SEED: A Minimal-Footprint TEE Framework for Verifiable, Confidential Microservice Deployment

Omar Jarkas
The University of Queensland
Brisbane, Australia
o.jarkas@uq.edu.au

Ryan K L Ko
The University of Queensland
Brisbane, Australia
ryan.ko@uq.edu.au

Naipeng Dong
The University of Queensland
Brisbane, Australia
n.dong@uq.edu.au

Redowan Mahmud
Curtin University
Perth, Australia
mdredowan.mahmud@curtin.edu.au

Abstract

We present SEED, a system that enables the deployment of distributed privacy-preserving micro-services in the cloud while maintaining the secrecy of user code and data and ensuring correct, complete results. Unlike prior approaches that minimize the TCB by pushing large parts of the software stack outside the enclave, SEED includes the entire container software stack—from the application layer up to the operating system—inside the TCB. This holistic design protects proprietary software, datasets, and optional ML models from exposure; prevents leakage of sensitive inputs or queries; and thwarts metadata-inference attacks that could reveal workload identity or versioning. Yet we achieve an optimized TCB (22 MB in total), over 30× smaller than the typical 690 MB TCB for confidential privacy-enhancing VMs. In practice, SEED runs on AMD SEV-SNP–capable machines and supports real container workloads (i.e., TensorFlow, OpenVINO inference, PyTorch training, Redis, NGINX, Apache httpd). We demonstrate that SEEDCore matches or outperforms mainstream runtime workload deployment, staying within 5% of native throughput and reaching up to 6× higher performance on CPU-bound jobs. Finally, we conduct a thorough privacy and security evaluation against 11 cloud attack vectors and show that SEED blocks or confines every exploit that remains possible even under the state-of-the-art Gramine-TDX model, thanks to late binding, per-container PCR chains, and continuous in-TEE attestation throughout the workload's lifetime.

1 Introduction

Modern AI/ML pipelines, language runtimes, and web services—even in heavily-regulated sectors such as finance, healthcare, and telecom edge—ship as *Open Container Initiative (OCI) containers*: self-contained images that a runtime (`crun` [20], `runc` [51]) or engine (Docker [46], Kubernetes [35]) can pull, replicate, and start in milliseconds. This late-binding model decouples build from run and has become the de-facto unit of cloud deployment. While hardware Trusted Execution Environments (TEEs) protect code in use [60, 19, 48, 62], running containers on an untrusted cloud still requires a seamless *push→run→prove* pipeline. Crucially, that pipeline must guarantee privacy for *all* tenant workloads—shielding code, data, and runtime characteristics from the cloud operator and neighbouring tenants. Effective TEE frameworks must therefore support unmodified container images and on-demand pulls—features currently incompatible with existing TEE designs [10].

This incompatibility arises primarily from the size and placement of the *Trusted Computing Base (TCB)*, comprising hardware, firmware, and software components that must remain trusted to preserve privacy and integrity. However, achieving comprehensive endpoint security and privacy through TEEs is subject to overcoming its structural challenges:

- **Process-level TEEs** safeguard individual address spaces but introduce a *pre-enclave window* where secrets appear in plaintext, while preventing containers from being pulled securely post-launch (§2.2);
- **VM-level TEEs**, broaden protection to entire virtual machines, enabling secure post-boot container pulls but incorporating a complete multi-user OS and container stack into the TCB, thereby significantly enlarging both the trusted footprint and the attack surface (§2.2).

Existing methods aiming to minimize the TCB—such as LibOS partitioning, security monitors, or unikernels (§3)—either disrupt essential features like on-demand pulls, verification, binary and forward compatibility, or inadvertently expose vulnerabilities through host interfaces that can leak workload content or operational metadata.

This study presents SEED, a novel TEE framework that preserves standard container workflows while shrinking the enclave to a verifiable, unikernel-scale TCB. Unlike prior designs that either drag a full multi-user OS into the enclave or freeze a single immutable process (§3), SEED embeds only a minimal container runtime inside the TEE and admits and tracks containers *after* attestation. This eliminates the pre-enclave window without sacrificing off-the-shelf container compatibility.

Technically, SEED statically links to a patched Linux 6.8, `glibc` [58], and a minimal runtime (erun) into one 22 MB ELF, then binds each pulled image to this core via vTPM-backed attestation and a container-aware Integrity Measurement Architecture (IMA) namespace. This co-design establishes a new paradigm in TEE deployment by unifying unikernel-scale TCB, late binding, and continuous streamlined hardware-rooted integrity in a single pipeline,

Proceedings on Privacy Enhancing Technologies 2026(1), 411–425

https://doi.org/10.56553/popets-2026-0021

outperforming LibOS, monitor-based, and traditional unikernel approaches in both attack-surface reduction and runtime flexibility. SEED achieves this via the following contributions:

- **Privacy-first *push→run→prove* Deployment Framework.** Unified system- and network-layer framework that turns TEE deployment model into a verifiable defense against pre-enclave attacks and tying each workload to a privacy-anchored attestation chain, SEED prevents exfiltration of code, data, or configuration throughout the workload lifecycle.(see §6)
- **Unikernel-scale TCB Optimisation.** SEEDCore[1] embeds only the essential container runtime inside the kernel, yielding a unikernel-scale TCB that closes the pre-enclave window yet preserves ordinary container workflows.
- **Kernel-resident, Statically Linked erun.** We demonstrate, for the first time, that a full-featured, OCI-compatible container runtime can be *statically linked* into the kernel without forfeiting lift-and-shift flexibility. Running entirely in kernel space eliminates the user-kernel syscall boundary, trims the dependency closure to ∼180 kLoC (versus >2 M LoC for Docker + containerd), and still supports unmodified images, registry pulls, and layer caching (§7.2). This fusion of unikernel tightness and late-binding enforces a strong privacy boundary for any container workload—web services, analytics jobs, or ML inference alike.
- **Hardware-rooted, Per-container Attestation for a Mutable TCB.** Traditional Linux IMA cannot streamline our deployment framework because, to the kernel, containers are just user-space constructs isolated by namespaces and cgroups; that user/kernel split hides per-container activity and blocks fine-grained integrity checks. By statically linking the container runtime into the kernel, SEEDCore collapses this boundary and promotes the container to a first-class kernel object. We therefore implement a container-aware IMA namespace: every file erun loads is tagged with its container ID and extends into a granular and unified view inside a lightweight vTPM built on AMD-SEV firmware (§ 7.1). This fine-grained chain of trust blocks inference attacks that could reveal which software stack is running, when it was updated, or how it is being exercised.
- **Performance evaluation in real confidential clouds.** On Azure SEV-SNP VMs, Seedcore delivers ≥95% overhead on CPU-bound micro-benchmarks, 20–60% higher ML throughput than mainstream runtimes, and ≥95% native web-server throughput across 1–100 concurrent clients, while erun outperforms crun, containerd, docker, podman on every AI/ML pipeline configuration tested (§9.1.4–9.1.3).

The methodology opens new directions for secure, multi-tenant cloud services that require drop-in privacy, confidentiality and integrity without re-engineering existing container ecosystems.

2 Trusted Execution Environments (TEEs)

TEEs establish isolated enclaves and leveraging CPU-enforced memory encryption and strict access control policies, TEEs can protect sensitive information against a wide range of threats, including those arising within hypervisors or operating systems. However, achieving comprehensive endpoint security through TEEs is fraught with practical challenges, stemming not only from the complexities of how TEEs are structured (e.g., as process-level or VM-based enclaves), but also from fundamental design trade-offs involving trust boundaries and the size of the TCB.

2.1 Process-level TEEs

Process-level TEEs (e.g., Intel SGX [15]) protect a *single* address space. At launch the CPU hashes the enclave's initial pages into `MRENCLAVE` [15]; any post-launch change breaks that hash. This immutability keeps the in-enclave code verifiably trusted, but it also complicates late-bound components: while SGX supports encrypted enclaves (where code and data are encrypted and only decrypted at runtime) and sealing mechanisms for offline secret protection, the fundamental challenge remains that container images cannot be dynamically pulled and verified post-attestation without breaking the enclave's measurement. Thus, while runtime isolation is strong, process-level TEEs integrate poorly with modern container workflows that rely on dynamic loading and updates.

2.2 VM-level TEEs

Given the limitations of process-level TEEs, many practitioners have moved to VM-level TEEs – systems which host an unmodified Linux kernel inside an enclave and still support modern container workflows, such as pulling and running standard container images. In these designs, the secure firmware boots a virtual TPM (vTPM) [53, 49], a software component providing cryptographic measurement and reporting functions to the guest VM. The firmware then measures the BIOS, bootloader, and kernel chain into Platform Configuration Registers (PCRs), and exposes those PCRs for remote attestation, a process allowing a remote party to cryptographically verify the integrity of these boot measurements. Only after this hardware-anchored baseline is verified does the in-guest container runtime pull and start user images, eliminating the *pre-enclave* window that process-level TEEs suffer from.

3 Related works

Prior work on confidential containers is shaped by a fundamental trade-off: Linux-with-containers preserves OCI semantics and late binding but inherits a large, multi-purpose TCB; unikernel designs achieve tiny TCBs but lose container workflows and typically lack continuous, workload-lifetime attestation. A third line of work inserts a security monitor to reduce audited code but adds costly boundary crossings and constrains container-granular trust. We use this lens to organize prior work.

Running a full Linux guest inside a TEE preserves POSIX and OCI semantics [77, 17, 56, 18], enabling unmodified containers with late binding and standard runtime ecosystems. Yet existing systems still forfeit at least one of *confidentiality*, *integrity*, or *practicality*.

This compatibility comes with a ballooning TCB: network stacks, filesystems, drivers, and kernel subsystems designed for multi-tenant bare-metal servers add little value inside single-tenant confidential containers [32]. Worse, Linux views containers as mere namespaces and cgroups—*userspace fiction* [6, 14]—preventing security modules like IMA from expressing per-container policies.

[1] **SEEDCore** is the core trusted computing base of SEED, comprising a minimal binary that includes a patched kernel and erun, and dependencies statically linked into a single ELF executable

Table 1: Qualitative comparison of TCB–minimisation strategies. Icons: ✅ desirable ⛔ partial ❌ undesirable.

Approach	TCB Size	Runtime Flex.	Pre-Enclave Exposure	Side-channel / Iago	Binary Compat.
Partitioned LibOS	✅ Small (8-15 MB)	⛔ Emulated kernel functions	❌ High – pulls pre-attestation	❌ Elevated – ABI leaks	✅ Needs patched `glibc`
Security Monitor	✅ Small (5-10 MB)	✅ Full Linux outside enclave	❌ High – data leaves TEE	⛔ Context-switch channels	✅ Excellent
Unikernel	✅ Tiny (2-5 MB)	❌ None – immutable image	❌ High – pre-boot linking	✅ Low	❌ ABI freezes
Monolithic Linux	❌ Large (690 MB)	✅ OCI-compatible	✅ Low	❌ High – full kernel in TEE	✅ Excellent
SEEDCore	✅ Small (22 MB)	✅ OCI delayed loading	✅ None – vTPM-enbled IMA	✅ Micro-architectural Only	✅ Excellent

Table 1 summarizes this landscape; we examine how each category attempts to minimize risk:

Partitioning the System into Trusted and Untrusted Components. Systems such as Gramine [69, 23], SCONE [2], Occlum [63], and Civet [70] embed a micro-kernel-sized set of services inside the enclave while delegating the rest—including the full container stack—to the untrusted host. This shrinks the kernel portion of the TCB but creates two high-risk interfaces: (i) the enclave–kernel boundary, vulnerable to *Iago attacks* [7], and (ii) the container-runtime boundary, where host-visible syscall/cgroup traces leak workload behaviour [73].

Leveraging a Security Monitor. SeKVM [38], HyperEnclave [29], Veil [1], and Keystone [37] interpose a hardware-rooted monitor to mediate privileged events and produce attestation. The audited code base remains small, yet each I/O or page-table update crosses monitor boundaries—incurring latency and obliging developers to re-encrypt or re-validate data.

Reducing TCB Size by Removing Functionality. Unikernels (MirageOS [44], OSv [32], X-Containers [64], UKL [72]) link one application to the kernel code it needs, producing few-megabyte images amenable to formal verification. Immutability is intrinsic: no new code can load after build time, clashing with late-bound container workflows.

