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

Grant Agreement no. 101017168

Deliverable D3.3
Trust and isolated execution on untrusted physical tenders

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:	31/03/2022
Actual submission date:	31/03/2022

Dissemination level:	Public
Revision:	V1, 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
01.11.21	MLNX	Draft	0.1	Initial ToC
31.01.22	NBFC	Draft	0.2	Structure & ToC
17.02.22	NBFC	Draft	0.3	Initial Content
23.02.22	NBFC	Draft	0.4	Confidential Computing, Secure & Measured boot explained
01.03.2022	CC	Draft	0.5	UC1 and attack scenarios description
07.03.22	MLNX	Draft	0.6	Smart NICS and Data Processing Units
08.03.22	NBFC	Draft	0.7	Merged input, minor updates & polish figures
10.03.2022	IDEKO	Draft	0.8	UC3 and attack scenarios description
20.03.2022	NBFC	Draft	0.9	Merged input, added confidential computing content & updated various sections
25.03.2022	NBFC	Draft	0.95	Incorporated AUTH author list, Added extra input/content, Finalize Intro, Conclusions, Polished References
30.03.2022	NBFC	Final	1	Reworked the whole text, addressing the review comments and suggestions.

Author List

Organization	Author
MLNX	Yoray Zack, Juan Jose Vegas Olmos
CC	Marton Sipos
AUTH	Argyrios Kokkinis, Dimosthenis Masouros, Aggelos Ferikoglou, Dimitris Danopoulos, Ioannis Oroutzoglou, Kostas Siozios
INB	Ferad Zyulkyarov
IDEKO	Aitor Fernández, Javier Martín
NBFC	Anastasios Nanos, Charalampos Mainas, Georgios Ntoutsos, Ilias Apalodimas

Internal Reviewers

IDEKO (Javier Martin)

HLRS (Kamil Tokmakov)

Abstract: This deliverable (D3.3) presents the outcomes of *Task 3.3 "Workload Isolation and Trusted Execution"*, *Work Package 3 "**Hardware and Software Platforms for Enhanced Security**"* of the SERRANO project, during the first iteration of the incremental implementation plan. The deliverable gives an overview of the hardware description of the SERRANO edge platform nodes, the attack vectors of the SERRANO use cases, and presents the rationale of the security design for the SERRANO platform. Additionally, D3.3 describes the systems software stack that enables secure and trusted execution in the SERRANO platform, focusing on the platform components (hardware and software), the OS and orchestration layers. Finally, D3.3 introduces the initial design of the proposed SERRANO secure infrastructure layer.

Keywords: SERRANO platform, transparent deployment, secure boot, trusted execution, confidential computing

Table of Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
AES	Advanced Encryption Standard
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BSI	Belief Desire Intention
CoT	Chain of Trust
DOCA	Data center On a Chip Architecture
DPU	Data Processing Unit
EDE	Event Detection Engine
GCM	Galois/Counter Mode
GDPR	General Data Protection Regulation
GPU	Graphics Processing Unit
HLRS	High Performance Computing Center Stuttgart
HPC	High Performance Computing
IO	Input/Output
IMA	Integrity Measurement Architecture
ML	Machine Learning
OCI	Open Container Initiative
OSS	Object Storage Server
OST	Object Storage Target
PCIe	Peripheral Component Interconnect Express
PCR	Platform Configuration Register
PEF	Protected Execution Family
PXE	Preboot Execution Environment
QoS	Quality-of-Service
RDMA	Remote Direct Memory Access
RLNC	Random Linear Network Coding
RoCE	RDMA over Converged Ethernet
RTL	Register-Transfer Level
S3	Simple Storage Service
SDK	Software Development Kit
SDN	Software-Defined Network
SEV	Secure Encrypted Virtualization
SoC	System on a Chip
SW	Software
TDX	Trust Domain Extension
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module
UC	Use Case
VM	Virtual Machine
VMM	Virtual Machine Monitor

1 Executive Summary

SERRANO envisages the development and deployment of disaggregated federated cloud infrastructures that operate, process and store in the edge, enabling accelerated edge nodes as integral parts of the computation and storage chain.

This document presents the SERRANO deliverable D3.3: “Trust and isolated execution on untrusted physical tenders”, focusing on the initial design of the SERRANO secure infrastructure layer.

In this deliverable, we first briefly present the SERRANO architecture, arguing about the various modes of execution in the context of the integrated platform, as well as the use cases and their possible attack scenarios. Afterwards, we describe our work on the systems software stack to mitigate the risks of potential malicious users and data leaks. Finally, we conclude with the first version of the SERRANO secure infrastructure layer.

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 will automate the process of application deployment across the 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 [1].

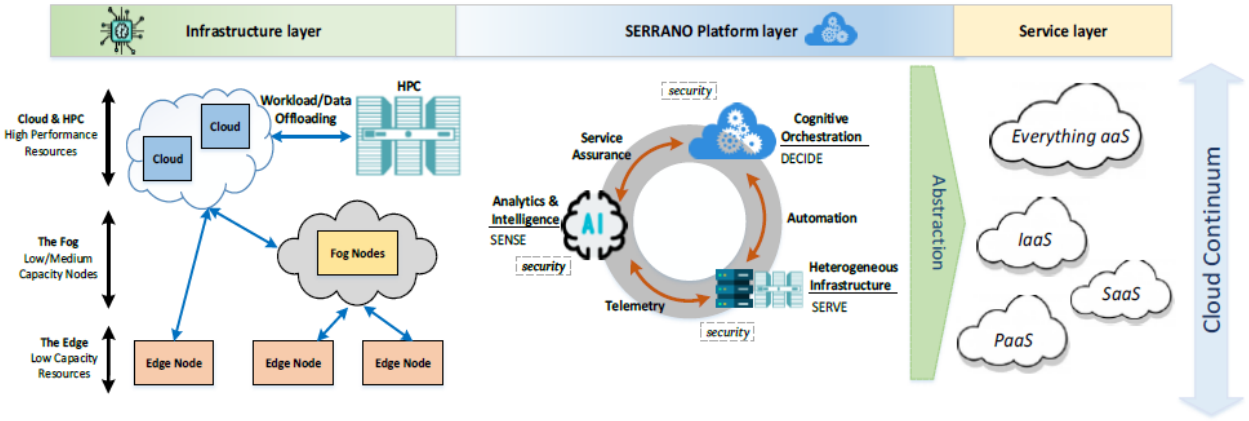


Figure 1: The SERRANO platform, utilizing edge, cloud and HPC resources and empowering the Everything as a Service (EaaS) notion towards the cloud continuum

This document presents the motivation behind the development of an end-to-end secure layer for application deployment in the Cloud-Edge continuum, focusing on the attack scenarios that the project's use cases are vulnerable to, as well as the technical components needed to protect the applications from malicious third-parties.

First, we present the rationale of a secure infrastructure layer, briefly describing the hardware and software components of the SERRANO platform, with a particular focus on the secure aspects. Then, we briefly present the Use-cases and their respective attack scenarios.

