Tyche: Composable Isolation as a Foundation to Manage Trust in the Cloud

Adrien Ghosn*
Azure Research, Microsoft
Cambridge, UK
adrienghosn@microsoft.com

Charly Castes*
EPFL
Lausanne, Switzerland
charly.castes@epfl.ch

Neelu S. Kalani
EPFL
Lausanne, Switzerland
neelu.kalani@epfl.ch

Yuchen Qian
EPFL
Lausanne, Switzerland
yuchen.qian@epfl.ch

Marios Kogias
Imperial College London
London, UK
m.kogias@imperial.ac.uk

Edouard Bugnion
EPFL
Lausanne, Switzerland
edouard.bugnion@epfl.ch

***Abstract*—Cloud workloads combine software components from different parties to process sensitive data. Each component has its own trust model—it must protect its assets from the rest of the system, yet share sensitive data with components it cannot trust to keep confidential. This tension requires composing isolation boundaries for confidentiality and encapsulation. Unfortunately, the cloud offers no direct way to compose such boundaries, forcing tenants to assemble, deploy, and maintain their own solutions. This paper shifts that burden back to the infrastructure by making composable, attestable isolation a first-class systems abstraction.**

We present Tyche, a security monitor that centers isolation around a unified composable abstraction: security domains (SDs). An SD is an execution environment whose access to machine resources—memory, cores, devices—is controlled through explicit capabilities. A small set of capability operations enables SDs to partition, share, and reclaim resources; by nesting recursively, SDs compose attestable trust boundaries for confidentiality and encapsulation. Tyche attests these compositions, providing end-to-end security guarantees for workloads made of mutually distrustful components. As a first-class cloud primitive, this single abstraction subsumes enclaves, sandboxes, CVMs, and their compositions.

Tyche provides composable isolation without sacrificing compatibility with existing hardware and software stacks. It runs on commodity x86_64 hardware without security extensions, and a RISC-V prototype demonstrates portability across platforms. Our SDK composes isolation for unmodified workloads within SDs with minimal overhead. In a confidential LLM inference scenario with mutually distrustful users, model owners, and cloud providers, the slowdown is just 2% compared to bare-metal Linux.

1. Introduction

The cloud lacks abstractions for managing heterogeneous trust. Modern workloads are multi-party: even within a single virtual machine (VM), container, or process, components supplied by independent parties – libraries, frameworks, services, CSP infrastructure – must cooperate to process sensitive data, but none fully trusts the others. A component must protect its own assets, such as ML models or proprietary code, while handing sensitive data to untrusted components that perform essential processing but may leak it. This simultaneous need for confidentiality and encapsulation is what we call *controlled sharing*. It is inherent to cloud deployments, yet no cloud abstraction directly supports it.

Cloud abstractions, *e.g.,* containers and VMs, ease deployment but offer only coarse-grained trust management. Even trusted execution environments (TEEs) are limited to a binary trust model: software inside an enclave [22], [58] or a confidential VM (CVM) [2], [57], [73] is trusted, and its confidentiality and integrity is guaranteed against privileged code, *e.g.,* the CSP's hypervisor. Yet managing heterogeneous trust within the TEE itself, and enforcing controlled sharing among its components, is left entirely to the tenant.

Tenants address this burden in one of two ways: (1) external isolation, running components in separate TEEs, or (2) intra-isolation, inside a single TEE.

In the external approach, each component runs in its own CSP-provided TEE. The TEEs mutually attest before exchanging encrypted sensitive data over untrusted memory. This protects the confidentiality of each TEE against other components, but fails to guarantee encapsulation once one TEE hands data to another. Preventing unintended leakage thus requires trusting or verifying the functional correctness of each TEE—an unrealistic assumption when TEEs contain proprietary code, a full commodity operating system and applications, or require frequent updates.

Alternatively, intra-isolation composes isolation by creating boundaries within the TEE to separate its software components. By leveraging the TEE's internal privilege hierarchies—rings, AMD SEV-SNP's VMPLs [90], TDX L1/L2 partitions [57], [60]—privileged software can enforce isolation for less-privileged components. This enables nested enclaves for confidentiality [17], [109], protecting assets from the rest of the CVM's code, or sandboxes for encapsulation [47], [107], preventing untrusted components from leaking sensitive data.

Intra-isolation is a popular research direction, and Table 1 surveys representative examples. However, the space is fragmented: each system provides nested isola-

* *Co-equal first author.*

Outer \ Inner	No Nesting	Sandbox	Enclave	VM	CVM
Sandbox	gVisor [47], Wasm [8]	Capsicum [103]	-	-	-
Enclave	SGX [58], Nitro Enclave [23]	Ryoan [53]	Nested-Enclave [81]	-	-
VM	VT-x [98]	Hyperlight [79]	SGX [58]	KVM Nested-Virt	Hyper-V confidential L2
CVM	TDX [57], SEV-SNP [90]	Erebor [107]	Veil [17]	OpenHCL [78]	-

Table 1: Nested composition of isolation abstractions in prior work. Rows denote the outer isolation boundary and columns the inner boundary. Prior systems support only specific combinations, each requiring dedicated hardware or software extensions, motivating the need for a unified composable isolation solution.

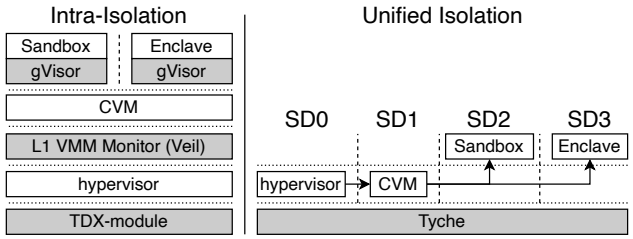

Figure 1: Composing enclaves and sandboxes within a CVM. Left: Intel TDX intra-isolation, with gVisor [47] sandboxing both an application (sandbox) and a VMM-provided enclave. Right: our approach. Arrows indicate management dependencies.

tion for a fixed threat model and platform, and none are designed to interoperate. Existing systems target different threat models, nested sandboxes [107] or enclaves [17] on different platforms (*e.g.,* Veil [17] on AMD, Erebor [107] on Intel) or compete for the same privilege layer (*e.g.,* Veil and Verismo [109] both require VMPL0), making these systems hard to compose. Hardware further limits composition, capping the number of sub-components (*e.g.,* only 1+3 partitions in TDX [60]) or prohibiting certain combinations outright (*e.g.,* no SGX enclaves inside TDX CVMs).

Worse, even setting these incompatibilities aside, combining separate solutions in a single deployment compounds complexity and is counter-productive for security. Consider private LLM inference: the user must reveal prompts to an untrusted LLM runtime without allowing exfiltration, while the model owner must keep proprietary weights hidden from both user and CSP. This demands composing nested enclaves and sandboxes within a single CVM, a typical case of controlled sharing. In Fig. 1(left), an L1 VMM monitor (*e.g.,* similar to Veil [17]) provides a nested enclave to protect the LLM runtime and model weights, while the user controls the surrounding CVM and uses gVisor [47] to encapsulate the enclave and prevent prompt leakage or sandbox other applications. Composing isolation further, *e.g.,* sandboxing libraries inside the enclave, deepens the hierarchy and would require Wasm [8] or NaCl [106] as in Ryoan [53]. Each layer duplicates enforcement, expands the trusted computing base (TCB), and forces tenants to integrate, configure, and maintain complex sub-systems. Correct isolation becomes fragile and error-prone, and attesting end-to-end security grows complicated by uncertainty about which layers must be measured. Here, to ensure prompts are not leaked, the user must attest the TDX module, the CVM, and gVisor, but also the monitor to ensure it does not provide enclaves with a covert channel to bypass the gVisor sandbox.

Our insight is that intra-isolation complexity is a symptom, not a necessity: it reflects the lack of a composable isolation abstraction in the cloud. All isolation mechanisms—regardless of trust model or platform—ultimately enforce access and boundary control, and differ only in where and how they apply it. Intra-isolation stacks them because the cloud offers no way to compose access and boundary control directly. A single enforcement layer that natively supports composition would eliminate the duplication entirely.

We thus propose a third approach that makes composable isolation a first-class cloud abstraction. A single trusted entity provides and attests composable access and boundary control. By making composition a core feature, this entity allows components to compose attestable isolation boundaries for diverse trust models, without duplicating enforcement. The result is lower deployment complexity, a smaller software and hardware TCB, and precise management of nuanced trust relationships in the cloud.

Our approach raises two key research questions: (1) *What primitives enable composable isolation for diverse trust models?* (2) *How can we introduce composable isolation without breaking existing software and hardware stacks?* These questions guide our design toward a general yet practical solution.

We realize this approach in Tyche, a security monitor that provides composable isolation through a unified abstraction: security domains (SDs). An SD is an execution environment whose access to machine resources is controlled through explicit capabilities. A small set of capability operations—creating SDs, partitioning, sharing, and reclaiming resources—lets SDs *recursively compose attestable isolation boundaries*. Tyche attests these compositions, making access and boundary control explicit end-to-end. This subsumes prior mechanisms: exclusive access supports confidentiality and integrity, while control over shared resources and interactions enables encapsulation.

Tyche runs on bare metal at the highest privilege level, attested by a hardware root of trust [96], and enforces isolation for all other software running as SDs. Implemented in Rust [7], it uses standard hardware access-control mechanisms and supports deployment across platforms. We present a full implementation on x86_64 using virtualization technologies [59], [98] and a firmware prototype on RISC-V [84]. For compatibility, our Tyche SDK allows Linux SDs to create and compose sandboxes, enclaves, and CVMs as nested SDs. This replaces ad-hoc intra-isolation mechanisms (Fig. 1, right) and covers the entire design space of Table 1 with unified semantics, attestation, and enforcement.

Our evaluation on x86_64 shows that Tyche SDs—replacing enclaves, VMs, CVMs, and nested enclaves within CVMs—perform comparably to their native equivalents across diverse workloads, including web servers,

databases, key-value stores, and LLM inference. In a confidential LLM inference scenario with mutually distrustful users, model owners, and CSPs, the slowdown is just 2% compared to bare-metal Linux. Tyche is an open-source research prototype [37] that independent researchers used to achieve custom isolation guarantees on legacy hardware [39].

The remainder of this paper covers background and motivation (§2), a high-level overview (§3), Tyche's capability API (§4), its platform-specific implementation (§5), and the Tyche SDK (§6). We then evaluate Tyche (§7) and discuss inspiration drawn from related work in monitors and micro-kernels (§8).