As Table 1 makes clear, each category attains at most two of the three goals—small TCB, first-class containers, or continuous attestation. SEED departs conceptually along three axes: (i) *kernel-native containers*, where policy and measurement are container-scoped in the kernel rather than via a userspace runtime; (ii) a *supervisor-mode process model on UKL* that preserves OCI late binding without re-inflating the TCB; and (iii) a *continuous, hardware-rooted attestation path* that binds code and data flows (I/O, storage, network) to current measurement state. These ideas break the architectural constraints that prevent this combination in other systems and enable a unikernel-scale TCB *with* full container support *and* continuous attestation. We detail mechanisms in §5 and analyze privacy implications in §8.1. To our knowledge, SEED is the first system to deliver all three properties simultaneously.

4 Threat Model

SEED protects the confidentiality, integrity, and correctness of code and data throughout all operational stages—at rest, in transit, and during execution. It achieves this by relying on a structured trust framework and deploying defensive measures against powerful adversaries.

Trusted Components Two primary actors govern the SEED architecture: the untrusted cloud provider (P_1), who controls the infrastructure and may attempt to compromise the workload, and the trusted cloud user (P_2), who owns the workload and seeks to ensure its security. Within this context, five trusted components form the trust foundation (see Figure 1):

- $\mathcal{E}_1$ ***CLI***: Acts as an administrative interface, managing deployment, encryption keys, and attestation policies.
- $\mathcal{E}_2$ ***CAS***: Functions as a secret broker and verifier. It attests to component integrity, encrypts and signs workloads and configurations, manages secrets, and authenticates entities via access keys [3, 10].
- $\mathcal{E}_3$ ***vTPM***: Derives a hardware root of trust to locally attest and authenticate components.
- $\mathcal{E}_4$ ***TCB***: Provides a confidential VM (CVM) enclave with minimized internal attack surfaces and integrated integrity measurement capabilities leveraging $\mathcal{E}_3$.
- $\mathcal{E}_5$ ***Workload***: Represents the containerized instance throughout its lifecycle, protected by the above components.

The first two components ($\mathcal{E}_1$ and $\mathcal{E}_2$) remain under the control of the trusted user, ensuring direct oversight and transparency. The others ($\mathcal{E}_3$, $\mathcal{E}_4$, and $\mathcal{E}_5$) reside in the cloud, where the provider may run arbitrary code, intercept data, and attempt to exploit vulnerabilities. To mitigate these risks, SEED relies on integrity measurements, attestation reports, and the continuous verification of component authenticity.

Certification Authorities (CAs) SEED utilizes three independent CAs to further solidify trust:

- CA_1 *Cloud CA*: Issued by the cloud vendor for authenticating and managing service credentials.
- CA_2 *AMD CA*: Ensures the hardware's authenticity and the validity of attestation reports.
- CA_3 *SEED CAS CA*: A self-signed authority within SEED that issues and manages internal certificates independent of external trust anchors.

Scope Limitation for Workload Users External workload users and systems interacting through application-level interfaces lie outside SEED's primary threat model. Application-specific threats such as deserialization vulnerabilities or advanced hardware-level attacks—cache [22, 43, 74], [76, 27], power analysis [39, 40], Spectre [34, 45], and Meltdown [41, 9]—remain unaddressed by SEED itself. While SEED provides guarantees at the container runtime level, complete protection requires complementary techniques such as microarchitectural defenses (i.e., hardware features like Intel CAT [31] or software mitigations (e.g., page coloring, cache partitioning) to defend against cache-based side channels), application hardening, and supply chain security (i.e., container image signing, vulnerability scanning, and software bill of materials (SBOM) verification to ensure trusted workload sources).

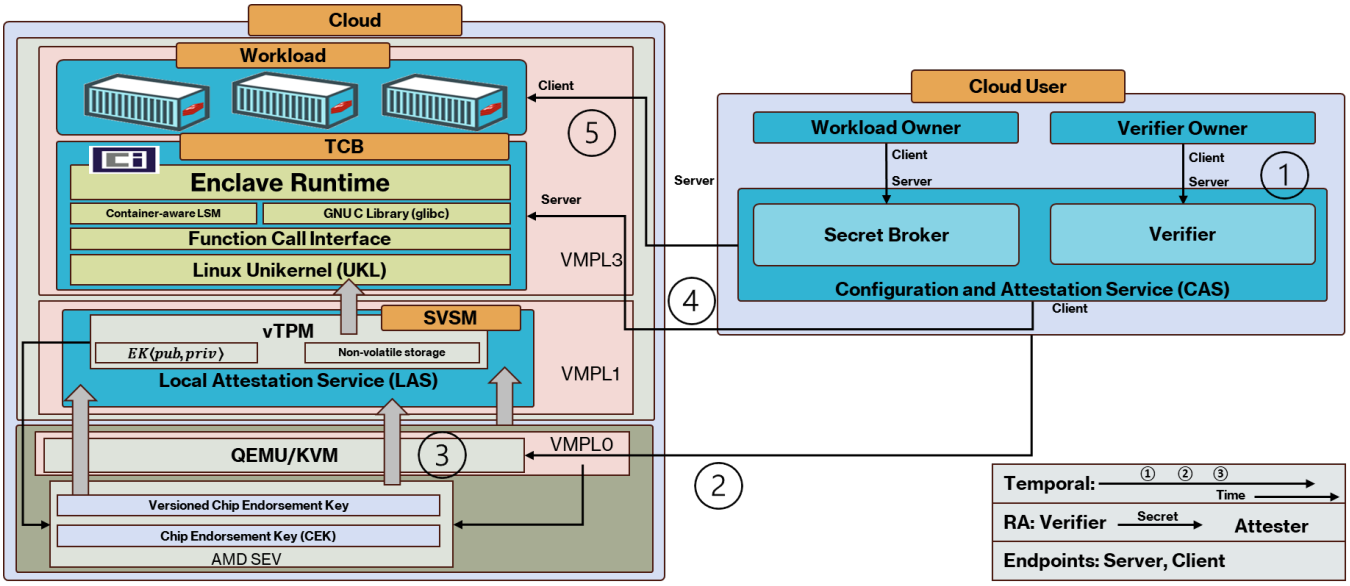


Figure 1: Architectural Overview of SEED's Software Stack and eRun, highlighting VM privilege levels used for vTPM segregation. While VMPL is AMD-specifc, such privilege level segregation is transferable across all VM-level encryption technologies, providing a confidential computing solution across various configurations (detailed in §2).

SEED focuses on eliminating the pre-enclave exposure window and minimizing the TCB while maintaining container compatibility—it serves as a foundational layer that must be augmented with these domain-specific protections for comprehensive security.

Algorithm 1 Attested TLS Handshake Protocol

Require: Participants: Client (**C**), Server (**S**)
Ensure: Mutual authentication and secure key exchange with attestation

procedure AttestedTLS(**C**, **S**)
  Notation and Cryptographic Functions:
    $H(\cdot)$: Collision-resistant hash function (SHA-256)
    $\text{PRF}_k(\cdot)$: Keyed pseudo-random function with key k
    $\text{Enc}_k[\text{ad}]\{\cdot\}$: Authenticated encryption with key k and associated data
    $\text{Sign}_{\text{sk}}(\cdot)$: Digital signature using private key sk
    g^x, g^y: Diffie-Hellman ephemeral public keys
    cr, sr: Client and server random nonces
    k_{sh}, k_{ch}: Server/client handshake encryption keys
    k_{sm}, k_{cm}: Server/client MAC keys for handshake
    k_{ap}: Application data encryption key (post-handshake)
    $\text{transcript}_S, \text{transcript}_C$: Handshake message transcripts
  Stage 1: Initial Handshake - TLS
  C → **S**: *ClientHello* ← (ciphers ‖ versions ‖ cr ‖ g^x)
  S → **C**: *ServerHello* ← (selected_suite ‖ selected_version ‖ sr ‖ g^y)
  S → **C**: *EncryptedExtensions* ← $\text{Enc}_{k_{sh}}[\text{ad}]\{\text{ext}\}$
  Stage 2: Endpoint Authentication and Attestation - mTLS
  S → **C**: *CertificateRequest* ← $\text{Enc}_{k_{sh}}[\text{ad}]\{\text{cert\_req}\}$
  S → **C**: *Certificate* ← $\text{Enc}_{k_{sh}}[\text{ad}]\{\text{cert}_{\text{pubTIK-S}}\}$
  S → **C**: *CertificateVerify* ← $\text{Enc}_{k_{sh}}[\text{ad}]\{\text{Sign}_{\text{privTIK-S}}(\text{transcript}_S)\}$
  S → **C**: *Finished* ← $\text{Enc}_{k_{sh}}[\text{ad}]\{\text{PRF}_{k_{sm}}(\text{transcript}_S)\}$
  C → **S**: *Certificate* ← $\text{Enc}_{k_{ch}}[\text{ad}]\{\text{cert}_{\text{pubTIK-C}}\}$
  C → **S**: *AttestationEvidence* ← $\text{Enc}_{k_{ch}}[\text{ad}]\{\text{Evidence}_C\}$ ▷ Attestation integrated in handshake
  C → **S**: *CertificateVerify* ← $\text{Enc}_{k_{ch}}[\text{ad}]\{\text{Sign}_{\text{privTIK-C}}(\text{transcript}_C)\}$
  C → **S**: *Finished* ← $\text{Enc}_{k_{ch}}[\text{ad}]\{\text{PRF}_{k_{cm}}(\text{transcript}_C)\}$
  Stage 3: Attestation Evidence (encrypted)
    ▷ First *application-data* record after the 2×Finished
  C → **S**: $Evidence^*$ ← $\text{Enc}_{k_{ap}}[\text{ad}]\{\text{quote, claims}\}$
    ▷ **S** MUST verify before accepting/issuing any further app-data

5 Design Rationale

SEEDCore revisits the basic assumptions behind confidential-container deployment. First, production confidential VMs are almost always single-tenant, so no adversarial processes inhabit the enclave. Second, once the VM is encrypted, multi-user Linux baggage—drivers, UID/GID isolation, ptrace hooks, hundreds of dormant syscalls—only swells the attack surface, while a third insight shows the top-20 Docker-Hub images (83 % of pulls) rely on a very small syscall set.

***Userspace collapse via statically-linked, kernel-resident* `erun`** We link `erun`—a modified `crun`—into the kernel at `VMPL3`, creating a single address space that (a) removes the enclave–runtime interface that leaks syscalls, (b) keeps every byte of code and data inside encrypted memory, and (c) trims the TCB to 22 MB—about 30× smaller than a full confidential VM. Third-party runtimes (Python, JVM, etc.) arrive as ordinary OCI layers, so compatibility is retained while the host sees no plaintext and no syscall traces. Implementation required complete container toolchain reimplementation, as Go-based utilities cannot execute in kernel space (see §7.2.1).

Code reduction methodology. To achieve the 22MB TCB, we systematically removed subsystems unnecessary for single-tenant confidential containers. Starting from crun's 26,529 lines in its core library, erun eliminates: cgroups (5,500 LOC)—redundant when the entire VM serves one tenant; eBPF (554 LOC) and seccomp (1,051 LOC)—attack surface with no benefit inside TEEs; CRIU checkpointing (1,053 LOC)—incompatible with attestation chains; Intel RDT (369 LOC)—inapplicable to AMD SEV. This surgical removal yields 14,708 LOC in erun's library (45% reduction) while preserving OCI compatibility. Combined with kernel-side stripping of unnecessary drivers and subsystems, the binary measures 22MB—small enough for line-by-line audit yet enough for production containers.

Container-granular, hardware-rooted attestation. Elevating containers to first-class kernel objects enables dedicated PCR chains per container. IMA measures each FileExec event before execution;

the vTPM quotes chains. Prior container-aware IMA attempts [42, 25] measure from userspace, creating imprecise detection. Only kernel-integration enables per-container measurement visibility.

Zero-exposure deployment pipeline. Attested-TLS (Algorithm 1) establishes trust among CAS, vTPM, and kernel before transmitting OCI layers. Container data never leaves protected DRAM, eliminating pre-enclave exposure while maintaining standard workflows on a small TCB with continuous hardware attestation.

> Standard TLS provides confidentiality and integrity for data in transit but cannot prove the *code integrity* of the endpoints. A malicious cloud provider could present a compromised TCB that passes TLS authentication while running tampered code. Attested-TLS extends the handshake to include hardware-rooted attestation evidence, cryptographically proving that: (i) the endpoint runs inside genuine TEE hardware (via VCEK signature); (2) the software stack matches expected measurements (via PCR values); (3) the attestation is fresh and bound to this session (via nonce). Only after verifying this evidence does sensitive data flow.