We then present the initial design of the systems software stack, focusing on the hardware and software design choices. We briefly describe the current status of the implementation, as well as the plan for further development. Specifically, we introduce the concepts of Secure and measured boot, a crucial part of the confidential computing concept. Moreover, we describe the way we reduce the attack surface of the applications running on the SERRANO platform: we introduce KVMM, an in-kernel Virtual Machine Monitor, tailored to short-lived, purpose-based workloads. We briefly describe the basic design principles, as well as the management interface (API) of KVMM.

2.1 Purpose of this document

The present deliverable (D3.3) presents the outcomes of *Task 3.3 "Workload Isolation and Trusted Execution"*, *Work Package 3 "**Hardware and Software Platforms for Enhanced Security**"* of the SERRANO project, during the first iteration of the incremental implementation plan. T3.3 is associated with the design and implementation of the secure boot and trusted execution mechanisms enabling workload isolation in the SERRANO platform.

The objective of D3.3 is to capture the initial design and implementation of the components that comprise the SERRANO Secure Infrastructure Layer.

D3.3 will serve as the basis for the SERRANO platform security features design and implementation. Finally, updated information regarding secure boot and trusted execution, as well as the final implementation description will be available at D3.4: Final release of SERRANO Secure Infrastructure Layer (M30).

2.2 Document structure

The present deliverable is split into 2 major chapters:

- Workload isolation and Trusted execution: this chapter describes two of SERRANO edge hardware nodes, the SERRANO use cases as well as possible attack vectors.
- Systems software stack: this chapter presents the software and hardware components that we develop, integrate, and use in the SERRANO project to address the attack scenarios presented in the previous chapter.

2.3 Audience

This document is publicly available and should be of use to anyone interested in the SERRANO Secure Infrastructure Layer and its individual components.

3 Workload Isolation and Trusted Execution

Edge computing brings memory and computing power closer to the location where it is needed. In edge computing systems, computation is rather offloaded to nearby resources than to the cloud, due to latency reasons. However, the performance demand in the edge grows steadily, which makes nearby resources insufficient for many applications. Additionally, the number of parallel tasks at the edge increases, based on trends like machine learning, Internet of Things, and artificial intelligence. This introduces a trade-off between the performance of the cloud and the communication latency at the Edge. Furthermore, the need to be energy efficient in a mobile environment is high. The Edge computing paradigm, nowadays, employs single-tenant tasks, scheduled in resource-constrained devices, needing specialized Operating Systems (OSes) to host these tasks.

To be able to move to a multi-tenancy execution model at the Edge, the system must ensure non-interference and controlled data access among different and possibly concurrently running applications. To this end, virtualization plays a significant role in adding secure multi-tenancy execution at the Edge. Adding another layer of abstraction facilitates unified execution frameworks but complicates the execution stack and consumes resources for ensuring correct management, isolation and resource sharing among tenant workloads.

To alleviate the significant overhead of adding a full virtualization stack (hypervisor & Virtual Machines) at the Edge, the research community has proposed the use of container technology. However, this execution model increases the attack surface, as it exposes the full OS and runtime layer to any malicious or compromised application running in a container. Related works and critical security advisories [2] have pointed out that containers are far too insecure for multi-tenancy. Recent works [3] have introduced a new type of resource virtualization, bringing the benefits of isolation and secure execution, without the burden of a full virtualization stack to support generic Operating Systems.

3.1 SERRANO Platform

The SERRANO platform integrates novel hardware and software technologies and methodologies towards an application-optimized and secure service instantiation. The SERRANO platform integrates various diverse hardware resources, ranging from powerful cloud computing nodes, to low-end Edge devices. In the following sections we briefly describe the Edge nodes, mainly susceptible to security attacks by third-parties.

3.1.1 Smart NICS and Data Processing Units

NVIDIA Mellanox ConnectX® SmartNICs utilize stateless offload engines, overlay networks, and native hardware support for RoCE[1] and GPUDirect™[2] technologies to maximize application performance and data center efficiency. Developers can use ConnectX custom

[1] https://docs.nvidia.com/networking/pages/viewpage.action?pageId=19816309
[2] https://developer.nvidia.com/gpudirect

packet processing technologies to accelerate server-based networking functions and offload datapath processing for compute-intensive workloads including network virtualization, security, and storage functionalities.

The NVIDIA® BlueField®-2[3] data processing unit (DPU), shown in Figure 2, is the world's first data center infrastructure-on-a-chip optimized for traditional enterprises' modern cloud workloads and high-performance computing. It delivers a broad set of accelerated software defined networking, storage, security, and management services with the ability to offload, accelerate and isolate data center infrastructure. With its 200Gb/s Ethernet or InfiniBand connectivity, the BlueField-2 DPU enables organizations to transform their IT infrastructures into state-of-the-art data centers that are accelerated, fully programmable, and armed with "zero trust" security to prevent data breaches and cyber-attacks. By combining the industry-leading NVIDIA ConnectX®-6 Dx network adapter with an array of Arm® cores and infrastructure-specific offloads, BlueField-2 offers purpose built, hardware-acceleration engines with full software programmability. Sitting at the edge of every server, BlueField-2 empowers agile, secured and high-performance cloud and artificial intelligence (AI) workloads, all while reducing the total cost of ownership and increasing data center efficiency. The NVIDIA DOCA™ software framework enables developers to rapidly create applications and services for the BlueField-2 DPU. NVIDIA DOCA makes it easy to leverage DPU hardware accelerators, providing breakthrough data center performance, efficiency, and security.

Figure 2: BlueField2 DPU: 8 ARM A72 CPUs, 8MB L2 cache, 6MB L3 cache in 4 Tiles, ARM Frequency: 2.0-2.5GHz, Dual 10-100Gb/s Ethernet & InfiniBand single 200Gb/s, PCIe gen4.

3.1.2 ML-enabled edge nodes

A powerful device for the Edge, with low-power footprint is the NVIDIA Jetson Nano [4] development board (Figure 3). These are small but powerful computers that are able to run multiple neural networks in parallel for various applications that involve Machine Learning techniques or algorithms [5].