2. Motivation: See the forest for the T(r)EEs

Observations: Computer system security requires resource (*i.e.,* memory, CPU, interrupts, devices) and fault isolation between software components that implement different tasks or services. Isolation is however rarely absolute, as components need to cooperate to carry out computations. For instance, two processes might share memory; isolation between a VM and a hypervisor is unidirectional to enable, *e.g.,* emulated devices or migration. Isolation in systems is thus more accurately described as an attempt to enforce the least privilege principle [88]: each component should be granted only the minimal access rights it needs to perform its intended operation. Historically, systems were designed under the assumption that a single organization controlled the entire software stack and hardware, but that assumption no longer holds as cloud platforms and modern deployments combine components from multiple independent sources and place critical privileged components under the control of external providers in the cloud.

As modern systems integrate components from diverse sources, enforcing least privilege increasingly relies on *compartmentalization*: structuring the system so that each component's access to the machine is tightly bounded. *Confidential computing* provides one form of compartmentalization by preventing access from more privileged code. It departs from traditional system designs that conflate privilege with trust and treat resource management as justification for access. Examples include enclaves [43], [58], [76] and CVMs [57], [73], [90], which differ mainly in how they interface with the surrounding system, not in their underlying trust model. Another form of compartmentalization is *encapsulation*, which restricts how a component may interact with the rest of the system, often to prevent information leaks or interference with other components. Sandboxing mechanisms [8], [105], [106], as well as containers, are examples of encapsulation.

While often treated separately, confidential computing and encapsulation are two sides of the same coin [31]. Both aim at controlling access to resources. Encapsulation focuses on *sharing*: explicitly granting a component-controlled access to resources such as memory or interfaces. Confidential computing, in contrast, focuses on *exclusivity*: ensuring that a component's resources cannot be accessed by others, either via access control mechanisms [50], [58], [76] or encryption and integrity guarantees [19], [57]. Existing solutions often emphasize only one side, reflecting the trust model of a particular component and producing rigid abstractions – for example, the binary notion of trust in TEEs – without recognizing the common underlying goal. From a holistic cloud security perspective, all solutions are mere instances of access and boundary control. Such control enables higher-level properties such as confidentiality, integrity, encapsulation, and controlled sharing: the ability to selectively expose component resources while guaranteeing they cannot be leaked to unintended parties. To address the varying components' trust models in a coherent whole, isolation needs to be *composed*, *i.e.,* components must be able to define and control their own boundaries.

We define composable isolation as the ability to assemble isolation boundaries in a way that preserves and combines their security guarantees. Each isolation boundary can be seen as a function that refines the access and boundary control of the components it contains: it restricts what they can access and with whom they can interact. Composing isolation is then composing these functions: each new boundary refines the previous one, analogous to function composition in mathematics where $f \circ g(x) = f(g(x))$: each layer further tightens access and boundary control. This composition can be enforced by hierarchical systems that each apply one refinement—first g, then f—as intra-isolation does today, or by a single layer that enforces the composed projection $f \circ g$ directly.

In a cloud setting, where control over software and hardware may lie with an external party, isolation must be *attested*. Attestation demonstrates to a local or remote entity that a known software version is running on a platform and is isolated from the rest of the system according to the intended security policies. It typically relies on a hardware root of trust, such as the CPU [19], [57], [58] or a Trusted Platform Module (TPM) [96] and may also involve trusted software.

Lessons learned and goals: From the discussion above, two key properties emerge as fundamental for composing isolation in heterogeneous cloud systems. First, management must be decoupled from access, and trust decoupled from privilege, reflecting the insight from TEEs. Second, access and boundary control must be attestable, so it is possible to verify that a component's interactions and resource accesses respect intended policies. These lessons address our first research question: *What primitives enable composable isolation for diverse trust models?* We argue that the two sufficient primitives are *attestation*, and *access and boundary control*. With those, we can not only reconstruct existing higher-level abstractions but also provide finer-grained and composable isolation.

Approach: Providing these primitives while addressing our second research question, *i.e.,* preserving the existing software stack, is challenging and requires careful design and implementation. In particular, it requires designing an interface that is small and easy to integrate with existing software, while implementing an enforcement that does not disrupt existing software hierarchies. For this, we identify capabilities and security monitors as attractive candidates.

Capabilities provide a structured and explicit mechanism for access control. They are unforgeable tokens – created either in software by a trusted entity [63] or implemented in hardware [18], [104] – that encapsulate

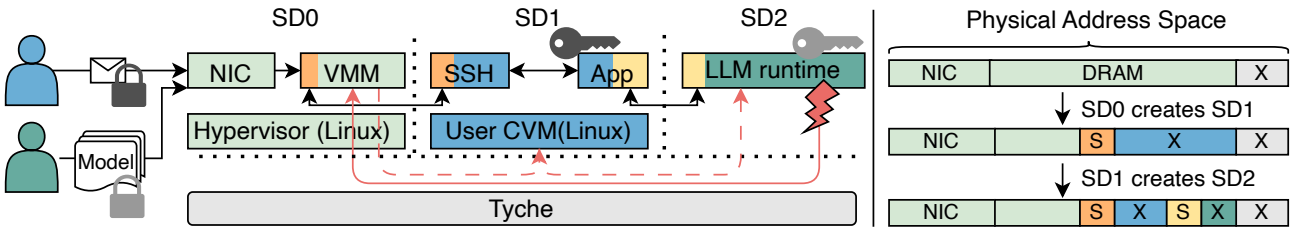

Figure 2: Mutual-distrust LLM inference deployment on top of Tyche. The left side shows the participating SDs – SD0 hypervisor, SD1 CVM, and SD2 nested enclave. The right side shows the machine's physical memory address space as SDs are created; the colors indicate the regions accessible to each SD on the left, distinguishing exclusive regions (X) from shared ones (S). The encrypted model, prompts, and replies traverse memory along the black path, while interrupts are routed as shown by the red arrows.

access to an object. Holding a capability grants specific access rights to the object, while transferring it to another component revokes access from the sender and grants it to the recipient. Capabilities are used by security-focused kernels [63] and provide clear semantics on resource access and sharing.

Monitors are small software components that conveniently *retrofit* [32], [108] new isolation boundaries into the existing software stack with minimal disruption by running at higher privilege levels, *e.g.,* through virtualization [32], [61], [76], [108]. They strive to remain passive unless their isolation services are used, leaving existing software execution mostly unchanged, *e.g.,* they do not replace the kernel for process isolation. They are popular for both confidential computing [17], [32], [43], [50], [61], [108] and encapsulation [24], [89], [107] to enforce fine grained access control.

Put together, a monitor that implements capabilities for access and boundary control can provide a single abstraction to compose isolation, support fine-grained heterogeneous trust models, and remain compatible with existing software and hardware stacks. This is what we design and implement with Tyche.

Unlike systems in Table 1, Tyche covers all scenarios with a single enforcement layer, provides coherent attestation across multiple (potentially nested) SDs, and goes beyond the limitations of prior work [17], [60] by supporting private shared memory between TEEs without imposing a cap on the number of isolated sub-components. Because its semantics are decoupled from the underlying platform, Tyche achieves this uniformly across hardware; the single enforcement layer further reduces deployment complexity, making Tyche both general and portable.

3. Tyche Overview

In this section we present an overview of Tyche's design, its threat model, and an example deployment in Fig. 2 that we will use throughout the paper.

3.1. Architecture

The Tyche security monitor exposes a capability-based API for creating and isolating security domains (SDs), its core abstraction. All software other than the monitor runs inside an SD, an execution environment whose configuration and owned capabilities define its access to machine resources – cores, devices, memory regions, and interrupts – attested by Tyche as either shared or exclusive. An SD can create child SDs, partitioning or sharing subsets of its resources and transferring control to children on selected cores through capability operations. Rather than catering to a fixed trust model, SDs provide a unified, attestable abstraction for controlling shared and exclusive resources, accommodating a wide range of security policies, such as sandboxes, enclaves, and CVMs, and enabling their composition.

The API decouples resource management from the attestable enforcement of access restrictions. Capabilities represent resources and govern SD interactions, letting domains implement their own isolation boundaries and scheduling policies. Tyche tracks resource allocation and attests whether resources are shared or exclusive, capturing and enforcing dependencies between SDs. It guarantees parents can reclaim resources or cores, while children are assured exclusive resources remain so unless explicitly shared, and that revocation or control transfer do not leak information.

Local and remote attestation make SD relationships explicit and verify end-to-end guarantees. Attestation has two parts: (1) a TPM-rooted measurement of the boot process and a public key, binding an Tyche binary to a physical machine and ensuring it runs alone at the highest privilege; and (2) a report generated by Tyche and signed with the private key describing the SD's resources, how they are shared, delegated to children, or can be reclaimed.

Tyche's support for composable isolation enables controlled sharing. SDs unify compartmentalization by supporting both confidentiality and encapsulation. Confidentiality and integrity derive from exclusive access; encapsulation is enforced by restricting a child SD's interactions and overlap of resources to trusted peers. Recursive SD construction enables composition, letting each domain define its own boundaries.

Tyche's platform-independent capabilities make it portable across architectures, with hardware-specific backends enforcing their semantics efficiently. On Intel, Tyche uses VT-x [98]'s extended page tables and the I/O-MMU [59]; on RISC-V, it runs as M-mode firmware using Physical Memory Protection [84]. The Tyche SDK adds a kernel driver to Linux environments to build nested sandboxes, enclaves, and CVMs using SDs. This compatibility with existing hardware and software stacks ensures Tyche's practicality.

3.2. Threat Model

Tyche assumes the underlying hardware, including physical devices and access control mechanisms, is trusted and part of its TCB. The CSP and tenants are adversarial. We consider an attacker, running arbitrary code within an SD, restricted to an authorized subset of the machine's resources, such as cores, memory, and device configuration space. In particular, the attacker might try to exploit: (1) the monitor's API, (2) device configuration space, and (3) privileged instructions to, e.g., emit or disable interrupts. The attacker aims to access resources or SD state outside of its or the device's authorized sets, compromise the monitor's metadata, steal its private key, or hog resources to prevent revocation.

Out-of-scope: Physical attacks, such as accessing DRAM or the PCI bus to read the monitor or an SD's memory, are out of scope of the current implementation, although they could be mitigated with hardware support, *e.g.,* total memory encryption [56] and PCI bus encryption [14]. **Side-channel**-based attacks are not explicitly addressed by Tyche, beyond appropriate flushes upon transitions, as it does not track shared micro-architectural state. They however can be mitigated within the current implementation through core partitioning [30], [110] and physical memory allocation based on cache-coloring [38], [101]. **Denial-of-service** attacks to exhaust the monitor's memory are possible, but they only prevent the creation of new SDs and do not prevent the revocation, attestation, or isolation enforcement of existing ones. On x86_64, they can be mitigated by requiring SDs to supply memory for the monitor's metadata. Inherent to the cloud, the CSP can deny service by turning off the platform [28].

