Rethinking the confidential cloud through a unified low-level abstraction for composable isolation

Adrien Ghosn*
Azure Research, Microsoft
Cambridge, United Kingdom

Charly Castes*
EPFL
Lausanne, Switzerland

Neelu S. Kalani
EPFL
Lausanne, Switzerland

Yuchen Qian
EPFL
Lausanne, Switzerland

Marios Kogias
Imperial College London
Azure Research, Microsoft
London, United Kingdom

Edouard Bugnion
EPFL
Lausanne, Switzerland

Abstract

Securing sensitive cloud workloads requires composing confidential virtual machines (CVMs) with nested enclaves or sandboxes. Unfortunately, each new isolation boundary adds ad-hoc access control mechanisms, hardware extensions, and trusted software. This escalating complexity bloats the TCB, complicates end-to-end attestation, and leads to fragmentation across platforms and cloud service providers (CSP).

We introduce a unified isolation model that delegates enforceable, composable, and attestable isolation to a single trusted security monitor: Tyche. Tyche provides an API for partitioning, sharing, attesting, and reclaiming resources through its core abstraction, trust domains (TDs). To provide fine-grain isolation, TDs can recursively create and manage sub-TDs. Tyche captures these relationships in attestations, allowing cloud tenants to reason about end-to-end security. TDs serve as the building blocks for constructing composable enclaves, sandboxes, and CVMs.

Tyche runs on commodity x86_64 without hardware security extensions and can maintain backward compatibility with existing software. We provide an SDK to run and compose unmodified workloads as sandboxes, enclaves, and CVMs with minimal overhead compared to native Linux execution. Tyche supports complex cloud scenarios, such as confidential inference with mutually distrustful users, model owners, and CSP. An additional RISC-V prototype demonstrates Tyche's portability across platforms.

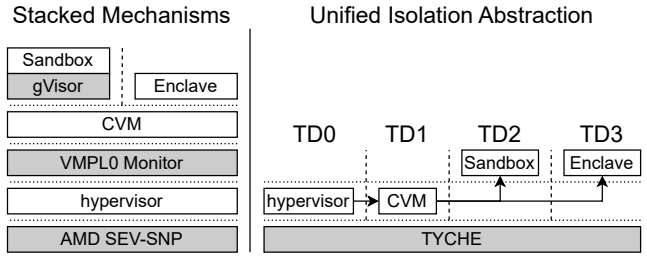

Figure 1. Example deployment of an enclave and a sandbox within a CVM. Left is based on AMD SEV-SNP, right is our solution, with management dependencies as arrows.

1 Introduction

Cloud deployments need to address complex threat models with many possible attack vectors. Consider the realistic deployment [37] of Figure 1 (left), that prevents prompt leakage from a confidential cloud LLM inference. It relies on the different isolation mechanisms in grey: AMD SEV-SNP [15] confidential virtual machine (CVM) protects the guest OS and its applications from the untrusted hypervisor. A software monitor [13] operates at the highest AMD VM privilege Level [13] (VMPL) within the CVM to implement nested enclaves that protect user private keys from a bug or backdoor [17, 74, 80, 81] in the OS. Meanwhile, gVisor [46, 104] uses Linux ptrace [47] to sandbox a potentially compromised LLM runtime [60, 82, 84, 91] that might leak prompts.

As each isolation mechanism protects against only a single attack vector, they must be stacked to compose their isolation guarantees and achieve end-to-end security for the workload. However, this stacking introduces several limitations: (1) It increases the trusted computing base (TCB), as each isolation boundary adds its own ad hoc mechanism; (2) It complicates attestation, as no mechanism explicitly captures the isolation guarantees enforced across all boundaries; (3) It fragments the cloud ecosystem, as porting the workload to a different CSP [18, 45] or platform [57, 72] might necessitate an entirely different set of vendor or hardware-specific mechanisms [47, 78, 79, 105]. Moreover, some mechanisms might simply not compose, *e.g.,* OS or hypervisor-based sandboxes within enclaves or CVMs, and have to be replaced [53, 103].