[3] https://www.nvidia.com/en-us/networking/products/data-processing-unit/

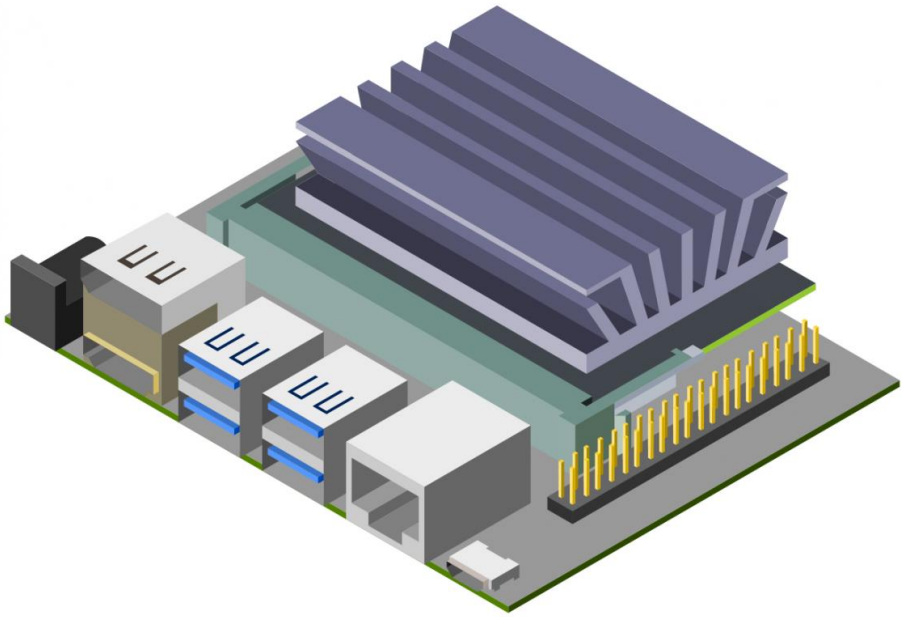


Figure 3: NVIDIA Jetson Nano development kit

The Jetson Nano is an AI development kit offering high-performance and power-efficient computing. It's built on the same architecture and software that powers the world's fastest supercomputers. It also features NVIDIA's JetPack SDK [6], which is built on CUDA-X and is a complete AI software stack with accelerated libraries for deep learning, computer vision, computer graphics and multimedia processing to support the Jetson Nano.

The compact yet powerful Jetson Nano platform delivers 472 Gflops of computing performance, consuming as little as 5 watts, making it the perfect choice for object detection, video search, face recognition and heat mapping. It supports high-resolution sensors, can process multiple sensors in parallel and has modern neural networks on each sensor stream. It complements many popular AI frameworks, so it's easy for developers to integrate into their products.

3.2 Use case description

3.2.1 Use case 1: Secure Storage

The project's first use case focuses on providing a secure, high-performance file storage solution. It builds on the multi-cloud technology adopted by SkyFlok [7], a commercially available file storage and sharing solution from Chocolate Cloud. The use case expands the purely cloud-based architecture to the edge of the network, to the customers' existing IT infrastructure. By leveraging edge storage locations, the latency and throughput of storage operations can be significantly improved. The inclusion of both cloud and edge locations makes it possible to better tailor the storage service to the requirements of enterprise users. To make integration with existing software simple, the use case will feature an S3-compatible storage API.

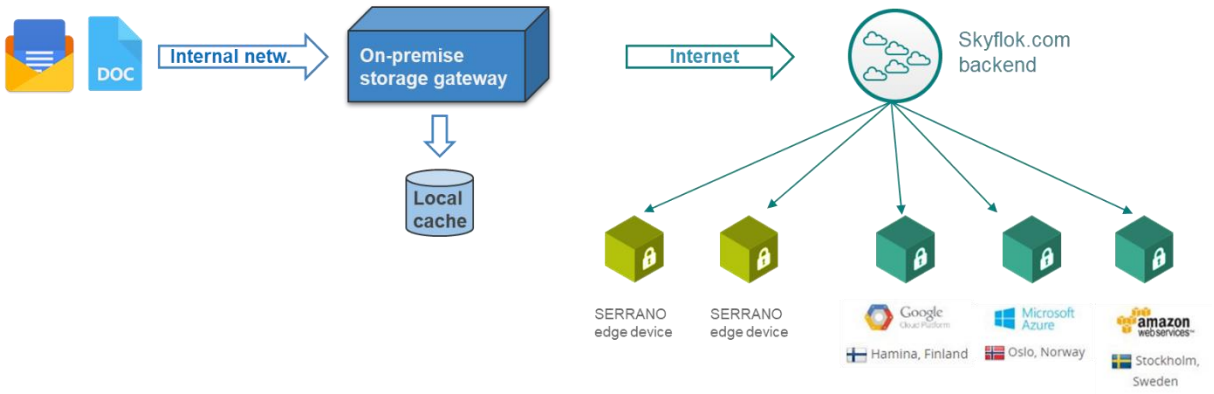


Figure 4: The components of the SERRANO-enhanced Storage Service

The use case relies on the SERRANO-enhanced Storage Service, developed in Task 3.2. Figure 4 shows the service's components and the ways in which they interact to provide the S3-compatible API for file-based storage operations. The users of the SERRANO platform only interact directly with the On-premise storage gateway. It is the role of the gateway to access the different storage locations. These include cloud-based object stores as well as SERRANO edge devices. Finally, the gateway also coordinates with the Skyflok.com backend as needed for authentication, authorization, file metadata storage and other, mostly non-storage-related functions.

3.2.2 Use case 2: Fintech Analysis

The deployment of the FinTech use case for portfolio optimisation is displayed in Figure 5. It consists of deployment on on-premise servers as well as cloud servers. The entire deployment will be managed by the SERRANO cloud orchestration and management components of the platform. The on-premise servers will execute functions that are not containerized, for example, obtain market data, create investment profiles, portfolio rebalancing and order operations. Multiple instances of microservices will be deployed through containers into SERRANO-enhanced nodes. Some of these microservices will use the available accelerators on the node that are deployed. For example, market analysis, forecasting and backtesting would benefit from the available acceleration on a particular SERRANO-enhanced node. Other microservices such as for investment strategies, cannot benefit from any hardware acceleration and may be deployed on cloud nodes that do not have accelerators.

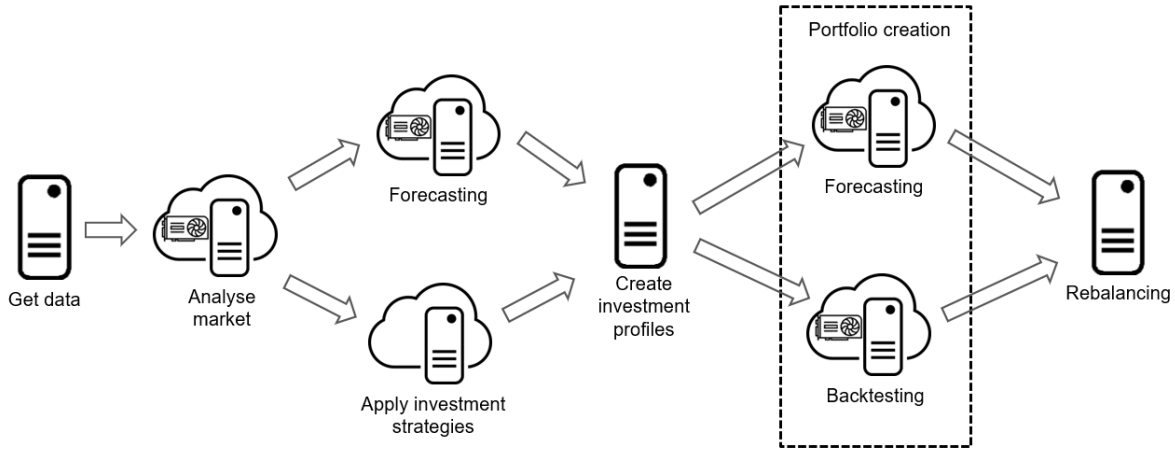


Figure 5: FinTech portfolio optimisation use case deployment overview

3.2.3 Use case 3: Anomaly detection in manufacturing setting

Companies that manufacture expensive parts with high added value are very demanding in terms of machine availability and quality assurance. Predictive maintenance, evaluation of remaining useful life, and diagnosis of critical machine elements are state-of-the-art practices. However, some of the techniques used require the machine to stop before performing the analysis. As a result, the various hardware components are idle most of the time, waiting for analysis procedures to begin, something the manufacturing industries are eager to avoid.