3.3. Running Example

Fig. 2 shows a public cloud deployment where a user performs LLM inference with a proprietary model. The threat model involves full mutual distrust: the user does not trust the CSP or model owner with prompts, the model owner does not trust the CSP or user with model weights, and the CSP does not trust either party. Private inference in this setting requires confidential computing to protect prompts and weights, composed with encapsulation to prevent the LLM runtime from leaking prompts.

At boot, Tyche controls all resources and assigns them via capabilities to SD0, the CSP's Linux+KVM [34] hypervisor running Tyche SDK. To instantiate the user CVM, SD0 uses the SDK to partition its memory, creating SD1 with exclusive memory (blue in Fig. 2), shared memory (orange), and CPU cores. SD0 then transfers control to SD1 on those cores. Once booted, SD1 (Linux + Tyche SDK) creates SD2 for the LLM runtime in userspace, partitions its exclusive memory, and transfers a region (dark green) to SD2, forming an enclave isolated from both the CSP and the CVM but encapsulated by SD1. The model owner attests SD2's exclusive memory before provisioning the encrypted model. Communication uses shared memory: SD0–SD1 share a region for VIRTIO [66] networking (orange); SD1–SD2 share a private region (yellow), not accessible by SD0, to exchange plaintext prompts and replies. Encrypted prompts arrive at SD1 via orange, are decrypted into private memory (blue), and forwarded to SD2 via yellow; replies follow the reverse path, encrypted by SD1 before being sent to the VIRTIO network. As there is no direct communication between SD2 and SD0, SD1 fully encapsulates the untrusted LLM runtime in SD2 and thus prevents prompt leakage. The model is delivered encrypted via orange, passed to SD2 through yellow, and decrypted in its private memory (dark green).

All parties use Tyche's remote attestation: (1) The CSP confirms it retains resource control and manages interrupts (red arrows). (2) The user confirms SD1 is isolated from SD0 and encapsulates SD2, which only communicates with the CVM. (3) The model owner confirms SD2 runs in exclusive memory without leaking the model. (4) The user and model owner confirm interrupts do not leak data and that Tyche zeroes memory before returning it to the CSP.

4. Tyche Design: API & Capabilities

This section addresses our first research question by designing a small set of primitives to compose isolation and support heterogeneous trust models. We focus on primitives for access and boundary control with clear, predictable semantics. To achieve this, we devise software capabilities centered on sharing, transferring, and reclaiming resources. Our capabilities make shared access and interactions explicit while decoupling policies from mechanisms [67] for portability across hardware platforms. Tyche capabilities encode enough information to attest how an SD was created, what interactions it may have with others, and under which conditions it can be revoked. This explicitness and fixed semantics gives the system predictable behavior, enabling long-term reasoning about isolation guarantees and trust relationships from a single attestation.

Tyche capabilities are unforgeable tokens issued by the monitor. They are owned by security domains (SDs) and mediate access to two object types: memory regions and SDs. Memory capabilities grant access to physical memory ranges, while SD capabilities govern SD management, *i.e.,* creation, configuration, attestation, and execution. CPU cores and interrupts fall under SD capabilities, and devices are modeled as SDs. In this section, we show that these two capability types and the narrow set of operations in Table 2 to derive new capabilities are enough to isolate SDs and manage their nuanced trust relationships.

4.1. Capability Derivation Trees

Tyche maintains two capability derivation trees (CDTs): one for memory regions and one for security

Call	Description
CREATE	Create a security domain
SET/GET	Set/Get a security domain's register or policy
SEND	Transfer a capability ownership to an SD
SEAL	Seal a security domain
ATTEST	Attest a security domain
ENUMERATE	Discover info. about owned capabilities
SWITCH	Control transfer into a security domain
ALIAS	Create new memory region by aliasing one
CARVE	Create new memory region by carving one
REVOKE	Revoke a capability's child
GETCHAN	Create a channel from an SD or existing channel

Table 2: Tyche's API

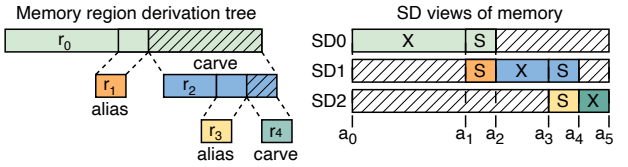

Figure 3: Memory region derivation tree and SD memory views based on Fig. 2. New regions are created by carving and aliasing an existing region. Sends between SDs are omitted. (S)hared and e(X)clusive memory is reported on views.

domains. CDTs, widely used in capability systems [63], [91], derive each capability from an existing one, inheriting equal or reduced access. Derived capabilities appear as children of their parent node in their trees. CDTs ensure (1) monotonicity of operations, (2) a record of operations, and (3) efficient cascading revocation of an entire subtree by revoking its root.

In Tyche, we further leverage the CDT structure to track overlapping resource access and dependencies between SDs, ensuring these relationships are reflected in attestation and that operations always leave the system in a consistent state. The region CDT tracks exclusive and shared access to physical memory, while the SD CDT encodes SD management hierarchies, scheduling, and interrupt routing. Both trees support efficient revocation and explicitly encode access policies and inter-object dependencies.

4.2. Memory Region Capabilities

Tyche region capabilities offer a compact, attestable representation of shared or exclusive access to physical memory ranges, with security *attributes* that define guarantees upon revocation. Tyche initializes the CDT with a root memory region marked as **exclusive**, defined by a start and end address, access rights (read, write, execute), and an empty set of attributes. Tyche transfers this root region's ownership to SD0, the first domain to run on the machine.

Memory operations: An SD creates new region capabilities from owned ones using Tyche's *alias* and *carve* API calls (Table 2). Alias creates a child region capability, marked as **aliased**, for a subrange of the parent region's physical addresses with equal or reduced access rights while preserving the parent's access. A carve similarly creates a subregion but removes access to it from the parent region (see Fig. 3, hatched area). If the parent region was **exclusive**, the carved region is exclusive; otherwise, it is marked as aliased. Carving enables confidential memory: an exclusive region stems from an unbroken chain of carves and no operation outside its subtree can alter its exclusivity. Tyche ensures a capability's access rights and exclusivity are determined locally, based on its initial range, exclusivity status, and direct children. In Fig. 3, r_0 initially grants exclusive access from a_0 to a_5. After an alias (r_1) and a carve (r_2), it retains exclusive access only from a_0 to a_1 and shared (aliased) access from a_1 to a_2, regardless of further subdivisions of r_1 or r_2.

Memory management: Tyche's *send* and *revoke* calls let SDs transfer and reclaim memory. *Send* transfers ownership of a region, allocating it to the receiver. *Revoke* acts on a parent region to undo a child alias or carve, letting the parent's owner reclaim memory sent via that child.

```
sd1 = domain {r1, r2, sd2}
      |registers.HASH: 8988ef57...
      |cores: 0b11
      |mon.api: 0b11111111111 | RECEIVE
      |interrupts: {
      | 0 -> {Report, registers: 0b0},
      | ...          }
sd2 = domain {r3, r4}
      |registers.HASH: 978de00f...
      |cores: 0b01
      |mon.api: 0b00001110000 | !RECEIVE
      |interrupts: {
      | 0 -> {Not report, registers: 0b0},
      | ...          }
r1 = aliased a1 a2 with RW_
r2 = exclusive a2 a5
     with RWX, HASH|CLEAN|VITAL
     |HASH: 755ee2b2...
     |alias at a3 a4 for r3 with RW_
     |carve at a4 a5 for r4 with RWX
signature: a0e0d23f26564bd5...
```

Figure 4: Simplified attestation for SD1 with Fig. 3's nomenclature. Allowed monitor API calls are encoded as bitmaps based on the order from Table 2 (*e.g.,* bit 0 is create).

Revocations cascade, deleting the entire subtree rooted in the revoked capability: revoking r_2 from r_0 also revokes r_3 and r_4, restoring r_0's access from a_2 to a_5. To provide security guarantees to the receiver, Tyche allows the sender to optionally attach *attributes* to the transferred region: (1) *hash* captures a cryptographic hash of an exclusive region's content, enabling the receiver to ensure it contains the correct initial data; (2) *clean* ensures the region is zeroed upon revocation to prevent data leakage; and (3) *vital* revokes the receiver if the capability is revoked, enforcing a minimal memory set necessary for functionality. Attributes are set only when sending to an unsealed receiver (§4.3), are bound to ownership rather than the CDT, and are non-monotonic.

The receiver then inspects received regions using *enumerate* or *attest*. These calls report a region's status (exclusive or aliased), initial range, access rights, attributes, and direct children, whether owned or not, as shown for r_2 in Fig. 4.

4.3. Security Domain Capabilities

Tyche initializes SD0, the first SD which owns the root memory region, runs on all cores, and handles all interrupts; all other SDs are created from subsets of these initial resources.

SD creation and configuration: *Create* enables an SD to spawn a child in the CDT and obtain an SD capability referencing the newly created domain. The capability enables the parent to configure, send regions, seal, attest, schedule, and revoke the child. The child is initially *unsealed* and cannot execute. The parent configures the child via *set*, specifying per-core registers state and SD *policies*. Policies define the child's allowed cores, permitted monitor calls and whether they are allowed from user space, whether it can receive new capabilities after *sealing*, and interrupt policies. Policies are monotonic: Tyche rejects *set*s exceeding the parent's rights. Memory is provisioned as in Fig. 5, with the parent aliasing and carving its regions before sending them. *Seal* makes the

SD executable and prevents further sets to its registers and policies. Revoking an SD triggers the cascading revocation of all its owned capabilities and its CDT subtree.

SD execution & interrupts: Control transfers between SDs *i.e.,* transitions on a core, occur via explicit monitor calls or upon interrupts. The *switch* API call implements a call-return model: Tyche ensures the callee is authorized to run on the core, saves the caller's state, loads the callee, and transfers control. Switching to a child requires its SD capability; a switch with no SD argument returns to the parent.

Interrupts trigger SD control transfers. They propagate through the CDT, which encodes routing and handling policies. For each interrupt vector, SD's interrupt policies specify whether to *Deliver*, *Report*, or *Not report* the interrupt, and which registers the parent can access during handling. An interrupt marked as *Deliver* is received by the SD directly. When an interrupt not marked as *Deliver* occurs, Tyche preempts the SD and walks the CDT upwards to transfer control to the first SD (the handler) it finds with a *Deliver* policy. To the handler, the interrupt appears as a return from a switch to its direct child, with the interrupt acting as the return value. After taking care of the interrupt, the handler resumes the child's execution by performing a switch. The return path walks the CDT downwards, with Tyche reporting the interrupt to all SDs with a *Report* policy in a manner similar to the original handler SD. These SDs observe the interrupt as if it originated from their direct child and decide whether to resume execution. Those with a *Not report* policy are skipped.