Our insight is that, although existing mechanisms and abstractions address different threats, they share a common goal: to restrict and attest a software component's access to the machine and other components' resources – that is, to control shared and exclusive access. Rather than stacking mechanisms and abstractions for each threat, we argue for a common abstraction that can adapt to diverse threat models by making shared and exclusive access explicit and attestable. Isolation enforcement can then be delegated to

*co-equal first author.

a single trusted entity that provides this common abstraction, along with a small set of primitives to compose and attest isolation. We further argue this can be achieved using standard access control mechanisms – without hardware security extensions [15, 57, 72].

This paper presents the **Tyche** security monitor. Tyche provides a unified low-level isolation abstraction – trust domains (TDs[1]) – as the building block to construct and compose higher-level isolation abstractions. TDs are execution environments with access to a restricted subset of the machine's resources. Unlike previous solutions that target specific threat models, Tyche's API lets software contain threats by controlling and attesting how resources are shared, partitioned, and reclaimed among TDs, enabling the construction of custom isolation boundaries. To compose isolation, TDs use Tyche's API to recursively create child TDs by subdividing or sharing their resources. Tyche tracks and attests TD's resources, explicitly distinguishing between shared and exclusive ones. Tyche thus replaces previous ad-hoc mechanisms: TD-exclusive resources enable confidential abstractions, such as enclaves and CVMs, while control over TD interactions and shared resources enables sandboxes.

Tyche runs on bare metal at the highest privilege level, is attested by a hardware root of trust [95], and enforces the attestable isolation of other software running as TDs. Tyche is implemented in Rust [5], designed for portability across platforms, and enforces isolation using standard hardware access control mechanisms, without extensions for confidential computing. We present a full implementation on x86_64 using virtualization technologies [59, 97] and a prototype running as firmware on RISC-V [85]. For software compatibility, our Tyche SDK enables TDs running unmodified Linux to create sandboxes, enclaves, or CVMs as separate TDs and replaces existing mechanisms with Tyche as shown on Figure 1 (right).

Our evaluation on x86_64 shows that Tyche's isolation model does not compromise performance. Tyche's TDs running unmodified real-world webservers, a key-value store, a database, and an LLM inference as enclaves, CVMs, and enclaves nested in CVMs, achieve near-native performance. We demonstrate that Tyche provides composable isolation for confidential LLM inference with a complex threat model, where the user, model owner, and CSP are mutually distrustful, with only a ~2% slowdown compared to bare metal Linux.

We make the following contributions: (1) a unified solution for composable and attestable isolation that eliminates the need to stack disparate mechanisms to secure cloud deployments; (2) Tyche's implementation that achieves good performance on commodity hardware and is portable across platforms; (3) a demonstration that existing workloads can be ported to run and address complex threat models on Tyche.

[1]Unrelated to Intel TDX [57]

Tyche is an open-source research prototype [33], which has been successfully used by independent researchers to achieve custom isolation guarantees on legacy hardware [36].

2 Background

Compartmentalization & Confidential Computing: While often treated separately, these are two sides of the same coin [27]. Compartmentalization protects software and sensitive information on a machine from an untrusted component, whereas confidential computing protects a trusted software component processing sensitive information from other software on the machine or even in some [15, 57, 58] but not all [28, 61, 76, 106] cases, physical access. Both achieve their goals by controlling access to resources. Compartmentalization restricts an untrusted component's access to the machine and its interactions with other software via **shared** resources or interfaces. Confidential computing ensures integrity and confidentiality, either through **exclusive** access [50, 58, 76] or via memory encryption and integrity protection [15, 57]. For both types of isolation, attestation allows a local or remote verifier to confirm that software is correctly isolated and establish trust in the deployment on a particular machine. Attestation relies on a hardware root of trust, such as the CPU [15, 57, 58] or a Trusted Platform Module [95] (TPM).

Monitors: Monitors are small software components that conveniently *retrofit* [28, 106] new isolation boundaries into the existing software stack with minimal disruption by running at higher privilege levels, *e.g.,* through virtualization [28, 61, 76, 106]. 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 in both confidential computing [13, 28, 41, 50, 61, 106] and compartmentalization [20, 89, 105] to enforce fine grained access control.

Capabilities: 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 [14, 102] – that encapsulate 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.

