained security processor that can be attached to a system bus as a simple peripheral[10]. Of the many functions a TPM can provide, one is the facility called Platform Configuration Registers (PCRs), used for storing hashes.

PCRs are registers in the TPM that are cleared only at hardware reset and cannot be directly written. One can "extend" values in PCRs, that is, to hash a new value into whatever was previously in the PCR. Thus, as the platform boots, each measurement can be accumulated in the PCRs to demonstrate which modules were loaded unambiguously.

Once the PCRs have been collected, the second step is for the TPM to report the values, signed by a key only the TPM can access. The resulting data structure, called a Quote, gives the PCR values and a signature, allowing them to be sent to a Remote Attestation server via an untrusted channel. The server can examine the PCRs and associated logs to determine if the platform is running an acceptable image.

Secure Boot and Measured Boot can be used simultaneously: Secure Boot ensures that the system only runs authentic software, and Measured Boot gives a much more detailed picture of how the platform is configured.

4.1.3 Trusted Platform Module as the Silicon Root of Trust

A TPM device supports many cryptographic functions. Notably the "PCR Extend" and "Seal" operations are used in popular measured boot architectures. Below, we elaborate on the two functions used in the SERRANO measured boot process.

PCR Extend: The TPM can be asked to perform a "PCR Extend" command, where a particular hash value would be added to the existing hash value in a Platform Configuration Register (PCR), and the resultant hash value can be stored back in the same PCR. For instance:

$PCR_Content_{New}$ = Extend ($PCR_Content_{Current}$, New_Measurement)

One can only extend the current value in the PCR with a new hash value, and the existing contents cannot be overwritten. This provides these key capabilities:

1. The order of measurements - Final value will be the same if and only if the measurements are done in the same order.
2. The final PCR value captures the whole history of measurements - useful in quickly validating final states against expected state.
3. Deterministic - If one repeats the same history of measurements, the same final PCR value will be produced - Useful in validating a change in one of the input sequences.
4. Seal: The TPM can seal a given secret information against the current PCR values through a TPM command called "Seal". Once sealed, the information can be read back

[10] http://forums.juniper.net/t5/Security-Now/What-is-a-Trusted-Platform-Module-TPM/ba-p/281128

only through an unseal command, which will succeed only if those PCRs hold the same set of values as they were during the sealing operation. In other words, the secret cannot be recovered if those PCR values are not the same.

Using these TPM capabilities, we build a powerful solution to measure and validate the software state of the SERRANO edge platform. The measuring process is called Measured Boot, and the method of getting the measurements verified and attested is called Attestation.

In Measured Boot method each of the software layers in the booting sequence of the device, measures the layer above and extends the value in a designated PCR. For example, BIOS measures various components of the bootloader and stores these values in PCRs 0-7. Likewise, bootloaders measure the Linux kernel boot and store the measurements in PCRs 8-15. The Linux kernel has a feature called Integrity Measurement Architecture (IMA), where various kernel executables/drivers can be measured and stored in PCR 10.

During these Extend operations, the operations are recorded by the BIOS and the bootloader, in a special firmware table, called the TPM Event Log table, and this table is handed over to the Operating System (OS) during OS takeover. By playing the same sequence of Extend operations recorded in each TPM Event Log, the system can check and verify if the final PCR values match, and if so, then the Event Log (and hence the software layers) can be trusted.

4.2 Confidential Computing and Trusted Execution

Security has long been one of the key goals of systems design[11]. Cryptography has enabled the safe storage (at rest) and transmission (in flight) of important data. However, there is still a situation when data can be vulnerable. The applications decrypt the data in order to save them; therefore, the decrypted version of data is stored in RAM, CPU caches, and registers. In recent years, a high number of memory scraping and CPU side-channel attacks have been reported. Under these circumstances, the wide adoption of cloud and edge computing, where users cannot control the underlying infrastructure, raises significant concerns regarding the security of data in use. In that context, the user cannot trust any parts of the system stack that cannot control such as the host Operating System and the hypervisor.

Confidential computing aims to address the data in-use security concerns. Due to the reasons explained previously, confidential computing cannot be a software-level solution. Accordingly, it is based on hardware extensions that modern CPUs include and provide Trusted Execution Environments (TEE). A TEE is an enclave that isolates the code and the data of a workload from any other system component.

Depending on the implementation, a TEE might use fencing and locking mechanisms to ensure the isolation of the trusted code. Only specific cores and memory cases are used when a trusted code is loaded in a TEE, aiming to avert side-channel attacks. Furthermore, TEEs can also use encryption for the data stored outside the TEE resources. The communication with a TEE happens through a well-defined interface, and all I/O operations are encrypted. As a

[11] L. Smith, "Architectures for secure computing systems," MITRE CORP BEDFORD MASS, 1975.

result, TEEs isolate the code and data inside a TEE from any other process, user, or system component. Only the trusted code is able to view or modify the encrypted data.