Practical deployment through UKL's hybrid model. Traditional unikernels achieve small TCBs by specializing for a single purpose and freezing binaries at build time, which breaks binary/forward compatibility and precludes dynamic container workflows. SEED instead adopts UKL (Unikernel Linux), which allows processes to run at supervisor privilege within a Linux kernel, preserving the Linux ABI and forward compatibility while retaining a unikernel-scale TCB. This yields in-practice benefits: (i) Binary compatibility with unmodified Linux applications; (ii) Dynamic container loading post-boot (late binding); (iii) Multi-workload support within a single enclave/VM; (iv) Compatibility with existing container tooling.

These choices target host-visible leakage channels (syscall/cgroup traces, system-wide hash exposure) while retaining OCI workflows (see §8.1).

We evaluate the design consequences in §8.1 and §9.

6 Privacy-first Deployment Framework

Overview. SEED turns a standard *push* into a hardware-rooted, three-step pipeline—*push*→*run*→*prove* (with optional *seal*)—shown in Figure 1. A developer drives the process from the CLI ($\mathcal{E}_1$), which calls the *Configuration & Attestation Service* (CAS, $\mathcal{E}_2$). The CAS first authenticates to the cloud control-plane, then launches a single SEV-SNP VM that contains both the Trusted-Computing Base ($\mathcal{E}_4$) and an in-guest vTPM ($\mathcal{E}_3$). Using an *attested-TLS* handshake (Algorithm 1) the CAS, vTPM, and TCB build a chain of trust rooted in AMD's SNP report: they verify the CPU, the vTPM firmware, and finally the kernel-integrated runtime `erun`. Only after that chain closes does the encrypted OCI image flow into the VM, where `erun` executes it, the vTPM continuously re-measures it, and—if requested—the workload's state is sealed to hardware-bound keys.

Deployment sequence. The numbered circles in Figure 1 trace the complete deployment flow: (1) The CLI configures the CAS with deployment parameters and container references; (2) CAS authenticates to the cloud control plane and requests a confidential VM; (3) The hypervisor boots the SEV-SNP VM containing the vTPM (VMPL1) and TCB (VMPL3); (4) CAS and vTPM establish an attested-TLS channel, validating the vTPM via AMD's VCEK chain; (5) CAS and TCB establish a second attested-TLS channel for secure container transfer. Once trust is established, encrypted OCI layers flow directly into kernel memory where erun executes them while IMA continuously measures each file operation into per-container PCRs. This choreography ensures that container data never exists in plaintext outside the hardware-encrypted VM boundary.

The deployment process begins when the *Configuration and Attestation Service* (CAS, $\mathcal{E}_2$) is configured via the CLI ($\mathcal{E}_1$) (step ①). The CAS obtains a access token from the cloud's control plane (CA_1) and forms a mutually-authenticated TLS (step ②).

During the confidential runtime-loading phase, the hypervisor boots the (TCB, $\mathcal{E}_4$) and a *virtual TPM* (vTPM, $\mathcal{E}_3$) inside an SEV-SNP-protected VM (step ③). The vTPM provisions its keys (§7.1) used for remote attestation, authentication, and sealing. Meanwhile, the AMD *Secure Processor* (ASP) generates an *SNP attestation report* whose measurement is cryptographically bound to the VM and signed by a VCEK certified by AMD.

An attested TLS handshake between CAS and vTPM (step ④) validates: (i) the integrity of the vTPM via its AIK-signed quote, and (ii) the authenticity of the CPU via the SNP attestation report and the AMD certificate chain CA_2. Once trust is established, a second TLS channel is negotiated between the CAS and the TCB (step ⑤) using CA_3-issued certificates, ensuring that only attested components exchange the encrypted container image. The TCB returns its own SNP report to the CAS, proving that it executes on genuine hardware and that its measurement matches the hash.

Once attested, CAS streams the encrypted OCI image. `erun` loads it in-kernel, while the container-aware IMA logic measures each executable page and extends the appropriate PCR. Optional sealing (§6.4): a data-encryption key derived from the TLS session (*Data-Encryption Key* (DEK)) is sealed under vTPM SRK and bound to the measured TCB, so only a re-attested SEED instance can re-hydrate state (§6.4). This flow executes unmodified container images, maintains continuous attestation, and never exposes secrets outside encrypted memory turning a normal image push into the hardware-rooted pipeline shown in Fig. 1: *push*→*run*→*prove*, with optional *seal*.

6.1 Push: Environment Setup and Verification

This phase securely prepares and verifies the cloud execution environment *before* any workload is introduced. Unlike a conventional container "push," it includes a secure boot sequence and remote attestation, confirming the trustworthiness of the TCB, $\mathcal{E}_4$. Specifically, it checks both the TCB's software integrity and the underlying CPU's authenticity—thereby establishing a hardware root of trust. Only after this verification can workloads run in a genuinely confidential, integrity-protected environment.

6.1.1 Authentication. This process begins with $\mathcal{E}_1$ establishing a secure, encrypted, and authenticated communication channel via HTTPS/TLS with the cloud infrastructure. The Configuration and Attestation Service (CAS, $\mathcal{E}_2$) initiates a TLS handshake with the cloud provider's API endpoint to establish a secure channel. This begins with a ***ClientHello*** message, in which the CAS advertises supported cipher suites, protocol versions, and includes a client random (cr) and an ephemeral public key (g^x). The cloud provider

responds with a ***ServerHello***, selecting the cipher suite and protocol version, plus its own random (sr) and ephemeral key (g^y).

Both parties compute a shared secret $k = (g^y)^x = (g^x)^y$, then derive session keys:

$$k_{sh}, k_{ch} \quad \text{for handshake encryption.}$$
$$k_{sm}, k_{cm} \quad \text{for secure message transmission.}$$

Mutual authentication follows via mTLS (Stage 2 of Algorithm 1). The server sends a certificate (encrypted under k_{sh} and signed by CA_1) confirming its identity:

$$\text{Certificate} = \text{Enc}_{k_{sh}}[\text{ad}]\{\text{cert}_{\text{pub}CA_1\text{-}P_1}\}$$

The CAS then provides its own certificate:

$$\text{Certificate} = \text{Enc}_{k_{ch}}[\text{ad}]\{\text{cert}_{\text{pub}CA_1\text{-}\mathcal{E}_2}\}$$

Upon successful exchange and verification, the CAS has an authenticated, encrypted channel to the cloud services.

6.1.2 Boot/Attest. Upon validating the request, the cloud platform executes predefined logic to load the necessary components. This step can be automated with Infrastructure-as-Code (IaC) tools such as Terraform or Pulumi, allowing developers to manage the cloud topology from declarative configuration files. In this sequence, the *untrusted* hypervisor loads the SEED guest image (pre-staged on the cloud platform). The image bundles the logic for the vTPM $\mathcal{E}_3$ and the TCB $\mathcal{E}_4$, but *excludes* the workload $\mathcal{E}_5$, which is injected later at runtime.

After the guest image is launched, the hypervisor invokes the `LAUNCH_START` ioctl exposed by KVM, triggering the AMD *Platform Security Processor* (PSP) to enter the **SEV-SNP launch flow**. During this *instantiation and provisioning* phase the PSP:

- allocates a unique `ASID` and derives a per-VM AES-XTS memory-encryption key,
- locks the guest's pages into the RMP so that only the VM can access them,
- records a cryptographic measurement of the initial guest memory via `LAUNCH_UPDATE` commands, and
- finalises the launch with `LAUNCH_FINISH`, returning an *SNP attestation report* bound to the VM's measurement and signed by the (VCEK certified by AMD.[2])

SEV-SNP's hardware state ensures that code running inside the VM (including the vTPM and TCB) executes at *VM Privilege Level 0 (VMPL0)*, which is higher than any guest user-space and isolated from the host hypervisor. All guest memory is transparently encrypted in DRAM, and any attempt by lower-privileged entities (e.g. the hypervisor or another VM) to access protected pages triggers a hardware fault, preserving confidentiality and integrity even in the face of a malicious host.

With the launch flow complete, the vTPM and TCB proceed to remote attestation and secure workload injection as described in the following sections.

6.1.3 vTPM Provision. Once the vTPM is activated inside the SEV/SNP VM, it generates:

- **EK Pair** – Endorsement Key used as the TLS client credential,
- **AIK Pair** – Attestation Identity Key for signing quotes,
- **SRK Pair** – Storage Root Key for sealing and binding operations (§6.4).

[2]See AMD SEV-SNP specification [30].

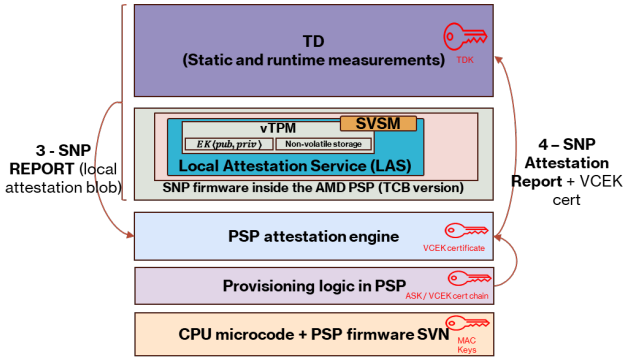


Figure 2: Overview of layered attester for SEAM-vTPM

All keys are derived from the vTPM's hardware RNG seed (§7.1). Immediately after key generation, the vTPM performs a *local* attestation of the SEV/SNP environment: it requests an **SNP attestation report** from the Platform Security Processor (PSP). The report cryptographically binds the VM's launch measurement to a VCEK signed by AMD and embeds EK_{pub} in one of the report's `REPORT_DATA` fields. The full report is persisted in the vTPM's `NVIndex` so that any remote verifier—over an attested TLS channel—can retrieve the tuple $\langle$SNP Report, $EK_{\text{pub}}\rangle$ and validate that the TCB is executing inside genuine SEV/SNP hardware (§6.1.4, §6.1.5).

6.1.4 Obtain Attestation Report. Figure 2 depicts the *local attestation* path when SEED runs on AMD SEV/SNP. Unlike Intel TDX—which relies on a TD Quoting Enclave (TDQE) and Provisioning Certification Enclave (PCE)—SEV/SNP delegates report generation to the on-die *Platform Security Processor* (PSP). The PSP produces a hardware-signed `SNP_REPORT` whose signature chain ultimately terminates at AMD's *Root Key* (ARK), removing the need for in-guest quoting enclaves.

Report Request. The vTPM populates a `REPORT_REQ` structure in the *Guest-Hypervisor Communication Block* (GHCB) and triggers the `VMGEXIT` instruction. The request is encrypted and integrity-protected with the VM-specific *Secure Guest Key* (SGK) so that an untrusted hypervisor cannot tamper with it. The `report_data` field is filled with a SHA-256 digest that binds the vTPM's public endorsement key to the current TLS session:

$$\texttt{report\_data} = \text{SHA256}(\text{protVerUp} \,\|\, \text{protVerLow} \,\|\, EK_{\text{pub}} \,\|\, n_c)$$

where `protVer*` encode protocol compatibility, EK_{pub} is the vTPM's endorsement key, and n_c is the TLS-exported nonce acting as a channel binder.

SNP_REPORT Generation. Upon the `REPORT_REQ`, the PSP

- verifies the VMPL-0 privileges of the caller,
- measures the guest-owner-supplied pages recorded in the RMP, and
- returns a 400-byte `SNP_REPORT` that includes (*i*) the VM's measurement hash, (*ii*) CPU and PSP Security Version Numbers (SVNs), and (*iii*) the user-supplied `report_data`.

The report is MAC-ed and signed with the socket-unique *Versioned Chip Endorsement Key* VCEK, which in turn is certified by AMD's *ASK* (AMD Signing Key) and *ARK* root certificate.

Remote-Evidence Assembly. The vTPM stores the SNP_REPORT in an NVIndex and bundles it with the certificate chain:

$$\text{Evidence} = \langle \texttt{SNP\_REPORT}, \text{VCEK}, \text{ASK}, \text{ARK} \rangle.$$

Any external verifier—over the attested TLS channel—can now
- validate the VCEK against AMD's ASK/ARK hierarchy,
- verify the PSP signature on SNP_REPORT, and
- check that the embedded hash matches EK_{pub} and the nonce n_c, thereby confirming both the vTPM's identity and the live TLS session.