Composable isolation: The ability to assemble isolation boundaries in a way that preserves and combines their security guarantees.

3 Tyche Overview

This section provides an overview of Tyche's design, its threat model, and an example deployment in Figure 2.

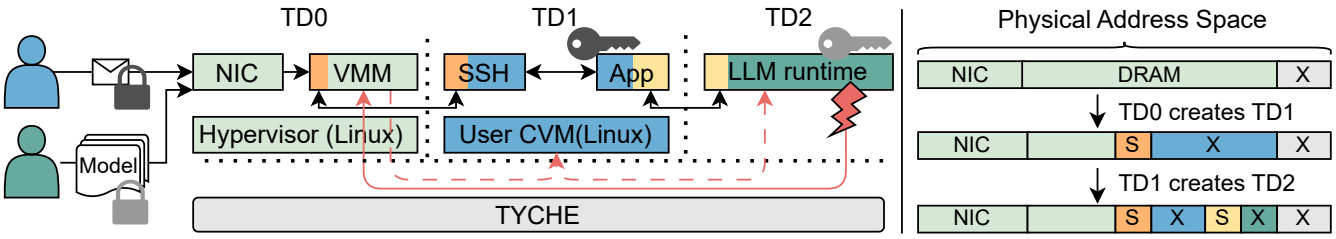

Figure 2. Mutual-distrust LLM inference deployment on top of Tyche; right-side represents physical memory available to each TD, distinguishing exclusive regions (X) and shared ones (S). The encrypted model, prompts, and replies are passed through memory along the black path, interrupts are routed as shown by red arrows.

3.1 Architecture

The Tyche security monitor exposes an API that uses capabilities to create and isolate its core abstraction: trust domains (TD). All software other than the monitor runs within TDs. A TD is an execution environment, whose configuration and owned capabilities grant it access to a subset of the machine's resources, attested by Tyche as either shared or exclusive: cores or devices the TD can run on, access to physical memory regions, and interrupts it can handle or must delegate to other TDs. Via the monitor's API, software running within a TD can create child TDs, partitioning or sharing subsets of its resource with its children, and transfer control to them on selected cores. TDs do not address a fixed threat model but rather provide a unified low-level abstraction for attestable control over shared and exclusive resources. This enables them to accommodate the isolation policies and threat models of existing abstractions – such as sandboxes, enclaves, and CVMs – while also supporting their composition.

Tyche's API decouples resource management from the attestable enforcement of access restrictions. Capabilities abstract resources and TD interactions, allowing TDs to implement their desired isolation boundaries and scheduling policies. Through its capability implementation, Tyche tracks the allocation of resources to TDs and attests whether they are shared or exclusive, hence capturing and enforcing explicitly the resource and management dependencies between TDs. Tyche provides guarantees to both parent and child TDs. The parent retains the ability to reclaim resources or regain control on a core, while children are assured that exclusive resources remain so unless they share them, and that their revocation and control transfers do not leak information.

Tyche supports local and remote attestations to make TD relationships explicit and attest end-to-end security guarantees. A TD attestation has two parts: (1) a TPM [95] root of trust measurement of the boot process, binding an Tyche binary to a physical machine and public key, ensuring it runs alone at the highest privilege level; and (2) a report generated by Tyche, signed with its corresponding private key, that describes the TD's resources, whether they are shared, how they can be reclaimed, and how they are delegated or shared with child TDs.

Tyche's isolation model unifies confidential computing and compartmentalization, enabling the composition of their security guarantees. Confidentiality and integrity follow from exclusive access, while compartmentalization is enforced by restricting a child TD to only interact and share resources with trusted TDs to prevent information leakage. Composition is achieved through the recursive construction of TDs, allowing each domain to define its own isolation boundaries.

Tyche is designed to be ported across hardware platforms, with platform-independent capabilities enforced by hardware-specific **backends**. On x86_64, Tyche enforces isolation using virtualization extensions (Intel VT-x [97]), with extended page tables and I/O-MMU [59] to protect memory. On RISC-V, the monitor runs as firmware in M-mode and uses Physical Memory Protection [85] (PMP). On both platforms, the Tyche SDK provides kernel drivers for unmodified Linux environments to run on top of Tyche and use trust domains to build and nest sandboxes, enclaves and CVMs.