Consider the deployment in Fig. 2. SD2 triggers a divide-by-zero exception (interrupt vector 0). Per Fig. 4, SD2's policy for vector 0 is *Not report*, so the exception traps to Tyche, which walks up the CDT searching for an ancestor SD with a *Deliver* policy. Skipping SD1, whose policy is *Report*, Tyche switches to SD0—the first ancestor with a *Deliver* policy—presenting the event to SD0 as a return from a switch to SD1. SD0 handles the exception as if originating from SD1's subtree, with access to SD1's registers restricted by SD1's policy, then switches back to SD1. Because SD1 is marked *Report*, Tyche delivers the return to SD1, which sees it as a return from a switch to SD2 caused by a divide-by-zero. Before resuming SD2, SD1 may inspect or modify SD2's register state—within the bounds defined by its policy—and decide how to proceed, for example resuming SD2 after correcting the faulting state. This routing and return protocol enables flexible interrupt delivery for isolation abstractions ranging from sandboxes to nested VMs, while providing attestable scheduling guarantees on how SDs can be preempted.

SDs interactions: SDs communicate via shared memory, control transfers to direct children, and interrupts. To ensure that parents remain responsible for scheduling and revoking their children, SD capabilities cannot be transferred between SDs. To enable non parent-child SDs to attest each other and exchange regions without relying on a common ancestor, Tyche supports *channel* capabilities. Channels are derived from SD capabilities using *getchan* and appear as children in the CDT. They act as weak references to SDs and can be transferred between SDs via *send*. Channels allow non parent-child SDs to directly attest each other, exchange memory regions (*e.g.,* to establish private shared memory),

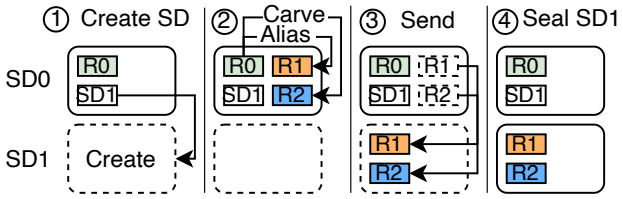

Figure 5: Capability operations to create and configure SD1.

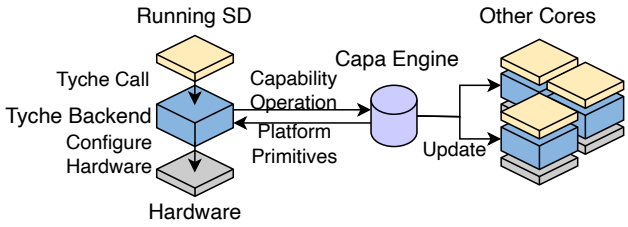

Figure 6: The capability engine maintains the system state across all cores.

or share other channels – but not to schedule, configure, or revoke an SD.

Devices: Tyche models devices as SDs, with configuration space, DMA, and port I/O access mediated through region capabilities. Interrupt routing follows the same protocol as above, but Tyche allows backends to optimize core-routing using platform specific hardware mechanisms. SDs on the CPU interact with devices through shared regions for the device's configuration space and MMIO. Tyche provides SD0 with a channel to every device. SD0 delegates device access to an SD by carving the configuration space and duplicating a channel, sending both to the SD to enable direct SD to device interactions.

4.4. Attestation & Security guarantees

SDs can request an attestation for themselves and any SD for which they own an SD capability (or channel). Tyche's attestation reports the SD's configuration, including a hash of initial register content, policies, and a description of owned capabilities. For remote attestation, an SD's *attest* call includes the remote verifier's public key and nonce, as well as a public key generated by the domain itself, ensuring they are all measured and signed by Tyche.

Conceptually, the attestation provides, for each owned capability. visibility into its direct children in their respective CDT, but not further. Fig. 4 shows the attestation of SD1. It reports the configuration of SD1 and its child SD2, along with their owned capabilities. Allowed cores, monitor calls, and registers that can be read and written upon interrupts are encoded as bitmaps. Lineage between regions is made explicit through naming: *e.g.,* r_3, owned by SD2, is derived from r_2. If SD2 held a capability not derived from SD1, it would appear only in SD2's set of owned capabilities with a fresh name and no further information. Indices are reused here for clarity; in practice, Tyche assigns fresh names, starting from 0 based on the requester's capabilities, to avoid revealing information about the broader system.

Tyche's attestation makes sharing and interactions explicit, while providing guarantees on initial SD state, run time events, and resource reclamation. Tyche thus supports confidentiality and integrity, encapsulation, or even the privacy of communications between distrustful

SDs. Confidentiality and integrity are ensured by exclusive ownership of regions and the absence of leakage through interrupts or revocation (*i.e.,* attributes). Encapsulation, as for SD2 in Fig. 4, is achieved by ensuring a child's regions are a subset of the parent's exclusive ones, and that its policies prevent receiving or sending capabilities after sealing. The child can still be confidential, as is the case for SD2, and SD1 can safely share information with it via their private shared memory in r_3 without risk of leakage or trusting SD2's implementation.

5. Tyche Implementation

Fig. 6 shows the Tyche security monitor has two components: (1) a platform-independent *capability engine* and (2) a platform-specific *backend*. The backend translates hardware events, such as interrupts or calls to Tyche, into capability operations and forwards them to the capability engine, shared across cores. The capability engine implements the system's global capability state machine, validates and executes operations and notifies the backend of configuration changes on affected cores via *updates*. The capability engine and backend are part of the TCB.

To construct a backend and provide the attestable enforcement of isolation by the monitor and its capability engine, the hardware must provide: (1) the ability to establish trust in the monitor; (2) the monitor's exclusive oversight of an access control mechanism to enforce resource isolation (memory, CPU, interrupts, devices); (3) a direct, secured communication channel between SDs and the monitor.

This section describes the custom loader we use on bare metal to establish trust in the monitor, the capability engine, details the access control enforcement and communication in the x86_64 backend, and provides an overview for RISC-V.

5.1. Tyche attested boot

Tyche boots via a custom bootloader that enumerates CPU cores, devices, and DRAM to generate a *boot-info*. It reserves memory for Tyche per a compile-time configuration, loads the monitor (capability engine and backend), and maps the boot-info into its address space. The bootloader generates an attestation key pair, measures the monitor, boot-info, and public key into a TPM [96] PCR, and transfers control to the monitor with the private key. Using the boot-info, the backend initializes the capability engine, creates SD0 (*e.g.,* a stock Linux kernel) with access to all unused memory and device regions, and sets up an I/O SD for each DMA-capable device with an alias to SD0's memory. Control is then transferred on all cores to SD0.

Like prior work [76], an SD attestation includes a TPM quote and a domain attestation signed by Tyche (§4). The quote binds the monitor binary, platform, and attestation key via PCR values, enabling trust in the monitor and transitively in SD attestations. The attestation covers the full boot chain, including our bootloader. We also prototyped dynamic root of trust (DRTM) support [20], [55] via Intel TxT [55], reducing the TCB to just the capability engine and backend.

5.2. Capability engine

The capability engine implements the region and SD capability derivation trees (CDTs from §4), validates and executes capability operations, and computes *updates* supplied to the backend to reflect configuration changes onto the hardware. It is implemented as a standalone, bare-metal (`no-std`) Rust library. The engine consists of 4K lines of code, uses no `unsafe` [86] to ensure memory safety, and is fuzzed as part of our continuous integration (CI) setup using LLVM's *libFuzzer* [75] to proactively detect and fix bugs.

The capability engine exposes the API defined in §4 for capability operations and integrates within a backend via a *platform* interface (Rust trait). This interface is supplied by the backend to provide platform-specific primitives to manage per-SD platform state (*e.g.,* per-core registers and access control mechanism configuration such as extended page tables (EPTs)), map and unmap physical memory ranges in an SD's platform state, manage interrupts, and implement cross-core synchronization primitives in the form of IPIs, barriers, and locks.

At all times, the engine tracks which SD executes on each core and ensures a consistent state view across cores. The engine uses the CDTs to determine which SDs and regions an operation affects and serializes operations with overlapping targets to ensure consistency. For each operation, the engine derives the set of affected cores and enqueues an update reflecting the result of the operation in per-affected-core queues, as shown on Fig. 6. The engine uses the platform interface to preempt affected cores (IPIs), requesting them to process the update and block on a barrier while the initiating core makes the new state globally visible. They are then unblocked and apply the changes to their local hardware state. This ensures the atomicity of capability and hardware state changes across cores. In practice, only a small set of updates are needed: access right changes, SD revocation, and interrupt delivery.

For example, sending a carve is handled by a core in the engine, which validates the request, computes the new capability state, and enqueues an access-right update on all cores running the sender or receiver. Using the platform IPI, it preempts these cores so they can process the update. The update involves two synchronization barriers: the first ensures all affected cores are preempted; the second waits for the new platform state. Between them, the initiating core updates the platform state – unmapping the region from the sender and mapping it into the receiver (*e.g.,* via EPTs on x86_64). Finally, it finalizes the capability state and releases the second barrier, letting other cores apply the new state locally (*e.g.,* TLB shootdown).

The capability engine requires memory to allocate capability metadata and SD state from. Tyche provides this memory in two ways: statically or dynamically. In the static configuration, Tyche reserves a contiguous block at boot and passes it to the engine. On RISC-V, this is required because the architecture provides only a small number of PMP entries, and Tyche must protect itself and its metadata using at most one region (§5.4). To prevent memory exhaustion from becoming a covert channel or denial-of-service vector, Tyche can enforce per-SD capability quotas.

On platforms without this restriction, *e.g.,* using EPTs on x86_64, Tyche supports dynamic memory management

in which SDs supply memory for their children's metadata [63], [70]. This is integrated into region capabilities semantics with a *META* attribute (§4.2). Regions transferred as *META* to a child must be exclusive, *vital*, and are *clean*ed (zeroed) on revocation. The monitor uses these regions to host, *e.g.,* a child's EPTs. They are inaccessible to both parent and child, but are included as other regions in attestations.

Both approaches introduce trade-offs. Static reservation exposes TYCHE to denial-of-service via resource exhaustion, though isolation, attestation, switches, interrupts, and revocation always succeed (§3.2). Dynamic management avoids this limitation but enables potential cache-based side-channel attacks on metadata, which should be mitigated with allocation based on cache coloring – hence their inclusion in attestation.