The encryption and signing keys that are used from a TEE should be saved in a hardware module. That module can be the starting point (Root of Trust) and should be trustworthy. Except for encryption and signing keys, the RoT might contain other root secrets and a set of functions needed for the encryption or validation of data. The code and data (keys) of a RoT are usually stored in a read-only memory (ROM), restricting any modifications. Trusted platform modules (TPMs) described in previous sections are examples of RoT that can generate cryptographic keys and protect critical information such as cryptographic and signing keys, and passwords.

Using the RoT platforms can secure the underlying firmware and extend the trust to higher levels of the software stack. A verified firmware can verify the OS boot loader, which can verify the Operating System and extend the trust to the hypervisor and/or container engine. The process of extending the trust from a RoT to higher levels of the software stack is called a Chain of Trust (CoT).

Apart from the isolation, a TEE should be able to verify the integrity of an application code. Even if the code inside a TEE is isolated and cannot be changed, there is still the danger of someone tweaking that code before it is launched inside a TEE. To be able to verify that the workload running on the hardware node is indeed the one intended by the system, we use attestation: through attestation, the workload tenant can verify that the workload is running on a genuine, authenticated platform and that the initial software stack is the expected one.

Our goal is to support as many TEEs as possible. Vendors provide a wide range of security mechanisms for TEEs, from memory isolation (e.g., Intel MKTME, Arm External Memory (DRAM) Encryption and Integrity with CCA), application isolation (e.g., Intel SGX, Arm Trustzone, IBM Apllication isolation technology) and virtual machine isolation (e.g., Intel Trust Domain Extensions (TDX), AMD Secure Encrypted Virtualization (SEV) or IBM Protected Execution Facility (PEF). We focus more on the latter case since we target multi-tenant environments as well as due to the extra layer of isolation offered by hardware-assisted virtualization.

4.2.1 Workload attestation

To securely sign, verify and provide attestable metadata to containers that will be deployed on a cluster, we are using the Sigstore[12] project. Sigstore is an open-source project that provides digital signing and verification of container images. Within the container image supply chain, it establishes confidence and maintains the image's integrity by utilizing cryptographic digital signatures and transparency log technologies.

Sigstore consists of a set of tools:

- Cosign (signing, verification, and storage for containers and other artifacts)

[12] Sigstore - https://www.sigstore.dev

- Fulcio (root certificate authority)
- Rekor (transparency log)
- OpenID Connect (means of authentication)
- policy-controller (enforcing container orchestration policy)

Cosign: Tool for signing/verifying containers (and other artifacts) that ties the rest of Sigstore together, making signatures invisible infrastructure. It includes storage in an Open Container Initiative (OCI) registry.

Fulcio: A free root certification authority, issuing temporary certificates to an authorized identity and publishing them in the Rekor transparency log.

Rekor: A built-in transparency and timestamping service, Rekor records signed metadata to a ledger that can be searched but cannot be tampered with.

OpenID Connect: An identity layer that checks if you're who you say you are. It lets clients request and receive information about authenticated sessions and users.

Policy Controller: An admission controller for Kubernetes for enforcing policy on containers allowed to run.

How Sigstore works

We are using cosign to sign and verify software artifacts, such as container images and blobs. Cosign operates in two different modes: key pair mode and keyless mode. We have chosen the keyless mode as our preferred option to simplify the process and avoid the burdensome task of securely managing and distributing keys. In keyless mode, the Sigstore associates identities, rather than keys, with an artifact signature. To do that, it utilizes Fulcio to issue short-lived certificates, binding an ephemeral key to an OpenID Connect (OIDC) identity. Fulcio uses OIDC tokens to authenticate requests. Subject-related claims from the OIDC token are extracted and included in issued certificates. Signing events are logged in Rekor, a signature transparency log, providing an auditable record of when a signature was created.

Verifying identity and signing the artifact

The process of verifying identity and signing the artifact is the following:

- An in-memory public/private key pair is created.
- The identity token is retrieved.
- Sigstore's certificate authority verifies the identity token of the user signing the artifact and issues a certificate attesting to their identity. The identity is bound to the public key. Decrypting with the public key will prove the identity of the private key holder.
- For security, the private key is destroyed shortly after, and the short-lived identity certificate expires.

Users wishing to verify the software will use the transparency log entry rather than relying on the signer to safely store and manage the private key.