This flow establishes a strong chain of trust from AMD hardware (ARK) through the PSP and into the guest VM, culminating in cryptographic evidence that the vTPM and the broader SEED TCB execute inside genuine, unmodified SEV̄/SNP silicon.

6.1.5 Verify. Finally, the CAS ($\mathcal{E}_2$) and vTPM ($\mathcal{E}_3$) establish an *attested TLS* handshake (differing from the earlier "cloud handshake") where the vTPM uses EK_{pub} as its TLS certificate and includes the TD Quote. Both endpoints incorporate a nonce n_c to bind the quote to the TLS session.

The CAS ($\mathcal{E}_2$) and the vTPM ($\mathcal{E}_3$) now perform an *attested TLS* handshake—This is distinct from the earlier *cloud handshake* in which the CAS authenticated to the provider's control plane. The vTPM presents EK_{pub} as its TLS certificate and embeds the AMD SNP_REPORT bundle (Evidence) in the first encrypted record. Both peers insert the nonce n_c into their random structures to bind the report to the session.

Handshake Process. The handshake starts with ***ClientHello*** and ***ServerHello***, both including n_c:

$$\mathcal{E}_2 \to \mathcal{E}_3 \;\; \textit{ClientHello} \leftarrow (\cdots \parallel n_c)$$
$$\mathcal{E}_3 \to \mathcal{E}_2 \;\; \textit{ServerHello} \leftarrow (\cdots \parallel n_c)$$

After deriving session keys, mutual authentication follows using CA_3-signed certificates. The vTPM sends its certificate, including EK_{pub}:

$$\mathcal{E}_3 \to \mathcal{E}_2 \;\; \textit{Certificate} \leftarrow \text{Enc}_{k_{sh}}[\text{ad}]\{\text{cert}_{EK_{\text{pub}}}\}$$
$$\mathcal{E}_2 \to \mathcal{E}_3 \;\; \textit{Certificate} \leftarrow \text{Enc}_{k_{ch}}[\text{ad}]\{\text{cert}_{CAS_{\text{pub}}}\}$$

SEED treats the TLS channel as *provisional* until the vTPM-side verifier validates *Evidence**. Although the handshake already establishes application-data keys (k_{ap}), *no* application payload is released by either endpoint until the attestation quote carried inside the first encrypted record is checked. The quote thus inherits TLS confidentiality and integrity, and a failed verification forces a fatal alert—preventing any sensitive bytes from crossing a channel whose peer lacks hardware attestation.

Attestation Evidence Exchange. Once authenticated, the vTPM provides attestation evidence—endorsements, reference values, and the attestation report:

$$\mathcal{E}_3 \to \mathcal{E}_2 \;\; \textit{Evidence} \leftarrow \text{Enc}_{k_{sh}}[\text{ad}]\{\texttt{SNP\_REPORT}, \text{VCEK}, \text{ASK}, \text{ARK}\}$$

Verifier Logic. The CAS validates evidence as follows:
- **Certificate Chain.** Check that the *Versioned Chip Endorsement Key* VCEK is signed by AMD's *ASK*, which in turn chains to the AMD *ARK* root.
- **Report Signature.** Verify the PSP-generated MAC/signature on SNP_REPORT with the VCEK.
- **Channel Binding.** Ensure report_data inside the report equals SHA256(protVers $\parallel$ EK_{pub} $\parallel$ n_c) to bind the report to the live TLS session and EK_{pub}.
- **Security Versions.** Compare the PSP and microcode Security Version Numbers (SVNs) against SEED's appraisal policy (revocation list, min-SVNs).
- **Measurement Match.** Confirm that the VM measurement in the report matches the expected hash for the vTPM + TCB image.

If all checks succeed, the CAS marks the peer as *trusted*; the TLS channel transitions from provisional to *active*, and encrypted workload transfer can begin. This procedure guarantees that the vTPM—and by extension the entire SEED TCB—runs inside genuine, up-to-date AMD SEV̄/SNP hardware with an untampered software stack.

6.2 Run: Workload Deployment and Execution

Once the environment is verified, the actual unmodified container images workload is securely loaded and executed (i.e., Launch containers inside a hardware-enforced TEE, minimizing TCB footprint).

6.2.1 Container Key Exchange and Image Decryption. A secure channel must be established between $\mathcal{E}_2$ and $\mathcal{E}_4$ to ensure confidential transmission of the encrypted container image and decryption keys. SEED's CA_3 provides certificates for both components, enabling mutual authentication without external dependencies. This internal trust anchor ensures that only client-approved components participate in the communication, reinforcing C_3 from our trust model.

Following a standard TLS handshake, $\mathcal{E}_2$ and $\mathcal{E}_4$ exchange their CA_3-issued certificates, verify each other's credentials, and execute the *CertificateVerify* steps by signing handshake transcripts, proving private key possession:

$$\mathcal{E}_4 \to \mathcal{E}_2 \;\; \textit{Certificate} \leftarrow \text{Enc}_{k_{sh}}[\text{ad}]\{\text{cert}_{\text{pubCA3-}\mathcal{E}_4}\}$$
$$\mathcal{E}_2 \to \mathcal{E}_4 \;\; \textit{Certificate} \leftarrow \text{Enc}_{k_{ch}}[\text{ad}]\{\text{cert}_{\text{pubCA3-}\mathcal{E}_2}\}$$

$$\mathcal{E}_4 \to \mathcal{E}_2 \;\; \textit{CertificateVerify} \leftarrow \text{Enc}_{k_{sh}}[\text{ad}]\{\text{Sign}_{\text{privCA3-}\mathcal{E}_4}(\text{transcript}_{\mathcal{E}_4})\}$$
$$\mathcal{E}_2 \to \mathcal{E}_4 \;\; \textit{CertificateVerify} \leftarrow \text{Enc}_{k_{ch}}[\text{ad}]\{\text{Sign}_{\text{privCA3-}\mathcal{E}_2}(\text{transcript}_{\mathcal{E}_2})\}$$

After mutual authentication, $\mathcal{E}_4$ provides an attestation report (*Evidence*) to $\mathcal{E}_2$, proving its integrity and that it is running on genuine hardware. $\mathcal{E}_2$ verifies this using CA_2 (AMD CA) and CA_3; similarly, $\mathcal{E}_2$ may send its own evidence if required.

$$\mathcal{E}_4 \to \mathcal{E}_2 \;\; \textit{Evidence} \leftarrow \text{Enc}_{k_{ch}}[\text{ad}]\{\text{AttestationReport}_{\mathcal{E}_4}\}$$

With mutual authentication and attestation complete, a shared secret key k_{shared} is established. Using k_{shared}, $\mathcal{E}_2$ encrypts the container image and transmits it securely to $\mathcal{E}_4$, which can then decrypt and run it using erun, ensuring the workload's confidentiality:

$$\mathcal{E}_2 \to \mathcal{E}_4 \;\; \textit{EncryptedContainer} \leftarrow \text{Enc}_{k_{\text{shared}}}[\text{ad}]\{\text{ContainerImage}\}$$

6.3 Prove: Continuous Runtime Verification

The *push* and *run* stages attest the SEEDCore TCB only once—at VM launch. To preserve trust for the remainder of the VM's lifetime, SEED must continuously demonstrate that (i) the root container image loaded by erun is pristine and (ii) every file subsequently executed inside each container remains unaltered.

SEED implements a **measure-before-execute** policy that captures all security-critical operations in real-time, preventing any window for undetected tampering:

- **Event capture (in–kernel hooks).** All security-sensitive operations—ContainerLoad and FileExec (see §7.1.1)—are *intercepted synchronously* by kernel-level IMA hooks **before** execution proceeds. These hooks:
  - compute h = SHA256(file);
  - **block execution** if the hash doesn't match a pre-approved whitelist (when configured);
  - emit $\langle$CID, path, $h\rangle$ into an in-kernel ring buffer; and
  - extend h into the container-dedicated PCR: $PCR_{\text{CID}} \leftarrow H(PCR_{\text{CID}} \parallel h)$.
- **Quote thread.** A kernel thread wakes every Δt (default 1 s) or when the ring buffer is full to transmit accumulated measurements. It issues `TPM2_Quote` on all changed PCRs using the CAS-supplied nonce, packages $\langle$`QUOTE`, log-delta$\rangle$, and streams it encrypted to the CAS.
  The periodic quotes serve as *checkpoints* rather than detection points—all malicious activity would already have been blocked at load time. The CAS can thus verify the complete execution history: (i) the root image accepted by `erun` matches expectations and (ii) the cumulative hash chain proves every executed binary was measured before running.
- **Secure transport.** The quote bundle travels inside the *same mutually-authenticated, attested TLS session* (Algorithm 1, Stage 3) that established k_{ap}. Thus the verifier checks freshness (nonce), integrity (AIK signature), and provenance (EK chain) *before* releasing further application data, preserving the pipeline's end-to-end security invariants.

Security Analysis. This design prevents the attack scenario where an adversary modifies container state between attestation quotes. Since every executable must pass through the IMA subsystem *before* execution, and the measurement occurs synchronously with mandatory PCR extension, there exists no temporal window for running unmeasured code. Even if an attacker somehow modified files on disk between quotes, those modifications would be caught at the next load attempt.

6.4 Seal: Secure State Persistence

SEED performs *dynamic sealing* to protect any state that must out-live a confidential-VM session. At the end of the attested TLS handshake between the CAS ($\mathcal{E}_2$) and the TCB ($\mathcal{E}_4$) both parties hold the shared session secret K_{TLS}. The TCB invokes an HKDF to obtain a purpose-bound DEK (see 2). Using a fresh context string prevents the raw TLS material from being re-used directly for data encryption. Inside the enclave, the DEK is sealed with `TPM2_Seal` under the vTPM's SRK_{priv} and a PCR policy that captures the TCB's trusted measurements. The call returns *only the opaque blob* B_{DEK}; the plaintext DEK never leaves the enclave.

The workload ($\mathcal{E}_5$) encrypts its artefacts D with the DEK, yielding $C_D = \text{Enc}_{\text{DEK}}(D)$. Both C_D and B_{DEK} are written to untrusted, persistent storage (e.g. object storage or returned to the client). Compromise of that store reveals no secrets.

As shown in algorithm 2, when the workload terminates, the enclave zeroises its memory; DEK and all derived keys vanish. What the client *regains* is sole possession of B_{DEK}, not the raw key. Upon redeployment the client supplies B_{DEK} to a freshly booted TCB and vTPM. After a new attested TLS session and PCR verification, the vTPM executes `TPM2_Unseal` which succeeds *only* if (i) the same SRK pair is present and (ii) the current PCR values match the sealing policy. The recovered DEK then decrypts C_D, restoring application state transparently. A PCR mismatch—or execution on hardware lacking the correct SRK—causes unsealing to fail, rendering the ciphertext permanently opaque.

Algorithm 2 Sealing & Restoration Protocol

Require: Actors: Client (**C**), TCB & vTPM (inside Server, **S**)
Ensure: Confidential, integrity-protected snapshot that can be resumed only in a re-attested TCB

procedure Seal(**S**, **C**)
  /* **pre-condition: attested TLS completed; both hold** K_{TLS} */
  S: DEK $\leftarrow$ HKDF(K_{TLS}, "seed for container data")
  S: $C_D \leftarrow \text{Enc}_{\text{DEK}}[\text{ad}]\{D\}$ ▷ Encrypt workload data
  S: $B_{\text{DEK}} \leftarrow$ TPM2_Seal(SRK_{priv}, PCRs$_{\text{TCB}}$, DEK)
  S $\rightarrow$ **C**: *Store or return* (C_D, B_{DEK})
  S: **zeroise** DEK ; **power off**

procedure Unseal(**C**, **S**)
  /* **new TCB instance, fresh attested TLS in place** */
  C $\rightarrow$ **S**: (C_D, B_{DEK})
  S: DEK $\leftarrow$ TPM2_Unseal(SRK_{priv}, PCRs$_{\text{TCB}}$, B_{DEK})
  if unseal **fails then**
    S: **abort** ▷ Wrong SRK or PCRs $\Rightarrow$ data stays locked
  else
    S: $D \leftarrow \text{Dec}_{\text{DEK}}[\text{ad}]\{C_D\}$ ▷ Restore state
    continue execution

7 Implementation

Implementing SEED involved overcoming challenges such as reducing the TCB while maintain trusted and confidentiality. SEED achieves its 22MB TCB through five key optimizations: (i) collapsing the container runtime into kernel space via static linking, (ii) eliminating multi-tenant features unnecessary in single-tenant TEEs, (iii) systematically removing 45% of crun's codebase (cgroups, eBPF, seccomp), (iv) replacing heavyweight libraries with minimal alternatives (700+ kLoC reduction), and (v) using static compilation to eliminate dynamic loading overhead. This section details the decisions made to address these issues.