Downtime for faulty devices in an industrial plant must be kept to a minimum to achieve high system availability. In the competitive world of manufacturing, getting the most out of your machine can be the difference between being competitive or not. Therefore, and mainly after the emergence of the Industry 4.0 paradigm, many techniques and methods are being widely applied to meet a simple but complex objective: to keep the machine running most of the time.

This use case aims at developing a system able to detect machine's ball-screws anomalies, by processing the amount of data generated in real-time by high-frequency sensors.

Figure 6: Anomaly detection in manufacturing setting

This use case will leverage SERRANO platform to change the state-of-the-art approach and to perform the ball-screws status assessment without stopping the machine. The challenges faced by this this approach are:

- High data volume to be processed at the edge
- High data velocity
- Performing anomaly detection over a non-controlled machine

3.3 Attack scenarios

A software supply chain attack [8] 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 different ways, such as via compromise of the source code repository, theft of signing keys, or penetration of distribution sites and channels. As a part of an authorized and normal distribution channel, customers unknowingly acquire and deploy these compromised components on to their systems and networks. 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. This is a threat that results from inheriting risk decisions made by a supplier within the supply chain that impacts the end user of the final product or service. The deeper into the supply chain it occurs, the more difficult it is to identify in advance. This inserted vulnerability may be used by the malicious actor as a part of a larger attack chain that uses the malicious code to gain access within the core network of the telecom and pivot towards other attack vectors.

Additionally, unauthorized access to software or network components provides a malicious actor with the opportunity to modify configurations to reduce security controls, install malware on the system, or identify weaknesses in the product. These vulnerabilities could be exploited for increased persistent and privileged access within a system or network. Malicious actors may also embed code in SDN controller applications to constrict bandwidth and negatively affect operations.

For instance, a potential attack vector stems from 'persistent threats', where malware is inserted into a system in a way 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 two technologies: Secure Boot and Measured Boot, in order to provide assurance that when a platform boots, it is running code that hasn't been compromised.

3.3.1 Secure storage

Data privacy and security are critical concerns for almost any IT system's users. This is especially true for enterprises, where many times the nature of the data means its protection is key to complying with laws and regulations. For example, GDPR regulations explicitly and precisely stipulate how private data must be handled. While cloud storage has seen a steady increase in its adoption rate, concerns over its security are still present. Cloud providers or the user accounts of cloud services may be hacked. Furthermore, some cloud providers explicitly state that they scan files uploaded to their service[4]. 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.

The Secure Storage use case presents a hybrid system that includes both cloud-based and on-premise 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 the use of HTTPS where the TLS protocol provides the protection through AES

[4]Is Google Drive secure? : https://support.google.com/drive/answer/141702?hl=en&ref_topic=2428743

encryption. Furthermore, the storage service employs two further simple design principles to limit the attack surface. First, data does not leave the company's infrastructure without being encrypted. In essence, 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 Skyflok.com backend, thus lowering the attack surface.

Data at rest is protected using AES encryption in GCM block mode with 256-bit keys. This guards against attacks targeting the different storage locations. In the case of more secure cloud-based locations its main purpose is to ensure that the providers themselves cannot read the data. In the case of edge locations, which might not have the same level of physical and digital protection, this is the single most important security measure. The encryption is complemented through the use of a novel erasure code, Random Linear Network Coding (RLNC) [9]. Because RLNC creates linear combinations of the original data using random coefficients, an attacker would need to guess the coefficients correctly to decode the data. Furthermore, the distribution of data to storage locations also has a security-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. 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 itself.

The On-premise storage gateway is in charge of accessing coded fragments at the storage locations. As such, it must be granted both read and write access to both cloud and edge locations. This raises a security concern, because if a malicious actor gains access to the credentials needed to access the locations, it could read, overwrite and delete data. The gateway is located on the customer's premises and is not under the control or supervision of the service's maintainer. This type of attack is mitigated by employing pre-signed URLs[5]. Instead of providing the gateway with credentials, it must request a special, cryptographically signed URL from the Skyflok.com backend every time it wants to access a file's data. The link has an expiration time and is tied to one specific resource.

The metadata and the encryption keys 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.

A more detailed explanation of the aforementioned security-related measures, as well as the entire data processing pipeline can be found in Deliverable 3.2.

[5] Amazon S3, Sharing objects using pre-signed URLs: https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html

3.3.2 Fintech Analysis

Microservice-based deployments, the core compute component of this use case, are susceptible to a number of potential attacks, either by data leaks or by the compromise of specific parts of the execution, such as tampering with the analysis model, create false investment profiles, portfolio rebalancing etc.

3.3.3 Anomaly detection in manufacturing setting

IDEKO empowers the machines with a SmartBox (Danobat Box). The SmarBox is an edge device with cloud connectivity that enables data gathering, an application execution environment and data persistence mechanisms at the edge.

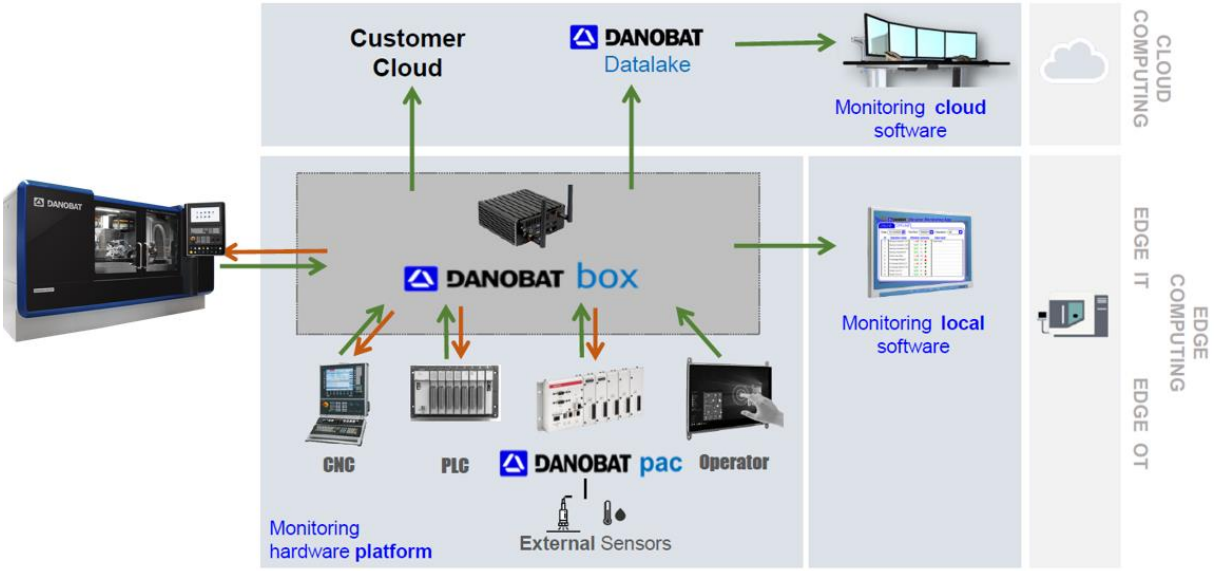


Figure 7: Machine connected to Edge/Cloud computing devices

Data collected from the machine-tools is considered as core knowledge of the production processes. Although this data per-se is complex to trace to the source machine and its underlying process, clients are very thorough with the security measurements around their data.