Recording signing event

To create the transparency log entry, a Sigstore client creates an object containing information allowing signature verification without the (destroyed) private key. The object contains the hash of the artifact, the public key, and the signature. Crucially, this object is timestamped. The Rekor transparency log "witnesses" the signing event by entering a timestamped entry into the records that attests that the secure signing process has occurred. Clients upload signing events to the transparency log so that the events are publicly auditable. Artifact owners should monitor the log for their identity to verify each occurrence. The software creator publishes the timestamped object, including the hash of the artifact, public key, and signature.

Verifying the signed artifact

When a software consumer wants to verify the software's signature, Sigstore compares a tuple of signature, key/certificate, and artifact from the timestamped object against the timestamped Rekor entry. If they match, it confirms that the signature is valid because the user knows that the expected software creator, whose identity was certified when signing, published the software artifact in their possession. The entry in Rekor's immutable transparency log means that the signer will monitor the log for occurrences of their identity and will know if there is an unexpected signing event.

4.2.2 Incorporating Sigstore in SERRANO

Signing

A set of steps is required to enable image signing using the Sigstore (Figure 17).

First, the image building process is automated using GitHub Action workflows. This approach grants us access to a GitHub Workflow identity token, which GitHub provides for each workflow run. This identity is specifically associated with the corresponding GitHub Action workflow. It includes additional metadata that helps identify the GitHub repository of the workflow, the workflow name, and more.

Using this OpenID Connect token available in the workflow's environment, we can sign the produced image using cosign. Acting as a Sigstore client, cosign will generate an in-memory public/private key pair and request a new short-lived certificate from Fulcio using the OIDC token and the key pair.

Fulcio then provides the certificate to sign the image. Fulcio will append the certificate to an immutable, append-only, cryptographically verifiable certificate transparency (CT) log, allowing for publicly auditable issuance.

Given the certificate, cosign will sign the image using the provided certificate and push the signature to the OCI Image Registry, where the image is stored.

The signing event is recorded in a transparency log entry. To achieve this, cosign creates an object containing information allowing signature verification without the (destroyed) private

key. The object contains the hash of the artifact, the public key, and the signature. Crucially, this object is timestamped.

The Rekor transparency log "witnesses" the signing event by entering a timestamped entry into the records that attests that the secure signing process has occurred.

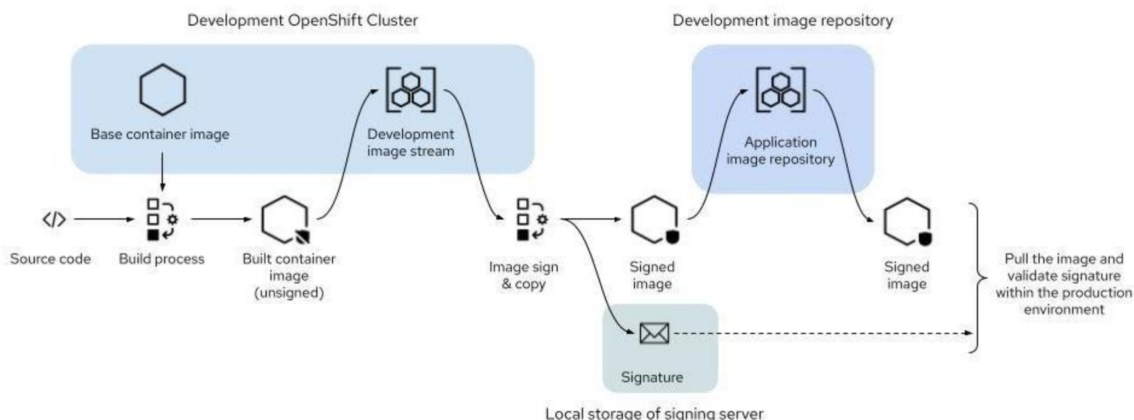


Figure 17: Image and signature creation[13]

Verifying

When a software consumer wants to verify the software's signature, Sigstore compares a tuple of signature, key/certificate, and artifact from the timestamped object against the timestamped Rekor entry.

If they match, it confirms that the signature is valid because the user knows that the expected software creator, whose identity was certified when signing, published the software artifact in their possession.

The entry in Rekor's immutable transparency log means that the signer will be monitoring the log for occurrences of their identity and will know if there is an unexpected signing event.

Consuming verified containers

To ensure that only legitimate container images are deployed in our Kubernetes (k8s) cluster, we can utilize Sigstore's policy-controller admission controller. This controller is responsible for enforcing policies that validate the proper signing of images and the presence of verifiable supply-chain metadata. Additionally, the policy-controller resolves the image tags to ensure that the image being executed is identical to the one initially admitted.