5.3. TYCHE x86_64 backend in root mode

Intel VT-x [98] accelerates virtualization by running a host in `root` mode and guest VMs in `non-root` mode. A virtual machine control structure (VMCS) governs guest execution, virtualizing privileged operations like `cr3` writes without host intervention. Extended page tables (EPTs) [15], [98] restrict non-root memory access. Guests trap to the host via the `vmcall` instruction.

The TYCHE x86_64 backend reuses VT-x to isolate SDs, running in root mode and executing SDs in non-root mode. Each SD has its own EPT (enforcing region capability access) and per-core VMCS. Devices are also modeled as SDs, with memory isolation enforced via Intel I/O-MMU [59]. TYCHE uses `INIT IPI`s to trap cores into the monitor and implements synchronization with semaphores, atomics, and spinlocks. The backend is 6K lines of Rust, bloated by error codes and VMCS field enums, with unsafe code for hardware setup. Along with the capability engine, it compiles to only 230KB.

SDs interact with the monitor via the non-interposable *vmcall* instruction. Arguments and return values are passed through general-purpose registers, with capabilities referenced by SD-local indices (like UNIX file descriptors). Large values (*e.g.,* attestations) use a multi-*vmcall* protocol. We measured register-based communication to be efficient, supporting tens of thousands of large (4KB) attestations per second. Registers avoid monitor access to SD memory, reducing risks like confused deputy [48] and cache attacks. The monitor accesses memory only when no SD can, to enforce `Clean` (zeroing) or `Hash` (reading). Future work aims to offload even these to more restricted environments [35], ensuring complete isolation.

SD switches save the caller's VMCS and load the callee's. The backend manages general-purpose registers in software and handles other state, including MSRs, using a mix of hardware and software to prevent leakage. EPT TLB entries are tagged with virtual processor IDs to reduce flushes on transitions. An SD runs until it returns or receives an unhandled interrupt. All control transfers—whether through `switch` or interrupts—go through the monitor, which tracks per-core state and enforces SD policies.

SD interrupt policies are offloaded to hardware via VMCS where possible. Intel VT-x allows fine-grained traps for exceptions (vectors 0–31) but only a single bit for external interrupts. Thus, external interrupts trap to the backend which selectively decides to either re-inject them or switch to an ancestor SD per §4. When supported, the x86_64 backend uses hardware APIC virtualization and I/O-MMU interrupt remapping to reduce traps and gain finer control. It can also expose VMCS fields as SD registers, letting parents manage interrupt delivery at the cost of limiting child policies.

5.4. TYCHE RISC-V backend in M-mode

On RISC-V, TYCHE runs in machine mode (M-mode), and the backend uses Physical Memory Protection (PMP) [84] to enforce region capabilities. PMP entries enforce permission on contiguous segments of physical addresses and can only be configured from M-mode. Each core has a limited number of PMP registers (up to 64), with one reserved to protect TYCHE itself. If an SD configuration cannot be satisfied with the available PMP registers, a synthetic exception is injected and delegated to a parent SD following the interrupt routing protocol. As capability revocations always succeed, the parent can recover regions sent to the child. Like x86_64's I/O-MMU, TYCHE requires IO-PMPs [6], [42] to defend against rogue DMA requests. Like x86_64, interrupts and exceptions can be either trapped or delivered directly to an SD by configuring the `mideleg` and `medeleg` registers. Communication with the monitor occurs via *ecalls* (for S-mode) or illegal instruction traps (for U-mode), as S-mode manages *ecalls* from U-mode within an SD to provide fast system calls.

Running in M-mode imposes more constraints than x86_64 virtualization-based backend, as PMPs are only available in limited supply (8 on our board [92]). The SDs must carefully manage memory to maximize the use of contiguous memory, like existing RISC-V security monitors [25], [35], [68], [110]. Alternatively, future support for H-mode could enable a virtualization-based backend.

6. Integrating Existing Software with TYCHE

The TYCHE SDK is a set of drivers and ports of popular software frameworks that let stock Linux SDs run unmodified workloads as sandboxes, enclaves, or CVMs in separate SDs. These domains can be composed to secure complex cloud deployments. The TYCHE SDK is **not** part of TYCHE's TCB.

6.1. Interfacing with TYCHE from Linux environments

We provide **TYCHE-Capa**, a kernel driver on x86_64 and RISC-V that lets Linux environments in an SD interact with the monitor and create new SDs. TYCHE-Capa abstracts capabilities and exposes a simple interface –*ioctl* commands, file descriptors, and `mmap` – to create, manage, and run SDs. It allocates memory from kernel pools, reserves pages (as in ballooning [102]), and ensures revocation to avoid leaks after crashes. To reduce fragmentation and work around RISC-V's limited PMP entries, it can reserve contiguous physical memory at boot via kernel parameters. TYCHE-Capa schedules SDs as part of their host process, like VMs under KVM [34], and allows them to coexist with OS isolation tools – cgroups, namespaces, and syscall filtering – at the process boundary.

6.2. Backward compatibility

The TYCHE SDK provides backward-compatibility with popular virtual machine monitors (VMMs) and enclave frameworks. It relies on the TYCHE-Capa driver as a compatibility layer to run VMs, CVMs, sandboxes, and enclaves as SDs.

VMs & CVMs with KVM on Intel x86_64: KVM-TYCHE is a fork of the KVM-Intel [34] driver, modified to run VMs and CVMs as child SDs (*not* as nested VMs). The porting effort involved modifying just 400 LOC out of 14.6k LOC, replacing Intel VT-x instructions and EPT management with TYCHE-Capa calls. With this patch, TYCHE can support popular KVM-based VMMs and container frameworks [4], [16], [29], [47].

KVM-TYCHE maintains the runtime behavior of VMs compared to KVM-Intel and thus uses TYCHE's ability to delegate APIC virtualization to a parent SD. For CVMs, additional logic was added to account for the VM's memory and state being unavailable to the host. KVM-TYCHE is backward-compatible with existing VMMs for non-confidential VMs. A small 20-line patch to the LKVM [4] VMM enables confidential VM support by requesting exclusive memory from TYCHE-Capa.

Sandboxes & enclaves on x86 with Gramine:

Gramine [10], formerly Graphene [97], is a library OS (libOS) for running applications inside Intel SGX [58] enclaves; it shields applications by mediating system calls to the untrusted OS. In most cases, Gramine supports unmodified binaries. **Gramine-TYCHE** is a fork of Gramine SGX platform abstraction layer that runs enclaves and sandboxes as SDs in userspace, isolating programs in exclusive or shared memory without Intel SGX hardware. We modified 300 lines of code (LOC) out of 14.9k LOC, replacing Intel SGX logic with TYCHE-Capa calls. The SDK also populates the SD's page tables: it allocates the corresponding physical pages, carves and sends them to the child SD, and initializes its page table root register (`cr3`). Once the enclave SD is sealed, its policy prevents the parent from reading or writing `cr3`, ensuring the parent cannot tamper with the child's address translation.

Gramine-TYCHE does not yet support `fork`. This is a limitation of our port, as doing so requires non-trivial cross-enclave synchronization and state transfer, not an intrinsic constraint of TYCHE. For sandboxing, SDs complement OS-based mechanisms by providing efficient, hardware-enforced, and OS-independent isolation with a reduced TCB. They enable attestation not only of the sandbox boundaries but also of the trusted code implementing higher-level policies (*e.g.,* syscall filtering). Future SDK support could enable sandboxing within enclaves, which is beyond the capabilities of standard OS isolation.

Enclaves on RISC-V with Keystone: Keystone [68] is a RISC-V TEE framework, consisting of a Linux kernel driver and the Eyrie enclave runtime. Porting Keystone to TYCHE required 20 LOC changes to the runtime for compatibility with TYCHE's API, and 150 LOC to the Keystone driver to interact with TYCHE-Capa for creating and managing SDs.

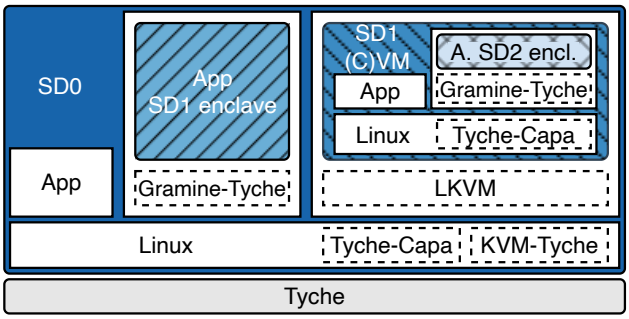

Figure 7: Superposed system and SD views of the deployments benchmarked in Fig. 11. SDs are blue rounded boxes. System abstractions (processes, kernels, and VMs) are full rectangles, and libraries and drivers dotted ones.

6.3. Preserving runtime behavior

TYCHE SDK preserves existing abstraction semantics, allowing fair performance comparisons (§7). Future work could better exploit TYCHE's features – for example, running Gramine's libOS as privileged software to reduce enclave exits [57], [65], or using SDs in a TYCHE-aware VMM for efficient I/O and delegated interrupt handling. Timer interrupts, currently routed to the hypervisor, could instead follow SD policies to avoid VM exits.

7. Evaluation

This section evaluates TYCHE design & performance on x86_64. It reports microbenchmarks for the full-fledged x86_64 implementation and compares it with the prototype on RISC-V (§7.1). It then focuses on x86_64 to measure the performance of real-world applications isolated with various threat models (§7.2), using native execution as a baseline and including Intel SGX enclaves and native VMs as reference points. Case studies (§7.3) then detail the confidential LLaMa CPU-based inference for mutually distrustful model and prompt owners and other isolation deployments enabled by TYCHE.

Experimental setup: On x86_64, TYCHE runs on a 16-core Intel i7-10700 CPU with Intel SGX [58] v1 and an EPC of 94MB, using a stock Linux kernel v6.2 as SD0. On RISC-V, it runs on a StarFive VisionFive2 board [92] (JH7110 SoC) with 8 PMP entries per hart and a stock v5.15 kernel. VMs and CVMs use v6.2 with 4 GiB, 8 cores, and VIRTIO devices [66]. CVMs also enable SWIOTLB [12] to copy I/O into SD0-shared memory. Network experiments use a separate 12-core client [3], [46] with 400 connections over Ethernet. Results average the best 9 of 10 runs.