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 a TD, 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 TD 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.

Guarantees for TDs: Tyche is part of the TCB of all TDs and supports a diverse range of trust and threat models between TDs, based on how their resources overlap and can be reclaimed. The monitor ensures correct, attestable isolation of TDs, restricting their access to assigned resources, explicitly reporting shared ones, preventing exclusive ones from transparently becoming shared, and allowing reclamation

of resources allocated to children. Tyche also prevents leaks during control transfers and resource reclamation, following the TD's configuration.

Out-of-scope: Physical attacks, such as accessing DRAM or the PCI bus to read the monitor or a TD'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 [10]. **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 [26, 108] and physical memory allocation based on cache-coloring [35, 100]. **Denial-of-service** attacks to exhaust the monitor's memory are possible, but they only prevent the creation of new TDs and do not prevent the revocation, attestation, or isolation enforcement of existing ones. They can be further mitigated by requiring TDs to supply memory for the monitor's metadata. Inherent to the cloud, the CSP can deny service by turning off the platform [24].

3.3 Running Example

Figure 2 illustrates a public cloud deployment where a user performs LLM inference using a proprietary model. This scenario has a more complex threat model than Figure 1 as it involves full mutual distrust: the user does not trust the CSP or model owner with confidential prompts, the model owner does not trust the user or CSP with the model weights, and the CSP does not trust either party. Such private inference requires confidential computing, to protect prompts and model weights, composed with compartmentalization to prevent the LLM runtime from leaking prompts.

Tyche controls all machine resources at boot and assigns them through capabilities to TD0, the CSP's hypervisor running Linux+KVM [30] and Tyche SDK. To instantiate a CVM for the user, TD0 interacts with Tyche via the SDK, partitioning its memory to create TD1 with exclusive access to a memory region (blue in Figure 2), a shared region (orange), and some CPU cores. TD0 then transfers control to TD1 via Tyche's API on these cores. Once booted, TD1 running a Linux kernel uses Tyche SDK to create TD2 for the LLM runtime in userspace, partition its exclusive memory to transfer a region (dark green) to TD2, effectively creating an enclave isolated from both the CSP and user CVM. The model owner attests TD2 runs in exclusive memory before provisioning the encrypted model. Communication between components relies on shared memory: TD0 and TD1 share a region for VIRTIO [66] network access (orange), while TD1 and TD2 share another (yellow) to pass the encrypted model, user prompts, and responses.

Tyche's remote attestation is used by all parties: (1) The CSP attests it retains resource control and manages interrupts (red arrows). (2) The user attests that TD1 is isolated from TD0 and encapsulates TD2, *i.e.,* that TD2 only communicates with the CVM; (3) The model owner ensures TD2 runs the LLM in exclusive memory and does not leak the model. (4) Both the user and model owner attest that interrupts do not leak information and that Tyche zeroes memory before releasing it back to the CSP.

The remainder of the paper describes Tyche's API and capabilities (§4), their implementation and enforcement on hardware by platform-specific backends (§5), and the Tyche SDK (§6). We then evaluate Tyche (§7) and discuss related work (§8) in security monitors and micro-kernels from which we drew inspiration for Tyche's design.

4 Tyche Design: API & Capabilities

Tyche exposes an API (Table 1) based on software capabilities, which ensure portability across platforms by decoupling policies from hardware access control mechanisms [67]. Capabilities are unforgeable tokens issued by Tyche and used as arguments and return values in API calls.

Tyche capabilities manage two types of objects: memory regions and trust domains (TDs). Capabilities are owned by a TD and mediate access to the associated object. Memory capabilities grant access to physical memory ranges. TD capabilities allow the management, configuration, attestation, and execution of TDs. CPU cores and interrupts are managed as part of TD capabilities and devices are modeled as TDs.

4.1 Capability Derivation Trees

Tyche maintains two separate capability derivation trees (CDTs): one for memory regions and one for TDs. CDTs are a well-established mechanism for implementing capabilities, used in several micro-kernels [63, 90], from which we draw inspiration (see §8).