By verifying each image against the workflow that created it, the policy controller can validate that the image was signed by a workflow deployed by a specific entity and within a specific GitHub repository. This verification process guarantees that the image has not been tampered with since its creation, providing an added layer of security.

[13] Source: RedHat

4.2.2.1 Policy Controller

The policy controller admission controller[14] can enforce policy on a Kubernetes cluster based on verifiable supply-chain metadata from cosign[15]. It also resolves the image tags to ensure the image being deployed is not different from when it was admitted.

By default, the policy controller admission controller will only validate resources in namespaces that have chosen to opt-in. This can be done by adding the label `policy.sigstore.dev/include: "true"` to the namespace resource.

An image is admitted after it has been validated against all `ClusterImagePolicy` that matched the digest of the image and that there was at least one passing authority in each of the matched `ClusterImagePolicy`. So each `ClusterImagePolicy` that matches is AND for admission, and within each `ClusterImagePolicy` authorities are OR.

In addition, the policy controller offers a configurable behaviour defining whether to allow, deny or warn whenever an image does not match a policy. This behaviour can be configured using the config-policy-controller ConfigMap created under the release namespace (by default cosign-system), and by adding an entry with the property `no-match-policy` and its value `warn|allow|deny`. By default, any image that does not match a policy is rejected whenever no-match-policy is not configured in the ConfigMap.

4.2.2.2 ImagePolicyWebhook

The ImagePolicyWebhook admission controller is an alternative method to statically attest images pulled to the specific node. It is a Kubernetes feature that allows us to enforce policies for image verification at runtime by calling an external webhook that can verify the digital signature of container images.

To use ImagePolicyWebhook, we follow these steps:

- Create a webhook service that can verify the digital signature of container images. The webhook service should be able to receive a request from the ImagePolicyWebhook admission controller and return a response indicating whether the image should be allowed or denied.
- Deploy the webhook service on the cluster.
- Configure the ImagePolicyWebhook admission controller to call the webhook service for image verification. This can be done by adding the ImagePolicyWebhook admission controller to the list of admission controllers in the Kubernetes API server configuration file and specifying the URL for the webhook service.

As mentioned earlier, we use the cosign tool to provide attestable metadata to containers, in order to be used by the ImagePolicy admission controller. Cosign, Image Signing, and

[14] https://docs.sigstore.dev/policy-controller/overview/
[15] https://github.com/sigstore/cosign

ImagePolicy Verification are the three components that make up the Attestation Mechanism for SERRANO.

ImagePolicy Admission Controller

The ImagePolicy admission controller acts as a gatekeeper for deploying container images inside our k8s clusters. It does this by enforcing regulations that specify which container images are permitted to operate and are therefore considered genuine.

When a container image is presented for deployment, the ImagePolicy admission controller validates the image by carrying out the following procedures to ensure that it is authentic[16]:

a. Image Retrieval: The admission controller is responsible for retrieving the container image from the registry or repository that has been defined.
b. Validation of the Signature: The Admission Controller makes use of Cosign in order to validate the embedded cryptographic signature that is contained within the image information. It compares the signature to the associated public key to confirm that the image has not been tampered with and came from a reliable source. This is done to ensure that the signature is valid.
c. Policy Evaluation: The admission controller examines the image in light of previously established policies. These policies may take into consideration aspects such as the image's provenance, the findings of vulnerability scanning, and the requirements for compliance. Our organization's security requirements and industry best practices served as the basis for establishing these rules.
d. Decision Making: After the findings of the signature validation and policy evaluation have been analyzed, the ImagePolicy admission controller will either decide to accept or refuse the deployment of the container image. If the image meets all of the requirements and is able to pass verification, it will be recognized as valid and will be granted permission to run inside of the cluster. In that case, it is rejected, which eliminates any potential threats to security.

We ensure that only trusted and certified container images are deployed in our cluster by using the Cosign software attestation mechanism and combining it with the ImagePolicy admission controller. This strategy improves the safety and integrity of our containerized programs and reduces the likelihood that those applications would run corrupt or modified image files.