7.1 vTPM and IMA Implementation

TPM core. SEED's vTPM is built with `libtpms` and follows the TPM 2.0specification [66, 67, 68]. It runs inside the SEVSNP SecureVM Service Module (SVSM) at VMPL0 [49]. The SVSM environment is bare-metal—no syscalls, timers, or libc—so we statically link only the required cryptography and replace OS-dependent calls with thin wrappers that use PSP GHCB services (e.g. monotonic counter, wall clock).

Entropy source. Key generation relies on the on-die RNG exposed via `RDRAND/RDSEED` on Zen 3+. These instructions are serviced by the PSP, so the entropy never leaves the package, preventing a malicious hypervisor from influencing or observing the seed [49].

Command buffer isolation. SEEDCore/vTPM traffic uses a single shared Guest-Hypervisor Communication Block (GHCB) page marked *Shared* in the RMP. Only VMPL0 vTPM and VMPL1 (guest

kernel) can touch the page; any host or VMPL> 1 access raises an RMP fault. SEV's inline AES-XTS engine encrypts the page transparently, while the PSP integrity-tree blocks replay, ensuring that vTPM state and IMA measurements remain confidential and tamper-evident throughout VM lifecycle events.

VMPL/RMP protection of PCRs. Because the vTPM's PCR bank resides in VMPL0-private pages, neither the cloud operator (hypervisor) nor SEEDCore (VMPL1) can read, reset, or roll back PCR values [65]. Thus, any attempt to subvert runtime measurements would need the sealed AIK private key—also locked behind VMPL0 isolation.

Measurement Isolation Guarantees. Pages holding the vTPM's code, sealed state, and PCR bank are marked *private* & VMPL0 in the RMP; hypervisor access triggers an RMP fault. SEEDCore executes at VMPL1; any VMPL1 read/write to VMPL0 pages also raises an RMP fault, ensuring the guest itself cannot tamper with PCRs or the AIK key. If a malicious container is introduced, its code is still hashed and extended into the IMA PCRs. The resulting quote diverges from policy and is detected by the CAS; forging or suppressing PCR updates would require the AIK privkey or direct PCR writes—both infeasible without VMPL0 access.

7.1.1 Container Promotion & Per-Container PCRs. By collapsing `erun`, IMA, and a lightweight LSM shim into one address-space image, SEEDCore removes the classic user/kernel split.

During ContainerLoad `erun`:

- allocates a `seed_container_t` descriptor,
- stores its pointer in `task→cred→security`, and
- reserves a dedicated PCR for that container's runtime measurements inside the vTPM.

IMA hooks with `CID` *context.* Existing hooks (`bprm_check_security`, `file_mmap(PROT_EXEC)`, `do_execve`, ...) inherit the container identifier (`CID`).

7.2 erun implementation

The design of erun begins with a *minimal, specification-conformant subset* of the `crun` code base [20]. All functionality that is superfluous in a single-tenant TEE—namely Linux Security Module hooks (seccomp, AppArmor, SELinux, keyrings), checkpoint/restore facilities (CRIU), resource-control subsystems (cgroups, eBPF, I/O priorities), and terminal handling—was eliminated. These mechanisms are valuable in multi-tenant settings, yet they introduce considerable attack surface inside an enclave whose boundary is already enforced by the TEE itself. The retained abstractions are limited to mount, PID, and network namespaces so that OCI containers continue to observe the expected isolation semantics.

Subsequently, this reduced runtime was **ported to a statically linked, kernel-resident environment**. A lightly patched Linux 6.8 and a size-optimised `glibc` are linked with erun, yielding a single 22 MB ELF in the style of Unikernel Linux (UKL) [58, 57]. Static linkage removes the dynamic loader, shared-library resolution, and user-space system-call trampolines while *preserving forward and binary compatibility* with conventional Linux binaries. The resulting unikernel-scale TCB boots within milliseconds, eliminates the pre-enclave exposure window, and provides the substrate for the IMA-namespaced, vTPM-bound, and attested-TLS subsystems that follow.

7.2.1 Building SEED's Toolchain. A unikernel–scale TCB keeps SEED verifiable, but it also removes the rich user-space that conventional container workflows assume. Vanilla `crun` can still launch a root file-system, yet it cannot *pull*, reshape, or lifecycle–manage images by itself. We therefore crafted a bespoke tool-chain that couples [58, 57] with a heavily pruned erun, restoring full OCI semantics without re-inflating the TCB.

OCI compatibility and orchestration. erun remains byte-for-byte compliant with the OCI Runtime and Image specifications, enabling transparent use beneath `containerd` [11], the Moby Engine [46, 47], and Kubernetes [54].

We evaluated three integration paths:

- a C-based gRPC client statically linked into UKL,
- an out-of-enclave Go proxy that terminates gRPC and forwards flattened messages, and
- a tiny C-native orchestrator embedded in UKL.

Option (ii) is the default: it leaves the Go stack outside the trusted boundary while adding only a 36 kB proxy inside.

Image-handling utilities. Container creation still needs *Buildah*, *Skopeo*, and *Umoci*. Because the reference implementations are written in Go—whose runtime, garbage collector, and reflection are unsupported in UKL [21]—we split responsibility: Buildah runs during CI outside the enclave [12], while Skopeo [13] and Umoci [71] must execute inside the TCB. We therefore re-implemented the Registry V2 pull path [16] and bundle conversion [50, 52] in 4.8 kLoC of ANSI C, streaming layers directly into OverlayFS.

Lean support libraries. To avoid dragging in `glibc`-scale baggage we curated BusyBox-style components: a 4 kLoC HTTP client replaces `curl`, a hand-rolled JSON parser supplants `cJSON`, and a trimmed `libarchive` covers compression duties. Together these cuts shed more than 700 kLoC of external code.

Shared-buffer optimisation. At boot the launcher creates a single 64 MiB page-aligned `memfd`, maps it `MAP_SHARED` across all enclave processes, and exports the file descriptor via environment variables `ERUN_SHM_FD` and `ERUN_SHM_SIZE`. A 20-line bump allocator carved from this region replaces `jemalloc`, eliminating heap metadata and steady-state `mmap`/`brk`. Because AMD SEV encrypts memory per-VM, the pool remains opaque to the hypervisor yet delivers end-to-end zero-copy DMA when registered with `io_uring` and `vhost-user`.

Targeted code reductions. A line-by-line audit of `crun` excised feature that added risk but no value inside a single-tenant TEE:

- **Archive path:** multi-process `tar` replaced by in-process `CPIO+gzip`, slicing peak RAM by 68 % and unpacking a 240 MB Ubuntu image in 412 ms (vs. 1.2 s);
- **Security modules:** `-DTEE_OPTIMIZED` stubs out seccomp, AppArmor, SELinux, capability dropping, and CRIU;
- **Namespaces:** keep mount and optional PID/network; drop user, cgroup, IPC, and UTS namespaces;
- **Cgroups, hooks, checkpoint/restore, state-root:** removed; resource policing is delegated to the hypervisor.

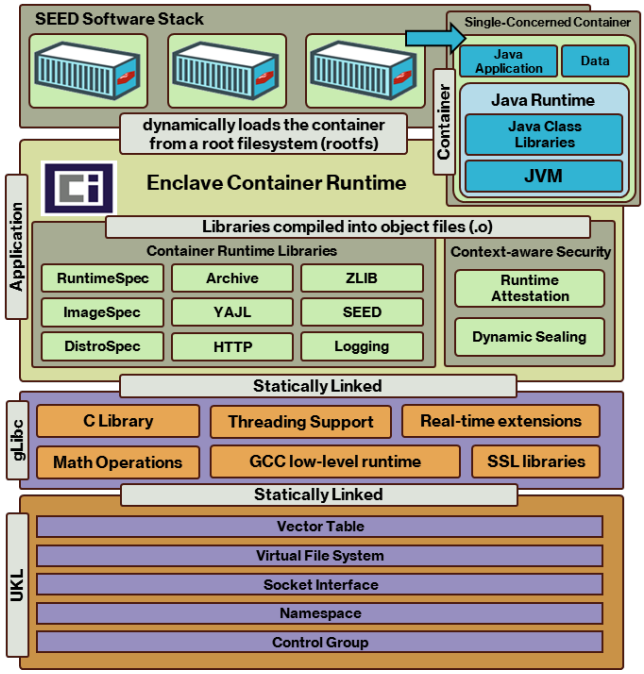


Figure 3: SEED software stack that to confidential and isolation inside the TCB

The final statically linked binary, including the C puller, weighs 1.7 MB (stripped) and lowers SEED's kernel-space TCB to ~220 kLoC—small enough for formal verification yet fully interoperable with mainstream container ecosystems [8, 75, 4].

7.3 OS-Layer Optimization

To reduce the TCB size and enhance both efficiency and security without sacrificing general-purpose capabilities, we adopted a hybrid strategy. Our approach combines specialized kernel configurations with the integration of UKL [57, 58], as shown in Figure 3.

Building on insights from Lupine [36], which minimized kernel features for cloud applications, we tailored the Linux kernel for container runtimes. We removed unnecessary functionality—such as single-machine administration interfaces, broad hardware abstractions, and redundant synchronization or scheduling mechanisms—thereby reducing system call overhead. Guided by a heuristic methodology, we refined UKL to support the top 20 most downloaded Docker Hub applications, significantly decreasing the TCB while preserving essential services.

However, customizing the OS layer often requires substantial kernel modifications, risking loss of binary compatibility and complicating broader adoption. Prior efforts, including NetBSD's Rump Kernel, Windows' Drawbridge, and Linux's LKL, extended kernel functionality but also struggled with maintainability and compatibility [59, 5, 55]. Similarly, while Lupine and X-Containers improved performance for specific scenarios, they faced challenges in preserving forward compatibility [36].

7.4 Linking erun to the Kernel

We address compatibility challenges posed by the architectural differences involved in running the container runtime directly within kernel space. Additionally, we refined our compilation strategies to enhance workload deployment. This section details the technical enhancements.

Running the runtime entirely in kernel mode imposes two key changes. **(i) Minimal libc.** Unikernel Linux (UKL) ships only a skeletal `libc`. Rather than restore a full user-space ABI, we remove or inline most system calls; the few still required—e.g. `setns` and `unshare` for namespace management—are invoked directly via`syscall(SYS_setns,...)` and `syscall(SYS_unshare,...)`. This approach preserves OCI namespace semantics while avoiding the code and attack surface of a full glibc, with erun providing the error handling normally hidden inside `libc` wrappers. **(ii) Kernel-level services.** TLS hand-shake logic, vTPM calls, and IMA namespace hooks are compiled directly into the supervisor binary, eliminating context switches and userspace helpers.

Static–link build pipeline. First, every C unit in `src/` (erun proper), the pruned `crun` glue, and a handful of support libraries (custom HTTP/JSON, `libarchive`, `zlib`) is compiled into position-independent objects. UKL's stripped `libc`, `libpthread`, `librt`, `libm`, OpenSSL (`libssl`/`libcrypto`), `libcurl` and `libgcc` are built *statically* with the same kernel-friendly flags (`-static -mcmodel=kernel -mno-red-zone`). The linker then performs a two-stage merge (the process implemented in UKL [58]):

- **relocatable link** (`ld -r`) combines all objects, start-up stubs (`crt1.o`, `crti.o`, `crtbeginT.o`) and termination stubs (`crtend.o`, `crtn.o`) into `temp.o`, using `–whole-archive` to pull every symbol but avoiding duplicate definitions;
- **archiving** packs `temp.o` into `UKL.a`, after which `objcopy` prefixes every global symbol with `seedcore_` and applies a small renaming map to prevent clashes with kernel symbols.

At kernel build time the Linux image links against `UKL.a`; the resulting `vmlinuz` therefore boots with erun, OpenSSL, vTPM stubs, and the IMA namespace already resident in ring 0. Because all code is immutable and address-space-layout-randomisation (ASLR) is disabled, the measured hash loaded into PCR 0 at boot uniquely identifies the entire TCB, simplifying remote attestation.

8 Security Evaluation

This section evaluates how SEEDCore achieves privacy through end-to-end confidentiality and integrity. Our analysis addresses different privacy dimensions against 11 attacks (**A1**–**A11**) from [28].