In CDTs, new capabilities are always derived from an existing one and can only inherit equal or reduced access to the associated resources. Derived capabilities appear as children of the operand node in the tree. CDTs serve three purposes: (1) they ensure the monotonicity of operations, (2) they provide a log of operations, (3) they enable the cascading revocation of an entire subtree by revoking its root node.

Call	Description
CREATE	Create a trust domain
SET/GET	Set/Get a trust domain's register or policy
SEND	Transfer a capability ownership to a TD
SEAL	Seal a trust domain
ATTEST	Attest a trust domain
ENUMERATE	Discover info. about owned capabilities
SWITCH	Control transfer into a trust 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 a TD or channel capability

Table 1. Tyche's API

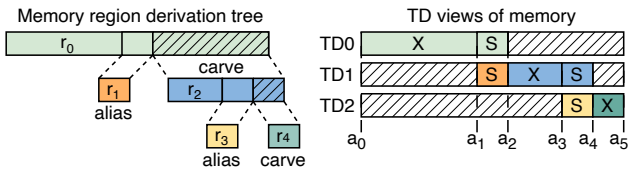

Figure 3. Memory region derivation tree and TD memory views based on Figure 2. New regions are created by carving and aliasing an existing region. Sends between TDs are omitted. (S)hared and e(X)clusive memory is reported on views.

Tyche leverages the CDT structure to further track and enforce security invariants. The region capability CDT tracks exclusive and shared access to physical memory, while the TD CDT encodes the management hierarchy between TDs and the routing of interrupts. Both are used to implement efficient revocation.

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 TD0, the first domain to run on the machine.

A TD can create new region capabilities using Tyche's *alias* and *carve* API calls (Table 1). An 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 Figure 3, hatched area). If the parent region was exclusive, the carved region remains exclusive; otherwise, it is marked as aliased. Carving enables confidential memory by ensuring that an exclusive region stems from an unbroken chain of carves, preventing any operation outside its subtree from altering its exclusivity. Tyche thus ensures that a capability's access rights and exclusivity are determined locally, based on its initial range, exclusivity status, and direct children. In Figure 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.