[16] https://cloud.redhat.com/blog/signing-and-verifying-container-images

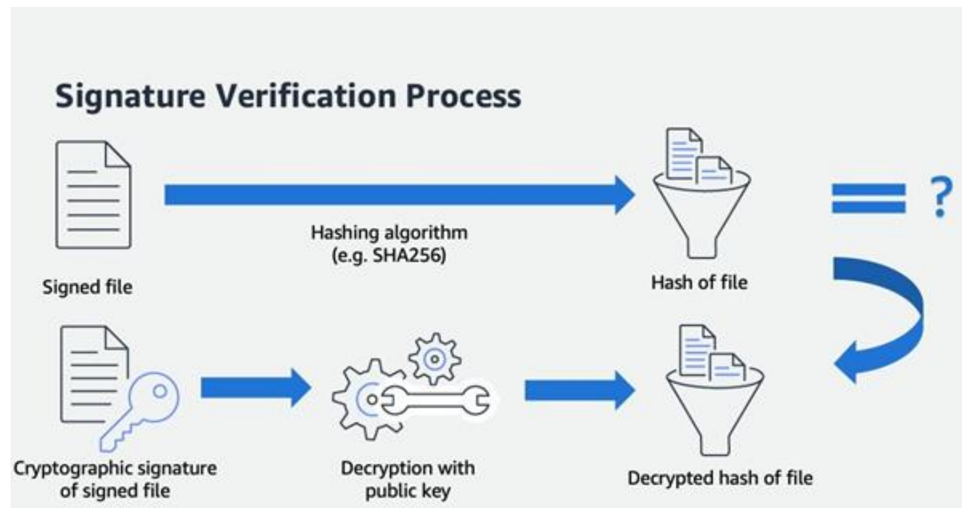


Figure 18: Signature Verification Process[17]

4.3 Isolation Between Tenants in Edge Computing

Edge resources are far more limited than those in the cloud, making it essential to use them efficiently. In this context, the underlying software, which enables the execution of applications at the edge devices, needs to be lightweight and consume as few resources as possible.

The virtualization layer is a vital component of the software system stack in edge computing. Virtualization allows the abstraction of the underlying resources and enables the concurrent execution of workloads from various tenants in an isolated environment. Nonetheless, this comes at the cost of consuming more resources and adding overhead to the overall execution of a workload. Consequently, the virtualization layer must be as lightweight as possible while not compromising the isolation and fair execution among the different tenants.

Virtualization has dominated the cloud. Instead of virtualizing the entire system, containers use Operating System mechanisms to provide the necessary isolation. Such a design requires fewer resources than traditional virtualization since the virtualized environment is much smaller. Furthermore, containers can achieve better performance, especially in the case of I/O and boot times, since the applications can directly communicate with the host Operating System without the mediation of any other software (e.g., hypervisor). On the other hand, relying on pure software solutions by sharing the Operating System among different tenants raises concerns regarding the level of isolation that containers provide. To this end, several recent studies have proved that container isolation is much weaker than traditional virtualization techniques. As a result, container deployment usually occurs inside virtual machines, increasing the overall system software stack.

Under these circumstances, traditional system-level virtualization is the only feasible solution in order to provide strong isolation. In system-level virtualization, a Virtual Machine Monitor

[17] Image Source: AWS

(VMM) creates an entire virtual machine, and a different Operating System runs inside. In most cases, the virtual machine monitor is a user space application that interacts with the host Operating System to create and manage the virtual machines. As a result, researchers and engineers focus on reducing the overhead that the virtual machine monitor induces and optimizing the I/O performance.

In that context, the concept of microVMs has emerged. Instead of using an entire Operating System inside a virtual machine, microVMs use a minimal kernel and only the necessary components to execute applications. The lightweight virtual machines are much more scalable since they can quickly boot and shut down while reducing resource consumption. Such virtual machines also require fewer functionalities from the underlying hypervisor. As a result, new hypervisors can only support the necessary functionalities, specifically for microVMs, reducing their codebase and the overhead of setting up the environment for the virtual machine.

VMM back-and-forth with the host OS can be expensive and redundant, but in some VMM designs, emulation of I/O devices makes it necessary. However, especially in the cloud, the VMM and the VMs mostly use virtual devices for I/O. In that context, the I/O request could be handled directly by the host OS without VMM mediation. Vhost follows such an approach and allows VMM to offload the data plane to another component, which could run inside the host OS. Vhost manages to improve the overall I/O performance significantly. Nonetheless, with Vhost, the guest-host communication operates asynchronously, requiring a thread to poll for the latest data. Using threads for polling might be fine on high-end servers with multicore CPUs, but it can create issues for edge devices with limited cores. Thus, despite the benefits of Vhost, such technology only applies to some edge devices.

To minimize the exposed privileged operations when executing workloads at edge environments, we introduce a sandbox mechanism. Specifically, we protect the host system from any workload running on it by (a) executing it in a contained environment; (b) reducing the exposed privileged operations to the absolute minimum required for the workload to run.