8.1 Privacy Analysis

Privacy prevents inference about *what* is running, *who* is using it, and *how* it operates. We examine three privacy dimensions:

Workload privacy. SEED achieves workload privacy by eliminating the pre-enclave window via post-attestation dynamic loading—admitting workloads only after verification. By contrast, Unikernels, though they place a statically linked workload+kernel as a single ELF entirely inside the TEE, still lack end-to-end privacy: during deployment the provider can see which specialized image is being loaded before protections engage; likewise, conventional TEE

Table 2: Resistance to an *outside* attacker. SEEDCore inherits AMD SEV-SNP; Gramine-TDX relies on Intel TDX.

#	Attacker move	SEEDCore (single container)	Gramine-TDX (single process)	Why SEEDCore wins for micro-services
1	Inject / swap layers post-attestation (A3 & A4)	✅ attested-TLS puller; every layer re-measured into *per-container* PCR ⇒ quote fails	❌ TD never re-pulls; host can replace whole virtio-fs tree; only manifest is covered	Supports late, *on-demand* OCI pulls without breaking integrity
2	Add a second, malicious container into the same VM (A1 & A6)	✅ new namespace set *and* separate PCR stream; CAS can withhold secrets only from the bad pod	❌ one process ⟶ new TD per side-car	Side-cars / service mesh stay in one VM
3	Incrementally corrupt files in a running container (A2:E9–E12)	✅ erun ⟶ IMA streams deltas; CAS detects <1 s	❌ no runtime measurement; tampering is silent until reboot	Continuous *run⟶prove* fits rolling updates
4	Roll back an old but signed layer (A3:E14)	✅ layer MAC binds freshness-nonce inside TLS Finished	⛔ files bound to measurement, not time	Freshness enforced by CAS, not operator discipline
5	Sniff/alter secrets in flight (A1:E1, E2)	✅ RA-TLS baked into pull path; host sees only ciphertext	⛔ RA-TLS helpers exist, but developer must use them; vsock leg plaintext on host	No extra work for app authors / DevOps
6	Abuse K8s control-plane knobs (A1:E6, A6:E22–E30)	⛔ dangerous knobs become no-ops under `TEE_OPTIMIZED` kernel build	✅ black-box VM; spec cannot affect policy	Keeps full pod-spec expressiveness for benign cases
7	Hypervisor reads plaintext pages (A10:E42)	✅ SEV-SNP AES-XTS + VMPL / RMP	✅ TDX AES-XTS + Secure-EPT	parity
8	Hypervisor re-maps or replays pages (A8:E37)	✅ RMP + vTPM quote	✅ Secure-EPT + RTMRs	parity
9	Crash container daemon with bogus requests (A4:E19)	✅ no user-space daemon; in-kernel ioctl per pod	❌ no runtime	removal beats hardening
10	Exploit kernel subsystems inside enclave (A9, A7)	⛔ eBPF/cgroups stripped, but minimal TCP/IP retained for pod networking	✅ no net-stack, no namespaces	Containers need a net-stack; SEED keeps it with reduced code size

containerizations that launch images pre-attestation leak workload identity across build/pull/launch.

Query and Input Privacy. SEED preserves query/input privacy by enveloping the entire container stack within the TCB (zero operational visibility) and using a kernel-integrated runtime that removes syscall visibility, preventing the hypervisor from distinguishing query types or inferring input features. In contrast, LibOS-based TEEs (Gramine-SGX [23], SCONE [2], Occlum [63]) lack container-native primitives (namespaces, OverlayFS) and push image/layer operations to the untrusted host; the host can observe pulls, layer diffs, and runtime telemetry—syscall sequences, memory-access rates, I/O mix—that fingerprint queries and reveal inputs (e.g., patient-scan inference vs. transaction-fraud analysis), while the narrow trusted/untrusted interface enables Iago and side channels.

Metadata and Co-residency Inference. SEED's limits disclosure to the requesting container's measurements plus shared dependencies, masking other containers' PCRs. This cuts metadata exposure by reducing the file hash to ≈200–500 container-specific file hashes vs 10,000–50,000 system-wide. Classic IMA discloses the node-wide measurement list to verifiers [61], revealing all running workloads and enabling co-residency inference (business relationships, usage patterns, deployment scale).

8.2 Security Analysis Against External Attacks

For each exploit family listed in the survey we ask: *How far can an outside attacker get before the TEE detects, confines, or prevents the action. A coarse rating scheme highlights the result (✅ blocked, ⛔ confined/detectable, ❌ still possible).* Table 2 summarises the outcome; below explains the differences.

Post-attestation tampering (Attacks #1, #3, #4) These attacks not only compromise integrity but enable workload identification through tampered files. SEEDCore prevents both through: (i) attested-TLS termination inside the TEE for all registry pulls, (ii) continuous IMA streaming measuring every file operation into per-container PCRs with sub-second validation, and (iii) freshness nonces preventing rollback. Gramine-TDX's single boot-time measurement cannot detect runtime tampering. *Privacy protected: Workload privacy & Metadata/co-residency.*

Container isolation (Attacks #2, #6) Multi-container microservices risk cross-workload inference. SEEDCore enforces separate namespace sets and PCR chains per container, preventing one container from inferring another's identity or behavior. The CAS can selectively distribute secrets while maintaining isolation. Gramine-TDX's single-process model requires separate TDs per container, ironically improving isolation but breaking service meshes and increasing observable deployment patterns. *Privacy protected:* Workload privacy & Metadata/co-residency.

Network security (Attack #5) Network patterns reveal workload behavior—API calls, model downloads, and communication patterns fingerprint applications. SEEDCore bakes RA-TLS into the image pull path with attestation-bound keys, encrypting all observable patterns. Gramine-TDX's plaintext vsock leg between host and TD leaks config data and connection metadata. *Privacy protected:* Query & Input privacy & (supporting) Metadata/co-residency.

Memory protection (Attacks #7, #8) Both platforms encrypt memory (SEEDCore via SEV-SNP AES-XTS, Gramine-TDX via Intel TDX Secure-EPT). However, access patterns remain vulnerable to cache-timing analysis—an active research area for both systems. *Privacy protected:* Workload privacy (contents hidden); residual side-channels discussed separately.

Runtime integrity (Attacks #9, #10) SEEDCore retains minimal TCP/IP for pod networking while stripping privacy-violating functions like eBPF (which exposes execution traces) and cgroups (which leak resource patterns). This trades slightly more code for compatibility with containerized microservices while maintaining operational privacy. *Privacy protected:* Query & Input privacy.

SEEDCore addresses privacy in container-native clouds, it adds comprehensive privacy protection against workload inference, query analysis, and metadata leakage—critical for protecting ML models, algorithms, and business logic in multi-tenant clouds.

Formal verification feasibility. SEED's reduced codebase suggests formal verification of security-critical components may be feasible. The `erun` runtime (≈14,708 C LOC) exceeds fully verified kernels like seL4 (≈8,700 LOC) and CertiKOS (≈6,500 LOC), as well as the 10k LOC threshold [33, 24, 26], however, `erun` modular design puts verification within reach, container library is (≈3,700 LOC) only.

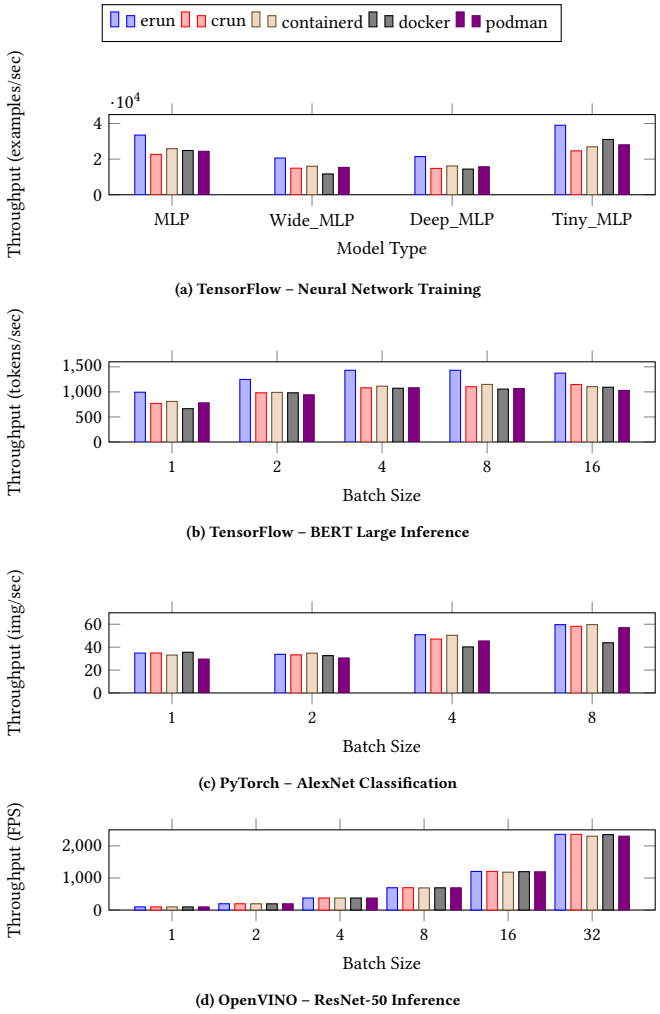


Figure 4: Performance of ML confidential workloads across different container runtimes and virtualization technologies. Each subplot compares the performance of different container runtimes (erun, crun, containerd, docker, podman) across various ML workloads: (a) TensorFlow Neural Network training throughput in examples/sec, (b) TensorFlow BERT Large inference throughput in tokens/sec, (c) PyTorch AlexNet image classification throughput in img/sec, (d) OpenVINO ResNet-50 inference throughput in FPS.

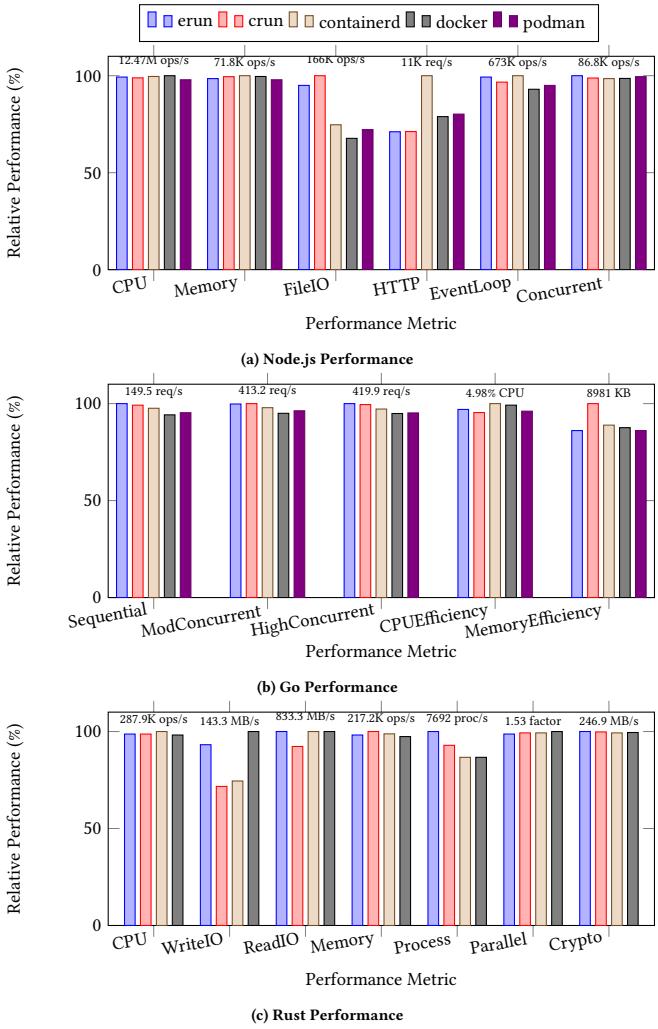
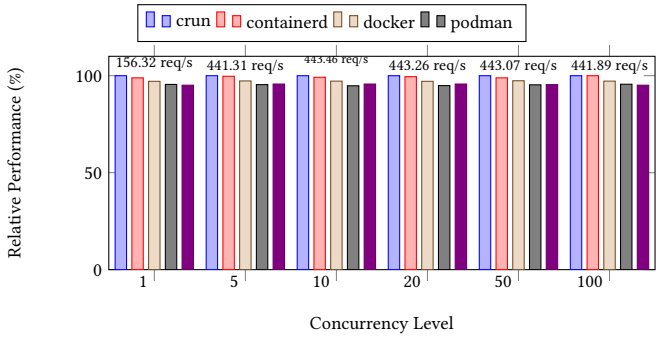


Figure 5: Runtime micro-benchmarks. (a) Node.js — Docker wins CPU tasks, containerd leads in HTTP + memory, crun dominates file I/O. (b) Go — erun tops sequential & high-concurrency throughput, crun excels at moderate concurrency & memory use, containerd is most CPU-efficient. (c) Rust — Docker leads CPU workloads, Podman excels at disk writes and parallelism, crun outperforms in fork + crypto, containerd delivers the best memory throughput.