Naming conventions: We refer to bare-metal Linux as "native" and KVM-based Linux VMs as "native VMs". SD0 runs Linux and creates SD1 VMs, CVMs, and Gramine-based enclaves. SD2 enclaves are nested inside SD1 CVMs and isolated from both SD0 and their parent. SGX enclaves run with Gramine-SGX. Sandbox SDs perform identically to enclave SDs and are omitted. See Fig. 7.

7.1. Microbenchmarks

We measure the average latency (over 1000 ops) of TYCHE's primitives on both platforms to compare hardware

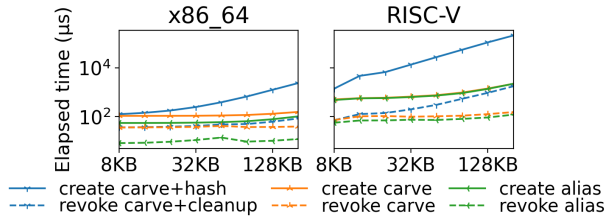

Figure 8: Create & revoke latencies as a function of size for carve, with and without hash & clean, and alias SD memory.

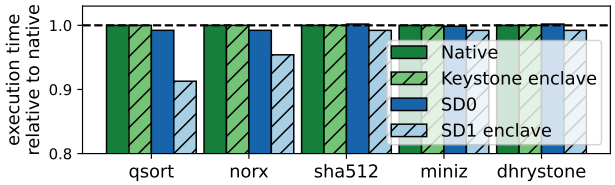

Figure 9: RV8 RISC-V CPU microbenchmarks comparison between native, unmodified Keystone, and Tyche.

mechanisms. We then evaluate CPU and I/O overheads on x86_64 across SD configurations, compared to equivalent native deployment.

Creation/Revocation: Fig. 8 shows carves are more expensive than aliases as they remove memory from SD0 and notify other cores via IPIs (§5.2). This difference is especially noticeable on x86_64, where carves trigger a walk on SD0's EPT. On both platforms, *hash* and *clean* increase latencies with the SD's size, as they require reading and writing memory, respectively.

Switches: In Table 3, SD switches on RISC-V are 3x slower than on x86_64, despite faster privilege layer transitions to/from Tyche. On x86_64, Tyche uses a hardware-software combination to efficiently save and restore SD states, while RISC-V relies entirely on software. Overall, Tyche's switch latencies on x86_64 (1.2µs) and RISC-V (3.9µs) are competitive with related work and hardware extensions [17], [57], [58], [68], [97].

RISC-V prototype: On Fig. 9, Tyche on RISC-V competes with native and unmodified Keystone on most RV8 microbenchmarks, with a <10% slowdown for SD1 enclaves on short-lived programs (qsort, norx) due to the extra indirection to the Tyche-Capa driver.

	Enter+Exit Tyche	Total switch cost
x86_64	0.493 +/- 0.017 µs	1.171 +/- 0.002 µs
RISC-V	0.246 +/- 0.000 µs	3.897 +/- 0.007 µs

Table 3: Average switch latency and standard deviation, including monitor entry+exit (hardware privilege transitions).

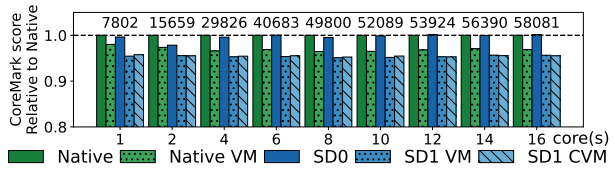

Figure 10: CoreMark-Pro performance relative to bare-metal Linux (Native) for different SD deployments and varying number of cores on x86_64 (raw native score at the top).

CPU overheads on x86_64: Fig. 10 shows CoreMark-PRO results from 1 to 16 cores. SD0 matches bare-metal, with minor slowdowns (up to ~2%) at 2-4-8 cores due to virtualization amplifying cache and TLB effects. SD1 VMs and CVMs show slight overhead: ~1% over native VMs and ~4% over native, mainly from timer interrupt transitions. CVMs add no extra cost over VMs

Multi-(C)VMs: We group the 16 cores into 2-, 4-, or 8-core sets, launching one Linux (C)VM per group (*e.g.,* 8 VMs for 2-core groups). Each VM boots a full Linux kernel, runs two-minutes *sysbench* [82] CPU and memory benchmarks, saves the results, and shuts down:

```
sysbench cpu --threads=$(nproc) --time=120
    run
sysbench memory --time=120 --threads=$(
    nproc)
--memory-block-size=128M --memory-access-
    mode=rnd
--memory-oper=write run
```

CPU throughput (events/sec) and memory throughput (random writes over 128 MiB) show only 2% CPU and <1% memory overhead for SDs versus native VMs. *kvm_stat* [33] shows similar event distributions; the extra cost is due to monitor indirections on timer interrupts and synchronized access to SD0 state. Running two SD (C)VMs per group yields similar overheads compared to native VMs, showing they remain stable under oversubscription.

I/O overheads on x86_64: We measure Redis I/O using memtier [3] with pipeline=1 GETs to avoid masking latency and to expose Tyche's overhead. Table 4 details the full configuration of the networking benchmarks. Fig. 11 and Fig. 12 show throughput and latency across SD configurations. Most native and SD (C)VM overhead comes from the LKVM user-space VIRTIO driver; running native and SD VMs with QEMU (no VHOST) improves throughput by 15%-points, confirming this bottleneck. With QEMU, SD VMs still achieve similar performance to native ones, even with VHOST enabled. This is expected as better drivers (QEMU) and VHOST improve the performance on the common path for both native and SD (C)VMs and are orthogonal to Tyche. Enclave overheads stem from libOS copies: SD2 in particular suffers from compounded I/O paths – VIRTIO, SWIOTLB, and enclave copying. The lack of pipelining degrades the throughput compared to the baseline due to the higher latency introduced by the different levels of isolation, as shown in Fig. 12. In particular, the relative performance of native and SD VMs compared to native or SD0 in Fig. 11 is due to the increased latency: from 0.86ms when running natively to 1.77ms in a VM at p50 on the Redis benchmark. Enabling pipelining hides the latency, we measure a baseline 4.16 millions req/s with pipeline = 30, and 3.54 millions in the native and SD VMs, *i.e.,* a 15% overhead rather than 40%. With lighttpd [64], we observe that increasing payload size amortizes copy and context switching overhead, reaching native and SD0 throughput for 10 KiB payloads.

7.2. Enclaves, VMs, CVMs, & composable isolation

Fig. 11 shows Tyche's performance isolating real-world applications on x86_64. We use native execution as the

Server	Benchmark client	Threads	Connections	Workload configuration	Pipeline	Termination condition
redis	memtier	12	400	10% write, 90% read	1	20 million requests
hyper	wrk	12	400	12B payload + 76B headers	1	2 minutes
lighttpd	wrk	12	400	10kiB payload	1	2 minutes

Table 4: Network benchmarks configuration

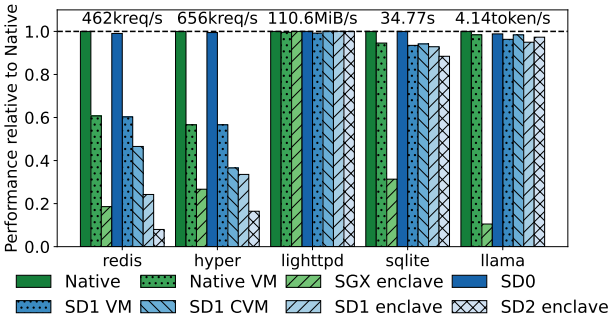

Figure 11: Performance relative to native execution (green) for real-world applications isolated as SDs on x86_64 (blue).

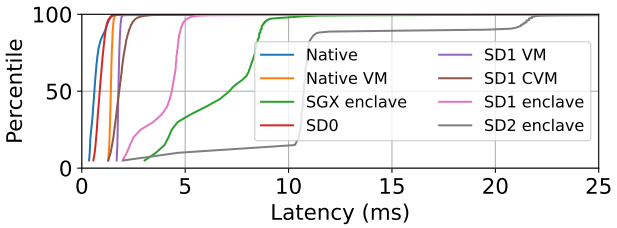

Figure 12: Redis GET latency distribution as measured by memtier (max throughput) during Fig. 11's experiment.

baseline and include native VMs and SGX enclaves as reference points.

Applications: We evaluate Lighttpd [64], Hyper [54], SQLite [49], Redis [5], and llama-cpp [45]. Lighttpd and SQLite are common enclave workloads [17], [97] and appear in nested enclave studies such as Veil [17], enabling direct comparison. Hyper is a high-performance, multi-threaded Rust HTTP server that scales with worker threads rather than processes [94], avoiding Gramine-Tyche's unsupported fork. HTTP servers are benchmarked with wrk [46], Redis with memtier, SQLite with speedtest1 [49] at size 1000 (which includes 500K inserts), and LLaMa reports throughput [45] after generating 1000 tokens.

Configuration: Binaries are from Gramine's default examples except for LLaMa (§7.3). Gramine applications use a *manifest* specifying the number of threads and enclave memory. We use default manifests [83] with *exitless* [9] disabled due to CVE-2022-21233 and CVE-2022-21166. All enclaves include at least one worker and three Gramine runtime threads [11] and 256MiB to 4GiB total memory. For Tyche, we supply one physical core per manifest thread and configure native execution with the same worker-thread count. Gramine-SGX and Gramine-Tyche use the same application binaries, which we compiled from source without modification.

SD0: SD0 shows no overhead in any benchmark compared to native, indicating that Tyche does not impact the performance of Linux applications. This result is expected, as there are no calls or exits to the monitor and SD0 has direct access to devices and the APIC.

VMs & CVMs: SD1 VMs match the performance of native VMs, demonstrating that KVM-Tyche can support existing VM deployments. For I/O benchmarks (*e.g.,* Hyper and Redis), SD1 CVMs experience additional overheads due to extra copies through bounce buffers for I/O (see §7.1). They otherwise perform similarly to native VMs.

Enclaves: SD1 enclaves outperform SGX enclaves across all applications and are close to native for non-latency sensitive applications. SGX performs poorly on LLaMa, even compared to an SD2 enclave. This is likely due to the limited EPC (94MB) size in SGX v1 compared to the high memory usage (4GiB) for the model [52], [77] and llama-cpp contexts [45] and possibly memory encryption and integrity protection, thus we would expect SGX v2 to perform better. Regardless, SD2 only incurs a reasonable overhead compared to our baseline native execution as long as extra memory copies can be amortized.

Memory usage: Tyche requires private memory to store capabilities. Running these workloads, we measured an average of two 4KiB pages per SD used for storing capabilities. This extrapolates to, *e.g.,* a 4MiB memory overhead for 512 SDs.