Recent works have introduced a new type of resource virtualization[18,19,20,21,22]. In the context of serverless computing[23], a sandbox mechanism is one of the ways to allow multi-tenancy execution, increasing the workload consolidation factor and reducing idle resources in a cloud

[18] Solo5, "The Solo5 Unikernel," [Online]. Available: https://github.com/solo5/solo5.

[19] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Sscott, B. Singh T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in ASPLOS, 2013.

[20] A. Kantee and J. Cormack, "Rump Kernels: No OS? No Problem!," login Usenix Mag, 2014.

[21] D. Williams and R. Koller, "Unikernel monitors: extending minimalism outside of the box.," in 8th USENIX Conference on Hot Topics in Cloud Computing, 2016.

[22] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," in 17th USENIX Symposium on Networked Systems Design and Implementation , 2020.

[23] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Popa, I. Stoica and D. Patterson, "Cloud Programming Simplified: A Berkeley View on Serverless Computing".

environment. Kata Containers[24] provide such a sandbox mechanism, allowing an OCI compatible container to run inside a traditional VM.

To achieve (a), we use sandboxing mechanisms, mainly containerization and virtualization techniques. In order to facilitate workload deployment, we keep the container concept, but instead of running workloads as containers on the host, we isolate the container execution using VMs. Additionally, we utilise hardware extensions when available.

To achieve (b), we use unikernels[19]. In the last years, a new approach in lightweight virtualization aims to bridge the best from both containers and virtual machines. Unikernels are specialized single-address space machine images constructed by using library operating systems. Some of their advantages include fast boot times, low memory footprint, and increased performance while providing stronger security and hardware isolation. However, unikernels come with many limitations, and running existing applications on top of them takes a lot of work. While some frameworks try to provide a POSIX-like environment, others prefer a clean state approach, requiring the complete refactoring of an application to be able to execute on them.

Following the same design principles, Solo5[18] is a specialized monitor for unikernels. Solo5 is an abstraction layer, permitting the same unikernel or application to be deployed in different environments (KVM, process, sandbox) without any modifications. One of the supported execution environments of Solo5 is a virtual machine running over KVM. In that scenario, Solo5 works like QEMU in a typical QEMU/KVM setup. It only provides a very minimalistic hypercall Application Binary Interface (ABI), which the guest can use for I/O requests. In more details Solo5 provides five hypercalls related to I/O:

- Read from network
- Write to network
- Read from block device
- Write to block device
- Poll

Virtual Machine Monitor

The case of Solo5 demonstrates how simple and minimal a hypervisor can be, highlighting an interesting aspect of I/O in hardware virtualization. When a privileged operation (like a network I/O request) occurs in the guest, the system traps (VMExit) in the host kernel (KVM), and then it is delivered back to the userspace monitor. In its turn, the monitor handles the request from the guest and asks KVM to resume the guest execution. While this path seems appropriate in the typical case (such as with general-purpose hypervisors, where different architectures or devices are emulated, in the case of lightweight virtualization, an additional, unnecessary switch from kernel space to user space incurs significant overhead. For instance, during a network I/O request, the host kernel will return the control to the user space monitor

[24] Redhat and IBM, "Kata containers," [Online]. Available: https://katacontainers.io

in order to handle the guest's request, and the user space monitor will eventually make a system call to transmit or receive the network packet, returning the control to the host kernel. We want to explore how significant the overhead of these mode switches is and find solutions that can substantially reduce the overhead.

To answer the above questions, we designed and implemented HEDGE (Figure 19), a minimal and simplistic VMM that resides inside the Linux kernel interacting directly with KVM without any intervention from the user space. HEDGE is essentially a simple dispatch handler in the kernel that services a guest's needs. It provides an interface to the KVM API, a Virtual Machine execution environment for each of the VMs spawned, generic device handling (network & block), and a management layer to perform basic VM operations (create, destroy, dump console, etc.).