The objective of an attacker is not only the data collected, but also the physical security of machines. An attack to a production machine may cause severe injuries to the operator and irreparable damage to the equipment. In fact, the VDW (German Association of Machine Tool Builders) has published a guide[6] for companies that helps to increase computer security in machine tools in a simple way.

Usually, most of the attacks to production machines come from the internal security protocols of the company and these attacks are caused by infection with malware via cloud or local network. Many of these attacks are caused by poor network configuration or total lack of network segmentation, lack of firewalls or anti-viruses in operating systems associated with the machines.

[6] https://vdw.de/wp-content/uploads/2021/03/pub_IT-Sicherheit-an-Werkzeugmaschinen_VDW.pdf

Industry 4.0 applications are also a security threat. Every day, more applications are developed and both frontend and backend servers are susceptible to be attacked. Open ports, SQL injection, remote code injection or cross-site scripting are possible attack vectors on poorly configured shopfloors.

Other common attack vectors are by the means of connecting external devices, such as USB devices, when third-party programs are installed or data from other servers is transferred through these devices, that could inject malware and may infect the devices associated with the machine.

4 Systems software stack

In this section we describe two complementary methods used in the SERRANO architecture to reduce the risk of the attack scenarios presented in Section 3. We first focus on the reduction of the attack surface by minimizing the exposure of privileged operations. This part addresses attack vectors related to the exploitation of bugs in software, as well as component misconfiguration. Then, we describe the hardware and software mechanisms used to enable trusted execution as part of the SERRANO architecture. The technical details of the implementation are included in Section 7 (ANNEX A). These mechanisms address attack vectors related to malicious users with physical access to the hardware running the software components.

4.1 Reduced attack surface

To minimize the exposed privileged operations when executing workloads at Edge environments, we introduce the use of 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 [10] [11] [12] [13] [14]. In the context of serverless computing [15], 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 environment. Kata Containers [16] 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 [11]. 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, increased performance while at the same time they provide stronger security and hardware isolation. However, unikernels come with a lot of limitations and running existing applications on top of them is not straightforward. While some frameworks try to provide a POSIX like environment some others prefer a clean state approach, requiring the complete refactor of an application to be able to execute on them.

Following the same design principles, Solo5 [10] is a specialized monitor for unikernels. Solo5 can be seen as 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, i.e. 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 5 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 and it highlights 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 common 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 user space monitor in order to handle the guest's request and the userspace monitor will eventually make a system call to transmit or receive the network packet, returning the control back to the host kernel. We would like to explore how big the overhead of these mode switches is and find solutions which can substantially reduce the overhead.

With the intention to answer the above questions we designed and implemented KVMM, a minimal and simplistic VMM that resides inside the Linux kernel interacting directly with KVM without any intervention from the user space. KVMM is essentially a simple dispatch handler in the kernel that services the needs of a guest. 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.).

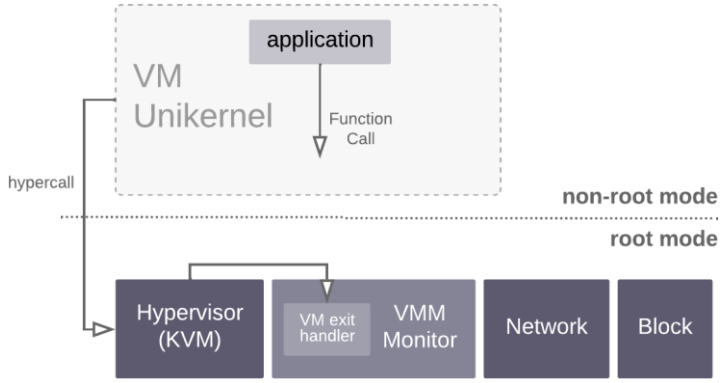


Figure 8: A unikernel running as a VM on KVMM

A major challenge in this approach is that KVM targets user space processes providing an API through file descriptors. Moreover, it is not possible to use KVM's API from inside the kernel because most needed functions are only used inside KVM. A way around is to create a glue code (which is actually some wrappers of KVM functions) between KVMM and KVM exposing all the needed functionality. For that reason, two small patches are required in order to be able to use KVMM. In all other cases KVMM works in a similar way as 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 a request to spawn a new VM is received by the KVMM and handles all privileged operations (VMExits). A worth noting design choice that we made, is that the new kernel thread will have its own memory mappings (mm struct). Moreover, KVMM allocates a virtual memory which will serve as guest's memory and it 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 userspace.

An important aspect of KVMM's design is reducing the noise that VMMs enforce to handle I/O requests. Performance is one of the main goals of this project and in order to achieve that the guest needs to run uninterrupted as much as possible. Apart from removing the mode switch overhead, KVMM 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 KVMM is as simple as forwarding the frame to the appropriate network interface. Receiving packages follow 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 KVMM leverages the device mapper (DM) functionality to create a virtual block device, mapped to a physical device. The guest using the block read/write hypercalls from Solo5 ABI makes I/O requests which are translated to read/write calls in the kernel to the DM block device. However, the plan is to add support for VirtIO, in an effort to host more unikernel frameworks and even a basic functionality of a Linux guest.

As every VMM, KVMM provides its own 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 KVMM from both locally (user space) or remotely. In that manner KVMM can be easily managed in cases where user space access is not possible, such as edge nodes. In both cases KVMM 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 at last dump the console output of the guest. Furthermore, a user is also able to access statistics such as boot and setup times, I/O operations (both disk and network) and generic stats regarding KVMM such as number of VMs, memory consumption and more. Someone can interact with the management interface locally via a specialized filesystem present in the linux kernel, *procfs*. When KVMM 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 which keeps the output of the virtual machine.
- */proc/vmstats*: A directory which keeps stats for 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 [17] and rumprun [12]. Currently, this is work in progress, planned to be finalized after the first release of the SERRANO integrated platform (M18).

4.2 Hardware Trust

A mobile device is susceptible to a number of attacks by malicious users, physical or otherwise. For instance, one could gain access to the storage backend of a mobile device (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 can not be modified at all, or else can not be modified without cryptographic credentials.