Tyche's *send* and *revoke* API calls allow TDs to manage memory between them. Send transfers ownership of a region, effectively allocating memory to the receiving TD. *Revoke* operates on a parent region to undo a child alias or carve, allowing the parent's owner to reclaim memory that was sent to another domain via a child region. Revocation has a cascading effect, *e.g.,* revoking $r2$ from $r0$ will also revoke $r3$ and $r4$ and re-enable access from $a2$ to $a5$ in $r0$. To provide security guarantees to the receiver, Tyche allows the

```
td1 = domain {r1, r2, td2}
      |registers.HASH: 8988ef57...
      |cores: 0b11
      |mon.api: 0b11111111111 | RECEIVE
      |interrupts: {
      | 0 -> {Report, registers: 0b0},
      | ...          }
td2 = 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 TD1 with Figure 3's nomenclature. Allowed monitor API calls are encoded as bitmaps based on the order from Table 1 (*e.g.,* bit 0 is create).

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 can only be set through a send if the receiver is unsealed (§4.3), are tied to ownership rather than the CDT, and thus non-monotonic.

A TD inspects its owned memory regions through *enumerate* or *attest*. As shown in Figure 4, for r_2, *enumerate* and *attest* report an owned region's status (exclusive or aliased), initial range, access rights, attributes, and direct children, whether owned or not. This enables (1) identifying accessible memory, (2) deriving local invariants, and (3) visibility into memory delegation.

4.3 Trust Domain Capabilities

Tyche initializes execution with the first TD, TD0, which owns the root memory region, executes on all cores, and can handle any interrupt. All other TDs are created from a subset of TD0's original resources.

TD creation and configuration: A TD creates new domains using the *create* operation, which returns a trust domain capability – a reference to the newly created TD – that appears as a child of the caller in the TD CDT. Initially, the child TD is *unsealed*, meaning it cannot yet be executed.

Through the child capability, the parent configures the TD using *set*. Specifically, it sets per core register state and *policies*. Policies define the cores the child can run on, the monitor calls it can perform, 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 that grant more rights than the parent has. Memory

allocation is performed as described previously and shown in Figure 5, the parent aliases and carves its own regions and sends the capabilities to the child. Finally, *seal* makes the TD executable and prevents modifications to registers and policies. A revoke on a TD triggers the cascading revocation of its owned capabilities and its entire subtree in the CDT.

TD execution & interrupts: Control transfers between TDs *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 TD capability; a switch with no TD argument returns to the parent.

Interrupts trigger TD control transfers. They propagate through the CDT, which encodes routing and handling policies. For each interrupt vector, TD'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 TD directly. When an interrupt not marked as *Deliver* occurs, Tyche preempts the TD and walks the CDT upwards to transfer control to the first TD (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 TDs with a *Report* policy in a manner similar to the original handler TD. These TDs 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. This routing and return protocol balances the delegation of interrupts with the ability to observe them and mitigate interrupt-based side channels. It also provides attestable guarantees on scheduling as it defines how TDs can be preempted.

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

Devices: Tyche models devices as TDs, with configuration space, DMA, and port I/O access mediated through region

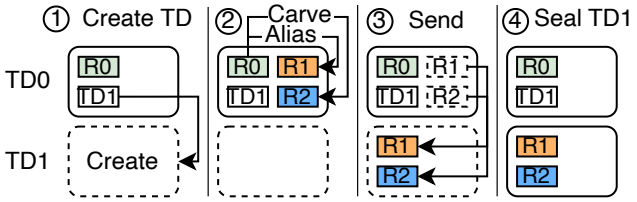

Figure 5. Capability operations to create and configure TD1.

capabilities. Interrupt routing follows the same protocol as above, but Tyche allows backends to optimize core-routing using platform specific hardware mechanisms. TDs on the CPU interact with devices through shared regions for the device's configuration space and MMIO ones. Tyche provides TD0 with a channel to every device. TD0 delegates device access to a TD by carving the configuration space and duplicating a channel, sending both to the TD to enable direct TD to device interactions.

4.4 Attestation & Security policies

Tyche's *attest* API call enables a TD to request an attestation for itself or any TD for which it owns a TD (including channels) capability. The attestation reports the TD's configuration, including a hash of its initial register content, policies, and a description of its owned capabilities. Conceptually, the attestation provides for each owned capability visibility into its direct children in their respective CDT, but not further.

Figure 4 provides the attestation of TD1, hashed and signed by Tyche. The attestation reports the configuration of TD1 and its child TD2, as well as their owned capabilities. Sets of cores, monitor calls, and registers that can be queried when handling an interrupt are encoded as bitmaps. Tyche explicitly represents the lineage between regions through naming; for example, it shows that r_3, owned by TD2, is derived from r_2. If TD2 possessed a capability derived from one not owned by TD1, the region would only appear in TD2's set of owned capabilities with a fresh name but no further information. Here, we reuse indices from other figures for clarity, but Tyche normally assigns fresh names starting from 0 based on the requesting TD's capabilities to avoid revealing information about the broader system.