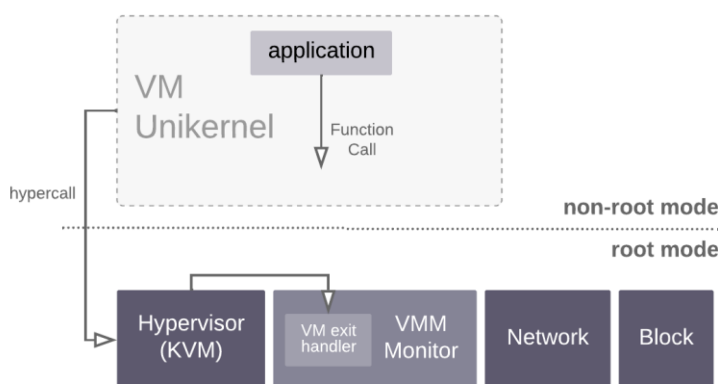


Figure 19: A unikernel running as a VM on HEDGE

A major challenge in this approach is that KVM targets user space processes providing an API through file descriptors. Moreover, using KVM's API from inside the kernel is impossible because most needed functions are only used inside KVM. A way around this is to create a glue code, which actually is some wrappers of KVM functions, between HEDGE and KVM, exposing all the needed functionality. For that reason, two small patches are required in order to be able to use HEDGE. In all other cases, HEDGE works similarly to most user space VMMs. As in the case of QEMU/KVM, each VM is associated with one kernel thread, which implements the vCPU. The thread's life cycle begins when the HEDGE receives a request to spawn a new VM and handles all privileged operations (VMExits). A worth noting design choice that we made is that the new kernel thread will have its memory mappings (mm struct). Moreover, HEDGE allocates a virtual memory that will serve as the guest's memory and maps it to a virtual address of the newly created kernel thread's memory area. Thereby kernel thread mimics a user space process tricking KVM that it gets used from user space.

An essential aspect of HEDGE's design is reducing the noise VMMs enforce to handle I/O requests. Performance is one of the main goals of SERRANO, and in order to achieve that, the guest needs to run uninterrupted as much as possible. Besides removing the mode switch overhead, HEDGE handles I/O requests with the minimum possible overhead. The simple and minimal hypercall ABI from Solo5 helps in that direction. Network packages are formed from

the guest, and when the I/O request occurs, the job of HEDGE is as simple as forwarding the frame to the appropriate network interface. Receiving packages follows the opposite route. Every guest is associated with a virtual interface (TAP), and we use raw ethernet sockets to receive and send network packets on behalf of the guest. Regarding block device support, HEDGE leverages the device mapper (DM) functionality to create a virtual block device mapped to a physical device. Using the block read/write hypercalls from Solo5 ABI, the guest makes I/O requests translated to read/write calls in the kernel to the DM block device. However, the plan is to add support for VirtIO to host more unikernel frameworks and even the basic functionality of a Linux guest.

As with every VMM, HEDGE provides its management interface. For the time being, it is minimal and is able to handle basic VM operations such as start, stop, etc. One can easily manage HEDGE locally (user space) or remotely. In that manner, HEDGE can be easily managed in cases where user space access is impossible, such as edge nodes. In both cases, HEDGE can be managed by the following commands:

- *Load*: loads a module (VM image) and prepares its deployment.
- *Start*: Executes the selected module.
- *Stop*: Stops the execution of a VM.

Moreover, a user can select which block or net device will be used, specify the command line arguments for the guest, and dump the guest's console output. Furthermore, a user can also access statistics such as boot and setup times, I/O operations (both disk and network), and generic stats regarding HEDGE, such as the number of VMs, memory consumption, and more. Someone can interact with the management interface locally via a specialized filesystem in the Linux kernel, *procfs*. When HEDGE is loaded, two new files and one directory is created under */proc* directory:

- */proc/monitor*: I/O file that can be used to control the hypervisor and its virtual machines.
- */proc/vmcons*: I/O file that keeps the virtual machine’s output.
- */proc/vmstats*: A directory that keeps stats for the hypervisor and virtual machines.

On the other hand, one can interact with the network management interface. In that case the commands are sent over UDP, while the files can be transmitted over *tftp.*

In the context of the SERRANO platform, we extract specific application code from the use cases and port it to two popular unikernel frameworks: Unikraft[25] and rumprun[20].

[25] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu and F. Huici, "Unikraft: fast, specialized unikernels the easy way," in Sixteenth European Conference on Computer Systems EuroSys '21, New York, NY, USA, 2021.

4.4 Integration of Security Mechanisms

In SERRANO, we build on the confidential computing paradigm to provide end-to-end secure tiers. During the initial design and provisioning phases, we have identified the following security levels that comprise the SERRANO secure infrastructure layer:

- ***Tier-0:*** No additional security, trustiness or enhanced isolation, execution through default containers.
- ***Tier-1:*** More isolated execution environment but no advanced security or trustiness, execution through containers in micro-VMs (sandboxing).
- ***Tier-2:*** More secure execution, better isolation but no advanced trustiness, execution through unikernels that reduce attack surface and provide ultra-fast boot.
- ***Tier-3:*** Advanced security and trustiness, default isolation*,* execution through container with secure boot and trusted execution extensions.
- ***Tier-4:*** Maximum security, trustiness and isolation, execution though container with secure boot and trusted execution extensions and within a sandboxed micro-VM.

Table 2 summarizes the provided functionality, the different security and trust levels that each deploy method provides and their trade-offs.