For many systems, that "Root of Trust" is provided by the Unified Extended Firmware Interface (UEFI) BIOS[7]; 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 can not 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.

[7] http://www.uefi.org/

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). But 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.

- In a Secure Boot chain, 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 that 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).
- In a Measured Boot chain, we still depend on a Root of Trust as the starting point for a chain of trust. But in this case, prior to 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 in a way 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 of interest.

4.2.1 Secure boot

Secure Boot is relatively self-contained. If the handful of signed objects have not been tampered with, the platform boots, and secure boot 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.

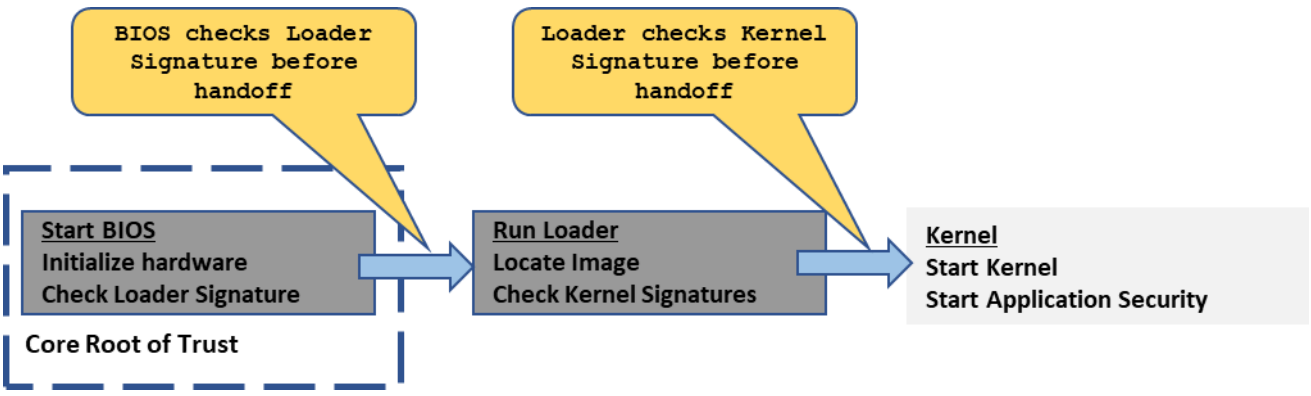


Figure 9: Secure boot execution flow

4.2.2 Measured boot

Measured Boot is more flexible, but also requires an important step. All those hashes have to be stored in a way 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[8]. Of the many functions a TPM can provide, one is the facility called Platform Configuration Registers (PCRs), used for storing hashes.

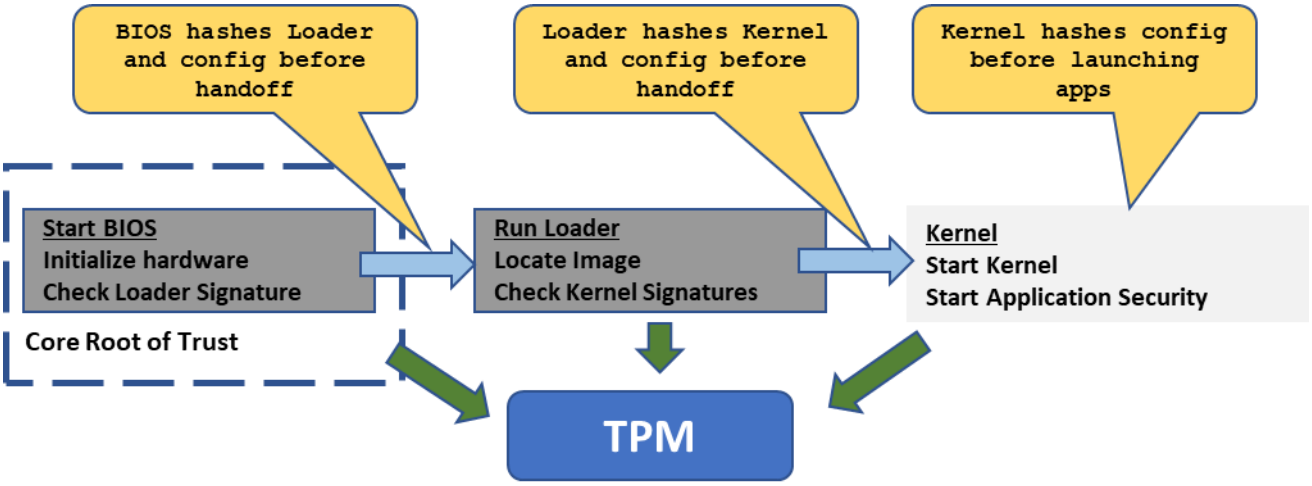


Figure 10: Measured boot execution flow

PCRs are registers in the TPM that are cleared only at hardware reset, and cannot be directly written. What one can do is "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 in a way that unambiguously demonstrates which modules were loaded.

Once the PCRs have been collected, the second step is for the TPM to report the values, signed by a key that 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 both be used at the same time: 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.

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

4.2.3 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. Final PCR value captures the whole history of measurements - useful in quick validation of 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 validation of a change in one of the input sequences

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 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, if those PCR values aren’t the same, the secret cannot be recovered.

Using these two 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

Measured Boot is a method where each of the software layers in the booting sequence of the device, measures the layer above that, 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 Extend operations are recorded by the BIOS and the bootloader, in a special firmware table, called the TPM Eventlog table, and this table is handed over to the operating system during OS takeover. By playing the same sequence of Extend operations recorded in a given 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.

4.3 Confidential Computing & Trusted execution

Security has long been one of the key goals of systems design [18]. 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 the recent years, there has been reported a high number of memory scraping and CPU side-channel attacks. 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 solution at a software level. 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, in order to ensure the isolation of the trusted code. As soon as a trusted code is loaded in a TEE, only specific cores and memory cases are used, aiming to avert side channel attacks. Furthermore, TEEs can also use encryption for the data that is 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 result, TEEs manage to isolate the code and data running 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 of 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 important information (cryptographic and signing keys, passwords etc).

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, which can extend the trust to the hypervisor and/or container engine. The process of extending the trust from a RoT to higher levels of software stack, is called 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 and due to the extra layer of isolation offered by hardware-assisted virtualization.

4.4 Orchestration

To facilitate the deployment of applications in the SERRANO platform and, at the same time, ensure string security guarantees for applications and data, we combine existing Trusted Execution Environments[9] (TEE) infrastructure support and technologies with the cloud native world.