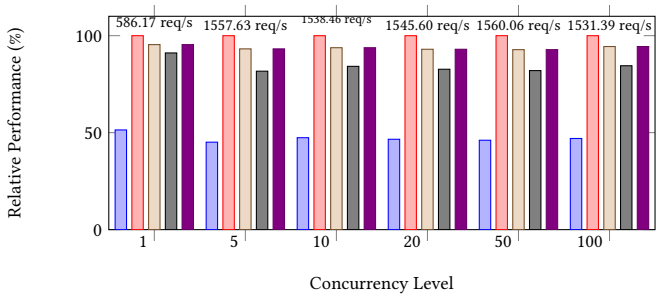
9 Performance Evaluation

Our evaluation examines the performance characteristics of our framework across diverse workloads. We assess both the performance of eRun within confidential VMs and the fundamental capabilities of seedcore. Our experiments focus on real-world applications spanning machine learning (ML) frameworks, programming language environments, and web servers to provide a comprehensive understanding of performance trade-offs in confidential computing scenarios. We present architecture-consistent comparisons first, followed by cross-architecture microbenchmarks that serve as indicative references rather than direct comparisons. ML workloads represent the most privacy-critical use cases for confidential computing. With the rise of LLMs and AI agents processing sensitive personal/corporate data, ensuring model confidentiality (preventing IP theft) and inference privacy (protecting user queries) has become paramount. ML training on datasets exemplifies workloads where both data and model privacy are essential.

Testbed All experiments were conducted on an *Azure Confidential VM (Standard DC4as v5, 4 vCPUs, 16 GiB RAM)* hosted in the East US. This instance is backed by a 3rd-generation *AMD EPYC 7763* processor with *SEV-SNP* enabled. The guest operating system is *Ubuntu 22.04 LTS* (kernel 6.8, Canonical *confidential-vm* build), and the system disk resides on encrypted Premium LRS NVMe storage. For each benchmark we execute three independent runs and report the arithmetic mean to smooth out minor runtime variance. Boot-time logs confirm that both SEV-SNP memory encryption and Azure Secure Boot are active. Our evaluation explores two dimensions to assess SEED: **Execution environments:** We compare (1) Ubuntu VM as the standard baseline, (2) SEEDCore, our minimal TCB with kernel integration, (3) Gramine-SGX, and (4)



(a) NGINX Performance Across Container Runtimes



(b) Apache httpd Performance Across Container Runtimes

Figure 6: Web-server throughput. (a) NGINX: erun is fastest at every concurrency level (crun ties at 100 conns). Containerd sustains 97 % of peak; Docker and Podman 95 %. (b) Apache httpd: crun dominates; containerd and Podman reach 93 %, Docker 82–91 %, erun 45–51 %. Note: httpd delivers 3–4× NGINX throughput under identical loads.

Gramine-TDX as state-of-the-art LibOS. This selection spans from Ubuntu through LibOS partitioning (Gramine) to our unified design, enabling direct comparison of TCB size versus functionality trade-offs across process and VM-level TEE architectures.

We evaluate (1) erun (2) crun as the lightweight reference, (3) containerd as Kubernetes' standard supervisor, (4) Docker as the most widely-deployed platform, and (5) podman as the daemonless enterprise alternative. This range from minimal (crun) to feature-rich (Docker) demonstrates that SEED maintains compatibility across the entire container ecosystem while adding hardware-rooted attestation. This multi-dimensional evaluation validates both TCB optimization approaches and runtime overhead.

9.1 Runtime Performance

We evaluate the performance of five container runtimes—*erun*, *crun*, *containerd*, *docker*, and *podman*—across four ML workloads.

9.1.1 ML Workloads. We benchmark Node.js, Go, and Rust on CPU, memory, file-I/O, HTTP, and crypto inside an SEV-SNP VM; Fig. 5 normalises throughput to per-test best. *Node.js:* erun reaches 98% on CPU/event-loop; containerd leads HTTP; Docker lags on file-I/O (67–79%). *Go:* erun leads single-thread and high-concurrency; containerd ties on CPU-only. *Rust:* Docker leads raw CPU, `crun` fork+crypto, Podman disk writes, containerd memory ops; erun stays within 5–15%.

9.1.2 Programming-Language Benchmarks. Using Node.js, Go, and Rust, we benchmark CPU, memory, file-I/O, HTTP, and crypto inside an SEV-SNP VM; Fig. 5 normalises throughput to the per-test best. *Node.js:* erun hits 98% of the leader on CPU/event-loop; containerd leads HTTP; Docker lags on file-I/O (67–79%). *Go:* erun leads single-thread and high-concurrency; containerd ties on CPU-only. *Rust:* Docker leads raw CPU, `crun` fork,+,crypto, Podman disk writes, containerd memory ops; erun stays within 5–15% of the best. Overall, erun's overhead is negligible, in-kernel deployment retains near-native while OCI-compatible.

Benchmark	Ubuntu VM	SEEDCore	Gramine-SGX	Gramine-TDX
Dhrystone	16.4M lps	46.7M lps (2.84×)	331M lps*	332M lps*
Whetstone	881 MFLOPS	1150 MFLOPS (1.31×)	5006 MFLOPS*	4987 MFLOPS*
Arithmetic	4.26M lps	738M lps (173×)	841M lps*	798M lps*
Pipe	705K lps	3726K lps (5.28×)	330K lps*	1620K lps*
Recursion	207K lps	447K lps (2.16×)	7.7M lps*	8.1M lps*
Switching	81K lps	2 lps (0.00002×)	N/A	N/A
System Call	378K lps	*Failed*	12.7M lps*	9.3M lps*
File Copy	1091K KBps	*Failed*	15–228K KBps*	10–137K KBps*
Spawn	785 lps	*Failed*	N/A	N/A
Execl	*Failed*	*Failed*	*Failed*	0.8K lps*
TCB Size	690 MB	**23.4 MB**	1.3 – 1.5 MB	1.2 MB

Table 3: Comparison of `UnixBench` performance across different execution environments *SGX/TDX measurements use Intel Xeon processors (E-2288G for SGX, 4th-gen "Sapphire Rapids" for TDX) while SEEDCore/Ubuntu use AMD EPYC—see section text for interpretation guidance.

9.1.3 Web Server Performance. We evaluate NGINX (event-driven) and Apache httpd (process-based) across runtimes at concurrency 1–100. Figure 6 shows normalized throughput. *NGINX.* erun leads at all concurrency levels; crun achieves 99–100% (matching at 100 connections); containerd 97%; Docker and Podman 95%. Throughput plateaus at 442 req/s (≥5 connections), indicating runtime—not NGINX—limits performance. *Apache httpd.* crun dominates; containerd and Podman reach 93–95%; Docker 82–91%; erun only 45–51%. Peak 1560 req/s (3.5× NGINX) stabilizes after 5 connections. The servers expose opposite characteristics: erun excels with NGINX but struggles with multi-process Apache; crun shows the reverse.

9.1.4 Cross-Architecture Microbenchmarks. We compare AMD EPYC (*SEEDCore*, Ubuntu VM) with Intel Xeon (*Gramine-SGX*, *Gramine-TDX*). Due to architectural differences, results are indicative—compare *within* platforms, not across; for like-for-like hardware, see prior SEV-SNP sections. Using *UnixBench*, we evaluate all four configurations; scores appear in Table 3. Key observations: *SEEDCore* delivers strong CPU performance (arithmetic, Dhrystone) with full OS functionality. Gramine variants excel at in-memory tasks but degrade on I/O—*Gramine-TDX* reaches only 5–6% of native on filesystem tests (simple `virtio-fs`); *Gramine-SGX* shows low pipe throughput due to encrypted host–kernel pipes, while *Gramine-TDX* retains pipe performance since pipes remain within the VM. Overall, *SEEDCore* preserves functionality with strong compute on AMD, while Gramine on Intel exhibits higher I/O with workload trade-offs.

10 Concluding Remarks

SEED embeds the *erun* container runtime in a minimal TCB, improving privacy and efficiency while shrinking the attack surface for scale. Leveraging TEE trust assumptions, SEED tunes the TCB to harden security without sacrificing practicality.

Acknowledgments

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

The authors used generative AI-based tools to revise the text, improve flow and correct any typos, grammatical errors, and awkward phrasing.

References

[1] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. 2023. Veil: A Protected Services Framework for Confidential Virtual Machines. en. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ACM, Vancouver BC Canada, (Mar. 2023), 378–393. ISBN: 9798400703942. DOI: 10.1145/3623278.3624763.

[2] Sergei Arnautov et al. 2016. {SCONE}: Secure linux containers with intel {SGX}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 689–703. Retrieved May 8, 2024 from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[3] aws. 2024. AWS Key Management Service - AWS Key Management Service. (2024). Retrieved Oct. 6, 2024 from https://docs.aws.amazon.com/kms/latest/developerguide/overview.html.

[4] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices architecture enables devops: Migration to a cloud-native architecture. *Ieee Software*, 33, 3, 42–52. Publisher: IEEE. Retrieved May 8, 2024 from https://ieeexplore.ieee.org/abstract/document/7436659/.

[5] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. en. *ACM Transactions on Computer Systems*, 33, 3, (Sept. 2015), 1–26. DOI: 10.1145/2799647.

[6] Christian Brauner. 2020. Seccomp Notify - New Frontiers in Unprivileged Container Development. (July 2020). Retrieved May 22, 2024 from https://people.kernel.org/brauner/the-seccomp-notifier-new-frontiers-in-unprivileged-container-development.

[7] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: why the system call API is a bad untrusted RPC interface. *SIGARCH Comput. Archit. News*, 41, 1, (Mar. 2013), 253–264. DOI: 10.1145/2490301.2451145.

[8] Lianping Chen. 2018. Microservices: architecting for continuous delivery and DevOps. In *2018 IEEE International conference on software architecture (ICSA)*. IEEE, 39–397. Retrieved May 11, 2024 from https://ieeexplore.ieee.org/abstract/document/8417115/.

[9] Yueqiang Cheng, Zhi Zhang, Yansong Gao, Zhaofeng Chen, Shengjian Guo, Qifei Zhang, Rui Mei, Surya Nepal, and Yang Xiang. 2022. Meltdown-type attacks are still feasible in the wall of kernel page-Table isolation. *Computers & Security*, 113, 102556. Publisher: Elsevier. Retrieved May 8, 2024 from https://www.sciencedirect.com/science/article/pii/S0167404821003801.

[10] 2024. Confidential-containers/trustee. original-date: 2022-04-25T12:45:02Z. (Oct. 2024). Retrieved Oct. 6, 2024 from https://github.com/confidential-containers/trustee.

[11] 2024. Containerd/containerd. original-date: 2015-11-13T00:27:43Z. (Apr. 2024). Retrieved Apr. 15, 2024 from https://github.com/containerd/containerd.

[12] 2024. Containers/buildah. original-date: 2017-01-26T16:59:13Z. (June 2024). Retrieved June 2, 2024 from https://github.com/containers/buildah.

[13] 2024. Containers/skopeo. original-date: 2016-03-07T20:23:29Z. (June 2024). Retrieved June 2, 2024 from https://github.com/containers/skopeo.

[14] Jonathan Corbet. 2018. Containers from user space [LWN.net]. (Jan. 2018). Retrieved May 22, 2024 from https://lwn.net/Articles/745820/.

[15] Victor Costan. 2016. Intel SGX explained. *IACR Cryptol, EPrint Arch.* Retrieved Nov. 23, 2024 from https://people.cs.rutgers.edu/~santosh.nagarakatte/cs544/readings/costan-sgx.pdf.

[16] Docker. 2024. HTTP API V2 | Docker Documentation. (2024). Retrieved June 7, 2024 from https://docker-docs.uclv.cu/registry/spec/api/.