Discussion: The latency of network operations is the primary issue but not a direct consequence of Tyche's design, as discussed in §7.1. This is a known problem with confidential environments [42], [69], [71], [80] that could be alleviated in Tyche, *e.g.,* with safe device passthrough. Tyche otherwise demonstrates that existing frameworks can be ported to run SDs, achieving performance comparable to equivalent native deployments and outperforming Intel SGX, even in nested cases, reinforcing our claim of backward compatibility.

7.3. Case Studies

Private inference with controlled sharing: This case study uses Fig. 2's scenario to protect user prompts from the CSP and model owner, while keeping model weights secure from the user and CSP. The CSP runs a Linux SD0 hypervisor with KVM-Tyche; the user runs an SD1 CVM that creates an SD2 enclave for CPU-based LLM inference using llama-cpp [45] with Gramine-Tyche and four threads. The "proprietary model" is an encrypted Meta LLaMa 3.2 Instruct model [52], [77], decrypted inside the enclave as it is read from disk. The enclave has no file or network access and shares memory only with the user's SD1 CVM, which receives prompts over SSH and forwards them to the enclave. User keys can also be stored in a separate enclave. As shown in Fig. 11, the SD2 enclave runs near native speed (~2% overhead), is 10× faster than SGX enclaves, and goes beyond prior industrial solutions [21], [40] by

also protecting the model from the user. Future work on KVM-Tyche may explore secure GPU passthrough.

Managing trust within user-applications: Tyche's SDs are orthogonal to privilege levels and can compose isolation even within user-level code. We prototyped compartmentalization of code within a user SD enclave. While we did not provide a full port of an intra-process isolation framework, such a port could be done in the future; this prototype demonstrates the concept and fine-grained control over application TCB.

In this experiment, a user enclave running bearSSL [1] hosts Redis inside a nested child SD, implemented either as a sandbox or a nested enclave. Both SDs minimize their TCB by including only a modified musl [13] libc, removing unnecessary system calls, and communicating through shared-memory queues. Before decrypting and forwarding requests to Redis, the SSL enclave attests that it fully encapsulates the Redis SD. Responses are returned via the shared queue and encrypted before being sent back.

A remote client (memtier) issues requests over a TLS connection with the SSL enclave. This experiment illustrates Tyche's generality, showing that arbitrary composition and nesting of isolation boundaries is possible, even entirely within userspace.

Sharing without the cost: In existing cloud abstractions, communication between trusted TEEs – enclaves or CVMs – must bounce through untrusted memory, incurring software encryption, replay protection, integrity checks, and data copies to defend against tampering by untrusted software. Tyche instead enables SDs to establish attestably private shared memory, accessible only to the two endpoints, eliminating this entire overhead. This simplifies communication, improves performance, and enables confidential service VMs—trusted VMs that expose devices or services directly to other confidential endpoints.

We compare direct SD–to-SD communication with the traditional integrity-protect, encrypt, bounce, decrypt path. Using AES-GCM and SHA-256 from the standard Linux crypto library, we measure added latencies of 20µs for 1KB messages, 2ms for 1MB, and 0.24s for 100MB. With Tyche, these costs are avoided entirely. Physical attacks are orthogonal and can be handled in hardware, *e.g.,* with single- or multi-key total memory encryption (MK-TME) [56], as discussed in §3.2.

8. Related Work

Tyche combines a security monitor design with an API inspired by micro-kernel to deliver a composable isolation solution.

Security monitors: They are trusted intermediaries [87] that enhance systems with new isolation and security guarantees [32], [43], [108] and are easier to inspect, update and extend than hardware solutions [27]. Arm CCA uses a monitor to support confidential VMs [73] and Komodo [43] enables enclaves in Arm TrustZone [22]. Security monitors are popular in confidential computing to restrict privileged software, often through the use of virtualization, *e.g.,* Inktag [51], Overshadow [32], and HyperEnclave [61] protect enclaves from untrusted OSes, Cloudvisor [108] isolates VMs from hypervisors, and Blackbox [50] secures containers. They can offer intra-VM isolation, *e.g.,* Veil [17] implements enclaves in AMD CVMs [19] while Erebor [107] focuses on sandboxes in Intel CVMs [57], or can harden kernel integrity [24], [36], [89].

Hardware vendors incrementally added privilege hierarchies within TEEs that can host such monitors – Intel TDX 1.5 [60] introduces L1/L2 partitions, allowing a trusted L1 VMM to enforce intra-isolation in a TEE for up to three L2 VMs. However, this remains subject to the limitations of intra-isolation discussed in this paper: the mechanism is platform-specific, hardware-capped in the number of sub-components, and scoped to a single CVM.

Tyche adopts a similar monitor-based implementation but differs in scope and abstraction. It provides a unified isolation abstraction, SDs, that subsumes and composes the isolation boundaries traditionally enforced by separate security monitors, and applies to *all* software on the machine – not only components within a CVM but also across independently deployed VMs. This design is inspired by micro-kernels.

Micro-kernels: Micro-kernels [63], [74], [95] follow a minimalist design, including only essential functions that cannot be implemented in user space, such as virtual memory, IPC, and thread management. This reduces kernel complexity, improving security and maintainability. By separating mechanisms from policies [67], micro-kernels provide flexibility in resource management and isolation. For instance, while the kernel handles page tables, user-space components configure virtual address spaces, enabling the *recursive construction of address spaces* [74] and custom isolation abstractions

Tyche leverages key micro-kernel concepts to deliver modern isolation guarantees. Its memory operations resemble L4 [74]'s *grant*, *map*, and *flush*. They however differ in that Tyche's operations focus on **attestable** isolation through guarantees on resource management: distinguishing explicitly between shared and exclusive resources and attesting memory is measured when received or scrubbed upon revocation (§4). Tyche also draws inspiration from Fluke [44], which enables recursive VM isolation through "nested processes" without the overhead of naive nested virtualization. Tyche generalizes this approach by providing a foundational mechanism to unify compartmentalization and compose confidential computing and encapsulation.

Despite its similarities to micro-kernel APIs, Tyche is *not* a kernel replacement, does not create directly usable system abstractions such as processes, and requires an untrusted kernel in SD0 to drive the machine. The principles behind Tyche's attestable isolation could however be backported into existing security-oriented micro-kernels [63].

Virtualization: Tyche is not a hypervisor even though it runs in root-mode and uses virtualization extensions on x86_64. It does *not* virtualize resources, provide full machine abstraction, or take allocation or scheduling decisions. Instead, Tyche is closer to an exokernel [41] or a trusted state machine: it validates operations against policies encoded by capabilities to correctly enforce and attest the partitioning and sharing of resources across SDs. Tyche shares some similarities with hypervisors that adopt micro-kernel principles to improve security [72], [93], but these efforts focus on minimizing the hypervisor's TCB rather than providing a unified isolation abstraction to compose security boundaries. Unlike type-1 hypervisors [26], [100],

which grant extra privileges to their initial domain (e.g., DOM0 in Xen [26] or the root partition in Hyper-V [100]), SD0 in Tyche has no special privileges.

Other mechanisms & platforms: Our RISC-V prototype in M-mode demonstrates that Tyche does not require virtualization and can adapt to simpler access control mechanisms [18], [62], [85], [104]. Tyche could use alternative architectures [62], [85], like NoHype [62], that eliminate virtualization and provide simpler primitives to partition I/O, memory, and cores. CHERI [104] capabilities offer a promising alternative to page-based mechanisms, already supporting enclaves [99] and secure embedded devices [18]. Tyche's software-defined capabilities provide a global view of system resources with extensible policies, complementing CHERI's efficient hardware enforcement for fine-grained memory isolation.

While we did not port Tyche to Arm, a virtualization-based backend could be implemented, running the monitor in EL2, similar to Blackbox [50], in normal, secure [73], or realm world [73]. Alternatively, it could run as firmware in EL3, akin to the RISC-V backend, and leverage Granule Protection Table [73] to isolate memory. Similarly, Tyche could conceptually run as an Intel TDX [57] module, providing attestable policies for resource management and private shared memory across CVMs. In practice, however, Intel is unlikely to allow custom TDX module implementations. A more realistic alternative would be to deploy Tyche as an L1 VMM within a TDX CVM to isolate SDs running as L2s. This would bring clear semantics for isolation, controlled sharing within a CVM, and multi-component attestation, but would be limited to three L2s and would remain less general than our current approach, which provides these benefits at the scale of the entire machine rather than a single CVM and is portable across hardware platforms.

9. Conclusion

Tyche shows that trust management can become a first-class cloud abstraction. Its security domain abstraction unifies composable isolation across privilege layers and system abstractions, shifting the burden of implementing and enforcing fine-grained isolation away from tenants. We showed that Tyche is both general, by supporting and composing enclaves, sandboxes, and confidential virtual machines, and practical by running unmodified applications with near-native performance. A confidential-inference case study with mutually distrustful users, model owners, and CSPs demonstrated its applicability to modern cloud workloads with complex trust models.

Acknowledgements

We thank the many anonymous reviewers and shepherd, Nate Foster, James R. Larus and Matthias Payer for their valuable feedback. This work has received funding from the Swiss State Secretariat for Education, Research, and Innovation (SERI) under the SwissChips initiative, from the Microsoft-EPFL Joint Research Center, and gifts from the VMware University Research Fund.

References

[1] Bearssl. https://bearssl.org/.

[2] Hyper-V patches for supporting AMD SEV-SNP enlightened guest. https://lore.kernel.org/lkml/20230627032248.2170007-1-ltykernel@gmail.com/T/.

[3] Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark.

[4] Native kvm linux tool. https://github.com/lkvm/lkvm.

[5] Redis. https://github.com/redis/redis.

[6] Risc-v iopmp specification. https://github.com/riscv-non-isa/iopmp-spec.

[7] The Rust programming language. https://www.rust-lang.org/.

[8] Web assembly. https://webassembly.org/.

[9] Gramine: Exitless. https://gramine.readthedocs.io/en/stable/manifest-syntax.html?highlight=exitless#number-of-rpc-threads-exitless-feature, 2020.

[10] The gramine library os. https://gramineproject.io/, 2020.

[11] Gramine: threading. https://gramine.readthedocs.io/en/stable/manifest-syntax.html?highlight=num%20threads#number-of-threads, 2020.

[12] Linux - DMA and swiotlb. https://docs.kernel.org/core-api/swiotlb.html, 2021.

[13] musl libc. https://www.musl-libc.org/, 2022.

[14] TEE Device Interface Security Protocol (TDISP). https://pcisig.com/tee-device-interface-security-protocol-tdisp, August 2022.