The key concepts to consider are the following:

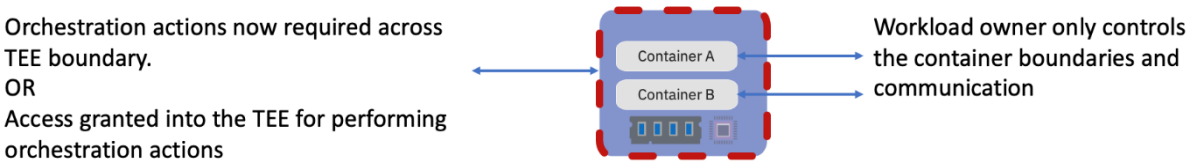
- Allow cloud native application owners to enforce application security requirements
- Transparent deployment of unmodified containers
- Support multiple TEE and hardware platforms
- Introduce a trust model which separates Cloud Service Providers (CSPs) from guest applications
- Apply least privilege principles to the SERRANO platform administration capabilities which impact delivering Confidential Computing for guest applications or data inside the TEE.

TEE's can be used to encapsulate different levels of the architecture stack with three key levels being **node** vs **pod** vs **container**. Container isolation was initially provided with hardware virtualization solutions, such as hypervisors. We now address pod level support for confidential computing. Node level introduces significant challenges around least privilege for kubernetes cluster administration. We will explore the combination of pod and container level isolation, and we expect that challenges explored will have relevance to future understanding of the use of TEE's at the node level.

The TEE seeks to protect the Application and Data from outside threats, with the Application owner having complete control of all communication across the TEE boundary. The application is considered a single complete entity and once supplied with the resources it requires, the TEE protects those resources (memory and CPU) from the infrastructure and all communication across the TEE boundary is under the control of the Application Owner.

[9] https://en.wikipedia.org/wiki/Trusted_execution_environment

4.4.1 Cloud Native Execution Environments



However, moving to a more cloud native approach, the application is now considered a group of one or more containers, with shared storage and network resources (pod). This pod is also subject to an orchestration layer which needs to dynamically interact with the pods and containers with respect to provisioning, deployment, networking, scaling, availability, and lifecycle management.

4.5 Integration

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

Table 1: 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

The respective technology components developed as part of the SERRANO secure infrastructure layer that comprise the relevant tiers will be finalized and presented in D3.4.

5 Conclusions

Deliverable 3.3 reports on the work performed in WP3, mainly regarding Task 3.3: Workload Isolation and Trusted Execution.

The deliverable presents initial findings and planning, as well as preliminary implementation efforts regarding the security aspects of the end-to-end SERRANO platform. Additionally, D3.3 gives a short overview of the Use-cases security concerns, and how the SERRANO secure infrastructure layer addresses them.

This document will be further used as a basis for the technical activities in WP3 and WP5, regarding the development of the SERRANO secure architecture, as well as the SERRANO orchestrator and virtualization software stack. This will also ensure that the outcomes of WP4 and WP5 are aligned with the SERRANO ambition, to secure application deployment across the Cloud-Edge continuum.

The concepts presented in this deliverable will be further developed and the final implementation report will be given in D3.4: Final release of SERRANO Secure Infrastructure Layer (M30).

6 References

[1] "OODA loop," [Online]. Available: https://en.wikipedia.org/wiki/OODA_loop.

[2] "CVE-2019-5736," [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5736.

[3] D. W. e. al, "Unikernels as Processes," in *Symposium on Cloud Computing (SoCC 2018)*, NY, USA, 2018.

[4] "NVIDIA Jetson Nano," [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[5] "NVIDIA Jetson nano overview," [Online]. Available: https://www.microcontrollertips.com/nvidia-jetson-nano-developer-kit-hosts-complete-ai-software-stack/.

[6] "JetPack SDK," NVIDIA, [Online]. Available: https://docs.nvidia.com/jetson/jetpack/introduction/index.html.

[7] "SkyFlok," [Online]. Available: https://www.skyflok.com/.

[8] "Supply chain attack," [Online]. Available: https://en.wikipedia.org/wiki/Supply_chain_attack.

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

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

[11] A. MADHAVAPEDDY, R. MORTIER, C. ROTSOS, D. SCOTT, B. SINGH, T. GAZAGNAIRE, S. SMITH, S. HAND and J. CROWCROFT, "Unikernels: Library operating systems for the cloud," in *ASPLOS*, 2013.

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

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

[14] 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.

[15] 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".

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

[17] 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.

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

[19] "OPTIGA™ TPM SLB 9670 TPM2.0," Infineon, [Online]. Available: https://www.infineon.com/dgdl/Infineon-SLB%209670VQ2.0-DataSheet-v01_04-EN.pdf?fileId=5546d4626fc1ce0b016fc78270350cd6.

[20] "Open Container Initiative Runtime Specification," [Online]. Available: https://github.com/opencontainers/runtime-spec.

[21] "OCI runtime specification," [Online]. Available: https://github.com/opencontainers/runtime-spec.

7 ANNEX A

Implement secure and measured boot using Infineon's TPM 9670 [19] and RPi4, Jetson nano and Xavier AGX development kits.

7.1 Compiling the firmware

U-Boot only accepts pre-baked EFI keys in the security database when stored on an SD card.

On arm64 booting the kernel directly is doable by using the EFI-stub. Distros usually use a heavily modified GRUB. So if the target is distro booting we need to include the Microsoft public certs on the EFI keyring and use MoK (Machine Owner Keys) to sign our binaries.

7.2 Generate certs

```
./gen_efikeys.sh regen <---- 'regen' arg will regenerate the SSL certs
```

- .keys/ —> Private and public certs
- .efi_keys/ -> EFI Signature list certificates

7.3 Building U-Boot

```
git clone https://github.com/u-boot/u-boot.git
cp configs/rpi4_secure_uboot_defconfig u-boot/configs
cp ubootefi.var u-boot/
cd u-boot
make rpi4_secure_uboot_defconfig
ARCH=arm64 CROSS_COMPILE=<cross compiler> make -j$(nproc)
```

7.4 Sign an Image

```
./sign.sh .keys Image
```

Image.signed —> Your signed Image

7.5 Install Ubuntu

```
sudo dd if=ubuntu-21.10-preinstalled-server-arm64+raspi.img of=/dev/sdb bs=
128M status=progress
```

Since Ubuntu pre-installed images don't include an EFI capable kernel, you need to copy your singed kernel to boot/ and your modules to lib/modules

Keep in mind that the RPi4 has several UARTs. Ubuntu seems to be using the real one so you need to edit your config.txt and add

```
overlays=disable-bt
```

7.6 Installing files

Ubuntu installation creates two partition on the SD card. Mount the first partition and edit config.txt to enable the TPM

```
# For more options and information see
# http://rpf.io/configtxt
# Some settings may impact device functionality. See link above for details

[pi4]
# Enable DRM VC4 V3D driver on top of the dispmanx display stack
dtoverlay=vc4-fkms-v3d
max_framebuffers=2

[all]
dtparam=spi=on
dtoverlay=tpm-soft-spi
dtoverlay=disable-bt
arm_64bit=1
enable_uart=1
kernel=u-boot.bin
dtparam=audio=on
```

and copy tpm-soft-spi.dtbo to overlays/

7.7 Booting a target

There's generally two ways of booting an EFI capable kernel either using *bootefi* command or the *EFI Boot manager*.