Table 2: Security Tiers for the SERRANO platform

	Tier-0	Tier-1	Tier-2	Tier-3	Tier-4
Isolation	minimal	Yes	Yes	Yes	Maximum
Encryption	No	No	No	Could have	Yes
Trusted Execution	No	No	No	Yes	Yes
CPU/MEM Footprint	Low	Medium	Low	Low	Medium
Spawn Time	Fast	Fair	Ultra-fast	Fast	Fair
Specialized software	No	Yes	Yes	Yes	Yes
Specialized hardware	No	No	No	Yes	Yes

Tier-0 refers to generic containers (this is the baseline). All other columns are nomalized to Tier-0. Tier-1 refers to microVM sandboxing[26,27]. The application (essentially the container) is executed on top of a microVM, which entails booting a full virtualization stack (VMM, kernel, rootfs) and keeping it active until the application terminates. Although there have been advances in reducing the overhead of VMMs in terms of CPU and memory footprint, even state-of-the-art VMMs[27] exhibit a 30% overhead for memory handling and address translation, as well as extra CPU time for handling I/O and context/mode switches. This includes both the extra memory that the VMM occupies and the fact that to spawn a container, a full OS system (the microVM) must be present and active. The storage overhead

[26] The speed of containers, the security of VMs, https://katacontainers.io/
[27] A. Agache, M Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.M. Popa, "Firecracker: lightweight virtualization for serverless applications", (NSDI'20), 2020

is proportional to the application; however, with bare-minimum rootfs (typically 10s of MBs), a microVM can support container execution (typically 100s of MBs).

Tier-2 refers to unikernel execution. CPU, memory and storage footprint is minimized, as the application itself is compiled as a machine image, removing any unwanted OS and library software stacks. The work[28] depicts at least 20% reduction in CPU and Memory overhead, whereas the application binary footprint is reduced by at least 60% since, apart from the optimized build, no OS/libraries is included in the application. Tier-3 and Tier-4 are equivalent to Tier-0 and Tier-1, with the added security of secure boot[29]. Trusted execution is realized using an attestation mechanism in the hypervisor layer (Tier-4), only affecting initial boot times.

[28] S. Kuenzer, V. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, “Unikraft: fast, specialized unikernels the easy way”, EuroSys '21, https://doi.org/10.1145/3447786.3456248

[29] M. Sabt, M. Achemlal and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not," 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 2015, pp. 57-64, doi: 10.1109/Trustcom.2015.357.

5 Conclusions

This deliverable (D3.4) presents the outcomes of Work Package 3 "Hardware and Software Platforms for Enhanced Security" of the SERRANO project, which comprises the work by five partners and an investment of 65 PMs. Each task of the three tasks has developed different aspects of the SERRANO platform: the storage system in itself, a scalable secure storage system, and workload isolation of processes. This deliverable owes to be read in conjunction with the previous three deliverables within this WP: D3.1 Accelerated encrypted storage architecture (M15), D3.2 Secure cloud storage system (M15), and D3.3 Trust and isolated execution on untrusted physical tenders (M15). The deliverable provides an overview of the final release of the SERRANO secure infrastructure layer and describes the developments in WP3 during the second iteration of the SERRANO implementation plan (M16-M30).

It summarizes the efforts and working system solution developed within this work package, which includes a solution for distributed storage that employs acceleration engines in distributed infrastructures and orchestration tools that rely on trusted execution environments and hardware security mechanisms.

The SERRANO-enhanced secure storage service encompasses geographically diverse storage facilities with multiple cloud and edge storage locations. The edge storage is based on the SERRANO developments that ensure privacy and security by design. In addition, the service leverages the acceleration capabilities of modern network interface cards that include the data encryption accelerators developed in SERRANO to improve the user experience by reducing end-to-end latency as well as freeing up local processing resources, which allows the network and storage fabric to scale up graciously. The implemented solution also includes the appropriate backend mechanisms, public secure and storage APIs, and a developer portal. The deliverable also provides different performance tests and optimizations introduced compared to the initial version. These optimizations include reducing the number of required edge-cloud HTTP calls (edge-cloud continuum) and file caching mechanisms. The report also provides visibility on S3 authentication, pre-signer URLs policies, and random linear network coding techniques.

Moreover, the deliverable describes the final SERRANO developments for providing trust and isolation execution on untrusted physical tenders. The provided solutions include software mechanisms and frameworks that enable the isolation of processes in the form of SW and at the HW layer. To this end, SERRANO built on the confidential computing paradigm to provide end-to-end secure tiers along with the required mechanisms for their integration in the overall SERRANO platform.