[15] Advanced Micro Devices, Inc. AMD-V™ Nested Paging. http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf, 2008.

[16] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 419–434, 2020.

[17] Adil Ahmad, Botong Ou, Congyu Liu, Xiaokuan Zhang, and Pedro Fonseca. Veil: A Protected Services Framework for Confidential Virtual Machines. In *Proceedings of the 28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVIII)*, pages 378–393, 2023.

[18] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert M. Norton, Simon W. Moore, Yucong Tao, Robert N. M. Watson, and Hongyan Xia. CHERIoT: Complete Memory Safety for Embedded Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 641–653, 2023.

[19] AMD. Sev-snp: Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.

[20] AMD. Amd64 architecture programmer's manual volume 2: System programming. 2023.

[21] Apple. Private cloud compute: A new frontier for ai privacy in the cloud. https://security.apple.com/blog/private-cloud-compute/, June 2024.

[22] ARM. Building a secure system using trustzone technology. *White Paper, April*, 2009.

[23] AWS. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/. Accessed: 2026-03-12.

[24] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 90–102, 2014.

[25] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proceedings of the 30th USENIX Security Symposium*, pages 1073–1090, 2021.

[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[27] Andrew Baumann. Hardware is the new Software. In *Proceedings of The 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)*, pages 132–137, 2017.

[28] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, 2015.

[29] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track*, pages 41–46, 2005.

[30] Charly Castes and Andrew Baumann. Sharing is leaking: blocking transient-execution attacks with core-gapped confidential vms. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24)*, 2024.

[31] Charly Castes, Adrien Ghosn, Neelu Shivprakash Kalani, Yuchen Qian, Marios Kogias, Mathias Payer, and Edouard Bugnion. Creating Trust by Abolishing Hierarchies. In *Proceedings of The 19th Workshop on Hot Topics in Operating Systems (HotOS-XIX)*, pages 231–238, 2023.

[32] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey S. Dwoskin, and Dan R. K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, pages 2–13, 2008.

[33] Linux Kernel Community. kvm_stat: KVM statistics reporting tool. https://www.kernel.org/doc/html/latest/virt/kvm/tools/kvm_stat.html, 2024.

[34] The Linux Kernel Community. Linux kernel virtual machine. https://linux-kvm.org/page/Main_Page, 2007.

[35] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of the 25th USENIX Security Symposium*, pages 857–874, 2016.

[36] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 191–206, 2015.

[37] Microsoft Azure Research & EPFL DCSL. Tyche github repository. https://github.com/epfl-dcsl/tyche-devel.

[38] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium*, pages 1441–1458, 2018.

[39] Jules Drean, Fisher Jepsen, Edward Suh, Srini Devadas, Aamer Jaleel, and Gururaj Saileshwar. Teaching an Old Dog New Tricks: Verifiable FHE Using Commodity Hardware. *Proc. Priv. Enhancing Technol.*, 2025(3):282–303, 2025.

[40] Edgeless. Edgeless continuum ai. https://docs.edgeless.systems/continuum/0.3/overview, 2024.

[41] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

[42] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, Wenbin Zheng, Siqi Zhao, and Haibo Chen. sIOPMP: Scalable and Efficient I/O Protection for TEEs. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*, pages 1061–1076, 2024.

[43] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, 2017.

[44] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating System Design and Implementation (OSDI)*, pages 137–151, 1996.

[45] Georgi Gerganov. Llama-c++. https://github.com/ggerganov/llama.cpp, 2024.

[46] Will Glozer. Wrk - a http benchmarking tool. https://github.com/wg/wrk, 2021.

[47] gVisor Authors. gvisor: The container security platform. https://gvisor.dev/, 2023.

[48] Norman Hardy. The Confused Deputy (or why capabilities might have been invented). *ACM SIGOPS Oper. Syst. Rev.*, 22(4):36–38, 1988.

[49] D. Richard Hipp. Sqlite. https://www.sqlite.org/, 2000.

[50] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th Symposium on Operating System Design and Implementation (OSDI)*, pages 683–700, 2022.

[51] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, pages 265–278, 2013.

[52] hugging quants. Llama-3.2-1b-instruct quantized. https://huggingface.co/hugging-quants/Llama-3.2-1B-Instruct-Q4_K_M-GGUF, September 2024.

[53] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 533–549, 2016.

[54] Hyperium. Rust hyper. https://hyper.rs/, 2024.

[55] Intel. Trusted execution technology. https://www.intel.com/content/www/us/en/developer/articles/tool/intel-trusted-execution-technology.html, 2014.

[56] Intel. Multi-key total memory encryption. https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/002/intel-multi-key-total-memory-encryption/, 2017.

[57] Intel. Architecture specification: Intel trust domain extensions (intel tdx) module. https://software.intel.com/content/dam/develop/external/us/en/documents/intel-tdx-module-1eas.pdf, 2023.

[58] Intel. Intel software guard extensions (intel sgx). https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html, 2023.

[59] Intel. Intel virtualization technology for directed i/o, architecture specification. https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html, 2023.

[60] Intel. Intel TDX module v1.5 – TD partitioning architecture specification. https://www.intel.com/content/www/us/en/content-details/773039/intel-tdx-module-v1-5-td-partitioning-architecture-specification.html, 2024.

[61] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. HyperEnclave: An Open and Cross-platform Trusted Execution Environment. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*, pages 437–454, 2022.

[62] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, pages 350–361, 2010.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[64] Jan Kneschke. Lighttpd. https://www.lighttpd.net/, 2003.

[65] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-tdx: A lightweight os kernel for confidential vms. In *ACM Conference on Computer and Communications Security (CCS)*, October 2024.

[66] KVM. Virtio. https://www.linux-kvm.org/page/Virtio.

[67] Butler W. Lampson and Howard E. Sturgis. Reflections on an Operating System Design. *Commun. ACM*, 19(5):251–265, 1976.

[68] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the 2020 EuroSys Conference*, pages 38:1–38:16, 2020.

[69] Hugo Lefeuvre, David Chisnall, Marios Kogias, and Pierre Olivier. Towards (Really) Safe and Fast Confidential I/O. In *Proceedings of The 19th Workshop on Hot Topics in Operating Systems (HotOS-XIX)*, pages 214–222, 2023.

[70] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Alexander Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 234–251, 2017.

[71] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC)*, pages 1–15, 2023.

[72] Shih-Wei Li, John S. Koh, and Jason Nieh. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *Proceedings of the 28th USENIX Security Symposium*, pages 1357–1374, 2019.

[73] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. Design and Verification of the Arm Confidential Compute Architecture. In *Proceedings of the 16th Symposium on Operating System Design and Implementation (OSDI)*, pages 465–484, 2022.

[74] Jochen Liedtke. On micro-Kernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–250, 1995.

[75] LLVM. libfuzzer: A library for in-process, coverage-guided fuzzing. https://llvm.org/docs/LibFuzzer.html, 2021.

[76] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[77] Meta-LLama. Llama-3.2-1b-instruct. https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct, September 2024.

[78] Microsoft. OpenHCL. https://openvmm.dev/guide/user_guide/openhcl.html. Accessed: 2026-03-12.

[79] Microsoft. Introducing hyperlight: Virtual machine-based security for functions at scale. https://opensource.microsoft.com/blog/2024/11/07/introducing-hyperlight-virtual-machine-based-security-for-functions-at-scale/, 2024.

[80] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 2017 EuroSys Conference*, pages 238–253, 2017.

[81] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020.

[82] Alexy Kopytov Peter Zaitsev. Sysbench: a scriptable multi-threaded benchmark tool. https://github.com/akopytov/sysbench, 2024.

[83] Gramine Project. Gramine CI Examples. https://github.com/gramineproject/gramine/tree/master/CI-Examples, 2024.

[84] RISC-V Foundation. RISC-V SBI specification. https://github.com/riscv-non-isa/riscv-sbi-doc, 2023.

[85] Michael Roitzsch, Till Miemietz, Christian von Elm, and Nils Asmussen. Software-Defined CPU Modes. In *Proceedings of The 19th Workshop on Hot Topics in Operating Systems (HotOS-XIX)*, pages 23–29, 2023.

[86] Rust Foundation. The rustonomicon - meet safe and unsafe. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html, 2023.

[87] Jerome Saltzer and M Frans Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[88] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proc. IEEE*, 63(9):1278–1308, 1975.

[89] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 335–350, 2007.

[90] AMD Sev-Snp. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 53:1450–1465, 2020.

[91] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 170–185, 1999.

[92] StarFive. Visionfive 2 riscv board. https://www.starfivetech.com/en/site/boards, 2024.

[93] Udo Steinberg and Bernhard Kauer. NOVA: a microhypervisor-based secure virtualization architecture. In Christine Morin and Gilles Muller, editors, *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 209–222. ACM, 2010.

[94] Igor Sysoev and Inc. NGINX. nginx: High Performance Load Balancer, Web Server, & Reverse Proxy. https://nginx.org/, 2024. Accessed: 2025-07-25.

[95] Andrew S. Tanenbaum. *Operating systems: design and implementation.* Prentice-Hall software series. Prentice-Hall, 1987.

[96] Trusted Computing Group. Trusted Platform Module (TPM) – ISO/IEC 11889. https://www.iso.org/standard/66510.html, 2015.

[97] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 2014 EuroSys Conference*, pages 9:1–9:14, 2014.

[98] Richard Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[99] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. Cheri-tree: Flexible enclaves on capability machines. In *EuroS&P-8th IEEE European Symposium on Security and Privacy*. IEEE, 2023.

[100] Anthony Velte and Toby Velte. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[101] Stavros Volos, Cédric Fournet, Jana Hofmann, Boris Köpf, and Oleksii Oleksenko. Principled microarchitectural isolation on cloud cpus. In *ACM Conference on Computer and Communications Security (CCS)*, October 2024.

[102] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI)*, 2002.

[103] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, pages 29–46, 2010.

[104] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[105] Zachary Yedidia. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIX)*, pages 649–665, 2024.

[106] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*, pages 79–93, 2009.

[107] Chuqi Zhang, Rahul Priolkar, Yuancheng Jiang, Yuan Xiao, Mona Vij, Zhenkai Liang, and Adil Ahmad. Erebor: A drop-in sandbox solution for private data processing in untrusted confidential virtual machines. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1210–1228, 2025.

[108] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 2011.

[109] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A Verified Security Module for Confidential VMs. In *Proceedings of the 18th Symposium on Operating System Design and Implementation (OSDI)*, pages 599–614, 2024.

[110] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *Proceedings of the 17th Symposium on Operating System Design and Implementation (OSDI)*, 2023.