7.7.1 bootefi

```
# RPI4 needs those since it pass them by defult on the DTB.
# Add these on top of your extra bootatgs or dont change thm at all
setenv bootargs "8250.nr_uarts=1 console=ttyAMA0,115200 kgdboc=ttyAMA0,115200 console=ttyS0
console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 rootwait"

load mmc 0:2 $kernel_addr_r boot/Image && bootefi $kernel_addr_r <---- will fail
load mmc 0:2 $kernel_addr_r boot/Image.signed && bootefi $kernel_addr_r
```

Note that the 8250.nr_uarts=1 implies dtoverlay=disable-bt is used!

7.7.2 EFI Boot manager

```
efidebug boot add -b 0 signed mmc 0:2 boot/Image.signed
efidebug boot order 0
bootefi bootmgr
```

The boot manager can support multiple pairs of kernel/initrd and is better for production.

7.7.3 TPM EventLog

The eventlog produces a human readable form of the system boot measurements

```
sudo tpm2_eventlog /sys/kernel/security/tpm0/binary_bios_measurements
```

7.8 Encrypted FS

Your kernel and initramfs need to move to the firmware partition. Adjust the boot commands accordingly. I.e replace 0:2 -> 0:1 and remove boot/ prefix

7.8.1 On the host

```
sudo cryptsetup luksAddKey /dev/sda2
sudo cryptsetup luksFormat  /dev/sda2
sudo cryptsetup open /dev/sda2 enc_volume
sudo mkfs.ext4 -j /dev/mapper/enc_volume
sudo mount /dev/mapper/enc_volume /media/
sudo cp -aR ~/debian_rootfs/* /media/
sudo cp -ar ~/modules/lib/modules/5.10.87-v7l/ /media/lib/modules/
sudo mount -t proc proc /media/proc/ && sudo mount -t sysfs sysfs /media/sys/ && sudo mount
--bind /dev /media/dev/ && sudo mount --bind /dev/pts /media/dev/pts/
sudo chroot /media/
```

7.8.2 chrooted

blkid -> find the root disk UUID. You can also use PART_UUID and format all boards with the same partition UUID echo “enc_volume UUID= none luks,keyscript=/etc/measuredboot/unseal” >> /etc/crypttab echo “CRYPTSETUP=y” > /etc/cryptsetup-initramfs/conf-hook exit

7.8.3 chrooted scripts

```
/etc/measuredboot/unseal <--- chmod +x
#!/bin/sh

exec 3>&1 1>&2

if [ "$CRYPTTAB_TRIED" = 0 ]; then
        if ! tpm2_unseal --tcti device:/dev/tpm0 -c 0x81000000 -p pcr:sha256:7 1>&3 3>&-; t
hen
                echo 'TPM unseal of LUKS key failed'
                exit 1
        fi
        # extend the PCRs we sealed against in order to make our key unreadable from linux
        tpm2_pcrextend 7:sha256=0000000000000000000000000000000000000000000000000000000000
00000

else
        echo "Decryption try $CRYPTTAB_TRIED"
        keyscript="/lib/cryptsetup/askpass"
        keyscriptarg="Please unlock disk $CRYPTTAB_NAME: "
        exec "$keyscript" "$keyscriptarg" 1>&3 3>&-
fi

/etc/initramfs-tools/measured <---- chmod +x
#!/bin/sh
PREREQ=""
prereqs()
{
        echo "$PREREQ"
}

case $1 in
        prereqs)
                prereqs
                exit 0
                ;;
esac
```

```
. /usr/share/initramfs-tools/hook-functions

copy_exec /usr/bin/tpm2_unseal /usr/bin
copy_exec /usr/lib/aarch64-linux-gnu/libtss2-tcti-device.so.0 /usr/lib/aarch64-linux-gnu
force_load tpm_tis
force_load tpm_crb
force_load tpm_tis_spi
```

7.8.4 FSTAB

```
/dev/mapper/enc_volume /               ext4    errors=remount-ro 0       1
UUID=FE3C-0011  /boot/efi       vfat    umask=0077      0       1
```

You EFI UUID will be different

7.8.5 chrooted

```
update-initramfs -k 5.10.87-v7l -c
cp /boot/initrd.img-5.10.87-v7l /boot/efi
reboot
```

7.8.6 On the host again

```
sudo umount /media/dev/pts  && sudo umount /media/dev/ && sudo umount /media/sys && sudo um
ount /media/proc
sudo umount /media
sudo mount /dev/sda1 /media/
```

At this point boot the board using

```
efidebug boot add -b 0 signed mmc 0:1 Image.signed -i mmc 0:1 initrd.img-5.10.87-v7l -s '82
50.nr_uarts=1 console=ttyAMA0,115200 console=ttyS0 root=/dev/mapper/enc_volume rootwait'; e
fidebug boot order 0 ; bootefi bootmgr
```

7.8.7 Login and seal a new key on the TPM

```
#!/bin/bash

set -e

mkdir -m 0700 /tmp/tpm2
cd /tmp/tpm2

tpm2_createpolicy --policy-pcr -l sha256:7 -L policy.digest
tpm2_createprimary -C e -g sha256 -G rsa -c primary.context
dd if=/dev/hwrng bs=1 count=32 | tpm2_create -g sha256 -u obj.pub -r obj.priv -C primary.co
ntext -L policy.digest \
    -a "noda|adminwithpolicy|fixedparent|fixedtpm" -i -
tpm2_load -C primary.context -u obj.pub -r obj.priv -c load.context
tpm2_evictcontrol -C o -c 0x81000000 || true
tpm2_evictcontrol -C o -c load.context 0x81000000

cd -
rm -rf /tmp/tpm2

tpm2_unseal -c 0x81000000 -p pcr:sha256:7 > new_key.bin
sudo
sudo cryptsetup luksAddKey  /dev/mmcblk0p2 new_key.bin
shred new_key.bin
rm new_key.bin
```

Most of the steps for updating the firmware are described here

7.9 Prepare for flashing

The Jetson nano can boot from either an SD card or the internal SPI NOR. We'll be using the internal SPI NOR for this.

Note: **USE A SEPARATE PSU**

- Download and extract the BSP

- Copy the new U-Boot binary (u-boot.bin) to bootloader/t210ref/p3450-0000

- Ensure the board is powered off

- Connect the USB OTG cable

- Place a jumper across the FRC pins of the Button Header on the carrier board to enable force recovery mode

- For rev. A02, these are pins 3 and 4 of Button Header (J40)

- For rev. B01, these are pins 9 and 10 of Button Header (J50)

- Connect a DC power adapter. Remove the jumper from the FRC pin

```
lsusb | grep -i nvidia
sudo ./flash.sh p3448-0000-max-spi external
```

After rebooting the board you should be on the new U-Boot loader

7.10 Booting a signed kernel

```
setenv bootargs "console=ttyS0,115200n8 console=ttyTHS0 console=tty0 root=/dev/mmcblk1p2 ro
otfstype=ext4 rootwait"
load mmc 1 $fdt_addr_r tegra210-p3450-0000.dtb && load mmc 1 $kernel_addr_r Image.signed &&
bootefi $kernel_addr_r $fdt_addr_r
```