[17] Anna Galanou, Khushboo Bindlish, Luca Preibsch, Yvonne-Anne Pignolet, Christof Fetzer, and Rüdiger Kapitza. 2023. Trustworthy confidential virtual machines for the masses. en. In *Proceedings of the 24th International Middleware Conference on ZZZ*. ACM, Bologna Italy, (Nov. 2023), 316–328. ISBN: 9798400701771. DOI: 10.1145/3590140.3629124.

[18] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. 2022. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. en. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Los Angeles CA USA, (Nov. 2022), 1231–1242. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560592.

[19] Tim Geppert, Stefan Deml, David Sturzenegger, and Nico Ebert. 2022. Trusted execution environments: Applications and organizational challenges. *Frontiers in Computer Science*, 4, 930741. Publisher: Frontiers Media SA. Retrieved Dec. 19, 2024 from https://www.frontiersin.org/articles/10.3389/fcomp.2022.930741/full.

[20] Scrivano Giuseppe\. 2024. Containers/crun. original-date: 2017-09-13T20:20:58Z. (May 2024). Retrieved May 26, 2024 from https://github.com/containers/crun.

[21] google. 2024. Golang - Google Search. (2024). Retrieved June 7, 2024 from https://go.dev/.

[22] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache Attacks on Intel SGX. en. In *Proceedings of the 10th European Workshop on Systems Security*. ACM, Belgrade Serbia, (Apr. 2017), 1–6. ISBN: 978-1-4503-4935-2. DOI: 10.1145/3065913.3065915.

[23] [n. d.] Gramine. (). Retrieved May 8, 2024 from https://gramineproject.io/.

[24] Ronghui Gu. [n. d.] CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. (). Retrieved Aug. 25, 2025 from https://www.usenix.org/system/files/conference/osdi16/osdi16-gu.pdf.

[25] Asier Gutierrez. 2023. IMA Namespaces for Containers - Asier Gutierrez, Huawei - YouTube. (2023). Retrieved May 11, 2024 from https://www.youtube.com/watch?v=bs7EcuF4SSo&t=8s&ab_channel=TheLinuxFoundation.

[26] Gernot Heiser. [n. d.] The seL4 Microkernel – An Introduction. (). Retrieved Aug. 25, 2025 from https://sel4.org/About/seL4-whitepaper.pdf?utm_source=chatgpt.com.

[27] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205. Retrieved June 5, 2024 from https://ieeexplore.ieee.org/abstract/document/6547110/.

[28] Omar Jarkas, Ryan Ko, Naipeng Dong, and Redowan Mahmud. 2025. A Container Security Survey: Exploits, Attacks, and Defenses. *ACM Comput. Surv.*, 57, 7, (Feb. 2025), 170:1–170:36. DOI: 10.1145/3715001.

[29] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. 2022. {HyperEnclave}: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 437–454. Retrieved Apr. 22, 2025 from https://www.usenix.org/conference/atc22/presentation/jia-yuekai.

[30] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper*, 13. Retrieved May 8, 2024 from https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf.

[31] Nguyen Khang T. [n. d.] Introduction to Cache Allocation Technology in the Intel® Xeon®... en. (). Retrieved May 29, 2025 from https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html.

[32] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. {OSv—Optimizing} the Operating System for Virtual Machines. In *2014 usenix annual technical conference (usenix atc 14)*, 61–72. Retrieved May 8, 2024 from https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[33] Klein. [n. d.] seL4: Formal Verification of an Operating-System Kernel. (). Retrieved Aug. 25, 2025 from https://read.seas.harvard.edu/~kohler/class/cs260r-17/klein10sel4.pdf?utm_source=chatgpt.com.

[34] Paul Kocher et al. 2020. Spectre attacks: exploiting speculative execution. en. *Communications of the ACM*, 63, 7, (June 2020), 93–101. DOI: 10.1145/3399742.

[35] T. Kubernetes. 2019. Kubernetes. *Kubernetes. Retrieved May*, 24, 2019. Retrieved July 2, 2024 from https://dzone.com/storage/attachments/14131598-dzone-kubernetes-bundle.pdf.

[36] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in unikernel clothing. en. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, Heraklion Greece, (Apr. 2020), 1–15. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387526.

[37] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. en. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, Heraklion Greece, (Apr. 2020), 1–16. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387532.

[38] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. Formally verified memory protection for a commodity multiprocessor hypervisor. In *30th USENIX Security Symposium (USENIX Security 21)*, 3953–3970. Retrieved Apr. 22, 2025 from https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei.

[39] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. {AMD} prefetch attacks through power and time. In *31st USENIX Security Symposium (USENIX Security 22)*, 643–660. Retrieved May 8, 2024 from https://www.usenix.org/conference/usenixsecurity22/presentation/lipp.

[40] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 355–371. Retrieved June 5, 2024 from https://ieeexplore.ieee.org/abstract/document/9519416/.

[41] Moritz Lipp et al. 2020. Meltdown: reading kernel memory from user space. en. *Communications of the ACM*, 63, 6, (May 2020), 46–56. DOI: 10.1145/3357033.

[42] Wu Luo, Qingni Shen, Yutang Xia, and Zhonghai Wu. 2019. {Container-IMA}: A privacy-preserving Integrity Measurement Architecture for Containers. en. In 487–500. ISBN: 978-1-939133-07-6. Retrieved Apr. 15, 2024 from https://www.usenix.org/conference/raid2019/presentation/luo.

[43] Yangdi Lyu and Prabhat Mishra. 2018. A Survey of Side-Channel Attacks on Caches and Countermeasures. en. *Journal of Hardware and Systems Security*, 2, 1, (Mar. 2018), 33–50. DOI: 10.1007/s41635-017-0025-y.

[44] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. en. *ACM SIGARCH Computer Architecture News*, 41, 1, (Mar. 2013), 461–472. DOI: 10.1145/2490301.2451167.

[45] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv:1902.05178 [cs]. (Feb. 2019). Retrieved May 8, 2024 from http://arxiv.org/abs/1902.05178.

[46] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014, 239, (Mar. 2014), 2:2.

[47] Moby. 2024. Moby/moby. original-date: 2013-01-18T18:10:57Z. (June 2024). Retrieved June 7, 2024 from https://github.com/moby/moby.

[48] Antonio Muñoz, Ruben Rios, Rodrigo Román, and Javier López. 2023. A survey on the (in) security of trusted execution environments. *Computers & Security*, 129, 103180. Publisher: Elsevier. Retrieved Dec. 19, 2024 from https://www.sciencedirect.com/science/article/pii/S0167404823000901.

[49] Vikram Narayanan et al. 2023. Remote attestation of confidential VMs using ephemeral vTPMs. en. In *Annual Computer Security Applications Conference*. ACM, Austin TX USA, (Dec. 2023), 732–743. ISBN: 9798400708862. DOI: 10.1145/3627106.3627112.

[50] [n. d.] Oci-distribution-spec - Google Search. (). Retrieved June 1, 2024 from https://github.com/opencontainers/distribution-spec.

[51] opencontainers. 2024. Opencontainers/runc. original-date: 2015-06-05T23:30:45Z. (Apr. 2024). Retrieved Apr. 15, 2024 from https://github.com/opencontainers/runc.

[52] J. Aaron Pendergrass, Nathan Hull, John Clemens, Sarah C. Helble, Mark Thober, Kathleen McGill, Machon Gregory, and Peter Loscocco. 2019. Runtime detection of userspace implants. In *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*. IEEE, 1–6. Retrieved May 15, 2024 from https://ieeexplore.ieee.org/abstract/document/9020783/.

[53] Ronald Perez, Reiner Sailer, and Leendert van Doorn. 2006. vTPM: virtualizing the trusted platform module. In *Proc. 15th Conf. on USENIX Security Symposium*, 305–320. Retrieved May 15, 2024 from https://www.usenix.org/event/sec06/tech/full_papers/berger/berger_html/.

[54] [n. d.] Production-Grade Container Orchestration. en. (). Retrieved May 13, 2024 from https://kubernetes.io/.

[55] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. 2010. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*. IEEE, 328–333. Retrieved May 13, 2024 from https://ieeexplore.ieee.org/abstract/document/5541547/.

[56] Dazan Qian, Songhui Guo, Lei Sun, Haidong Liu, Qianfang Hao, and Jing Zhang. 2020. Trusted virtual network function based on vTPM. In *2020 7th International Conference on Information Science and Control Engineering (ICISCE)*. IEEE, 1484–1488. Retrieved May 8, 2024 from https://ieeexplore.ieee.org/abstract/document/9532194/.

[57] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. en. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, Bertinoro Italy, (May 2019), 7–13. ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321445.

[58] Ali Raza et al. 2023. Unikernel Linux (UKL). en. In *Proceedings of the Eighteenth European Conference on Computer Systems*. ACM, Rome Italy, (May 2023), 590–605. ISBN: 978-1-4503-9487-1. DOI: 10.1145/3552326.3587458.

[59] 2024. Rumpkernel/rumprun. original-date: 2015-02-20T17:55:46Z. (May 2024). Retrieved May 13, 2024 from https://github.com/rumpkernel/rumprun.

[60] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*. Vol. 1. IEEE, 57–64. Retrieved Jan. 1, 2025 from https://ieeexplore.ieee.org/abstract/document/7345265/.

[61] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and Implementation of a {TCG-based} Integrity Measurement Architecture. en. In Retrieved Apr. 15, 2024 from https://www.usenix.org/conference/13th-usenix-security-symposium/design-and-implementation-tcg-based-integrity-measurement.

[62] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE symposium on security and privacy*. IEEE, 38–54. Retrieved May 13, 2024 from https://ieeexplore.ieee.org/abstract/document/7163017/.

[63] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. en. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, (Mar. 2020), 955–970. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378469.

[64] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. en. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Providence RI USA, (Apr. 2019), 121–135. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304016.

[65] Brijesh Singh. [n. d.] Confidential Computing with AMD SEV-SNP. (). Retrieved Apr. 24, 2025 from https://kvm-forum.qemu.org/2021/SEV-SNP-AMD-Brijesh-Singh.pdf.

[66] [n. d.] Stefanberger/libtpms: The libtpms library provides software emulation of a Trusted Platform Module (TPM 1.2 and TPM 2.0). (). Retrieved Nov. 14, 2024 from https://github.com/stefanberger/libtpms.

[67] [n. d.] Tpm 2.0: a brief introduction. (). Retrieved Nov. 14, 2024 from https://trustedcomputinggroup.org/wp-content/uploads/2019_TCG_TPM2_BriefOverview_DR02web.pdf.

[68] [n. d.] TPM 2.0 specification - Google Search. (). Retrieved Nov. 14, 2024 from https://www.google.com/search?q=TPM+2.0+specification&oq=TPM+2.0+specification&gs_lcrp=EgZjaHJvbWUyBggAEEUYOdIBBzUzNmowajeoAgCwAgA&sourceid=chrome&ie=UTF-8.

[69] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 645–658. Retrieved May 8, 2024 from https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[70] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An efficient java partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, 505–522. Retrieved Apr. 22, 2025 from https://www.usenix.org/conference/usenixsecurity20/presentation/tsai.

[71] umoci. 2023. Releases · opencontainers/umoci. en. (2023). Retrieved June 2, 2024 from https://github.com/opencontainers/umoci/releases.

[72] 2024. unikernelLinux/ukl. original-date: 2018-06-20T18:34:48Z. (Oct. 2024). Retrieved Nov. 10, 2024 from https://github.com/unikernelLinux/ukl.

[73] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. 2019. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. en. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, London United Kingdom, (Nov. 2019), 1741–1758. ISBN: 978-1-4503-6747-9. DOI: 10.1145/3319535.3363206.

[74] Stephan Van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 339–354. Retrieved June 5, 2024 from https://ieeexplore.ieee.org/abstract/document/9519461/.

[75] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2019. Microservice based architecture: Towards high-availability for stateful applications with kubernetes. In *2019 IEEE 19th international conference on software quality, reliability and security (QRS)*. IEEE, 176–185. Retrieved May 8, 2024 from https://ieeexplore.ieee.org/abstract/document/8854724/.

[76] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning power {Side-Channel} attacks into remote timing attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*, 679–697. Retrieved June 5, 2024 from https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen.

[77] Xuyang Zhao, Mingyu Li, Erhu Feng, and Yubin Xia. 2022. Towards a secure joint cloud with confidential computing. In *2022 IEEE International Conference on Joint Cloud Computing (JCC)*. IEEE, 79–88. Retrieved May 8, 2024 from https://ieeexplore.ieee.org/abstract/document/9898084/.