For remote attestation, a TD calls *attest* to obtain its own attestation and supplies as arguments the remote verifier public key and nonce, as well as a public key generated by the domain itself, ensuring they are all measured and signed by Tyche. From this attestation, a TD or remote verifier can check whether a TD is a confidential environment (*e.g.,* an enclave or CVM), an encapsulated compartment (*e.g.,* a sandbox), or both, and uses the domain's public key to establish a shared secret and bootstrap a secure communication channel [34]. Confidentiality is ensured by the ownership of exclusive memory ranges and the absence of leakage through interrupts or revocation, *i.e.,* attributes on memory capabilities. Compartmentalization and full encapsulation

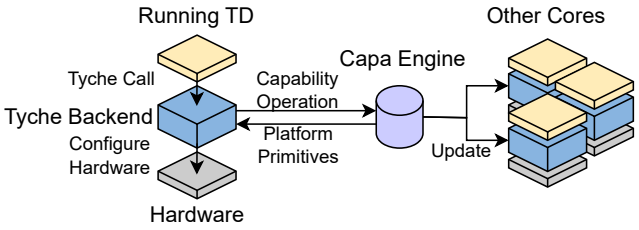

Figure 6. The capability engine maintains the system state across all cores.

is achieved by ensuring a child TD's regions are a subset of the parent's exclusive regions, and that its policies prevent it from receiving or sending other capabilities after it is sealed, as shown in Figure 4 for TD2.

5 Tyche Implementation

Figure 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 TDs 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

On all platforms, Tyche boots via a custom bootloader that enumerates CPU cores, devices, and DRAM to generate a platform description called the *boot-info*. It loads the monitor – comprising the capability engine and backend – into a reserved memory region reported in the boot-info and maps the boot-info into the monitor's address space. The bootloader then generates a fresh attestation key pair, measures the monitor, boot-info, and public key, and extends a TPM [95] Platform Configuration Register (PCR) with the result before transferring control to the monitor and passing it the private key. The monitor's backend, using the boot-info, initializes the capability engine, creates the initial TD0 (*e.g.,* a stock Linux kernel) with access to the memory not used by the monitor, including local APIC and device configuration spaces. It then creates an I/O TD for each DMA-capable device with an alias to TD0's memory. Finally, the monitor transfers control on all cores to TD0.

Like prior work [76], a TD's attestation includes a TPM-generated quote and a domain attestation issued and signed by Tyche (§4). The quote records the boot chain as PCR values, binding a specific monitor binary to an attestation public key and a machine, allowing to derive trust in the monitor and transitively into the signed TD's attestation. The platform attestation covers the full boot chain including our bootloader. We prototyped support for a dynamic root of trust (DRTM) [16, 55] using Intel TxT [55], to further reduce the TCB to only the capability engine and the backend.

5.2 Capability engine

The capability engine implements the region and TD 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` [87] 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-TD platform state (*e.g.,* per-core registers and access control mechanism configuration such as EPTs), map and unmap physical memory ranges in a TD'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 TD executes on each core and ensures a consistent state view across cores. The engine uses the CDTs to determine which TDs 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 Figure 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, TD revocation, and interrupt delivery.

For example, sending a carve is processed in the engine by a core which first validates the request and computes the resulting capability state. It then enqueues an access right update on all cores running either the sender or the receiver and

uses the platform IPI to preempt them so they can process the update. The update's logic includes two synchronization barriers: the first to ensure all affected cores have been preempted, and the second to wait until the new platform state is available. Between the two barriers, the initiating core running in the engine uses the platform primitives to update the platform state, *i.e.,* it unmaps the region from the sender, maps it into the receiver (*e.g.,* via EPTs on x86_64), before finalizing the capability state, making it globally visible. It then unblocks cores waiting on the second barrier, allowing them access to the new state which they apply onto their local core (*e.g.,* TLB shootdown).

5.3 Tyche x86_64 backend in root mode

Intel VT-x [97] is a common hardware extension to accelerate virtualization in which a host in `root` mode runs guest VMs in `non-root` mode. A virtual machine control structure (VMCS) controls guest execution on a core, safely virtualizing guest privileged operations such as writes to *cr3*, without host intervention, while extended pages tables (EPTs) [11, 97] restrict non-root access to memory. The *vmcall* instruction enables direct traps from non-root to root mode.

Tyche x86_64 backend repurposes Intel VT-x to isolate TDs. The backend consists of 6K lines of Rust code and uses the unsafe keyword to configure hardware. Most of the code is due to enumerations for error codes, hardware bitmap values, and VMCS field identifiers. Tyche on x86_64 is only 230KB of compiled code.

The Tyche x86_64 backend executes in root mode and runs all TDs in non-root mode, with one EPT per TD to restrict memory access to its owned region capabilities, and one VMCS per TD allowed core. Devices are abstracted as TDs as well, with memory access enforced via the Intel I/O-MMU [59]. Internally, the backend uses `INIT IPI`s to make other cores trap into the monitor and implements cross-core synchronization with barriers based on semaphores, atomics, and spinlocks that poll for updates on each locking attempt.

TDs interact with the monitor through non-interposable *vmcall* instruction. API arguments are passed via general-purpose registers, and capabilities referenced via TD-local indices, similar to UNIX file descriptors. Return values follow the same convention, and large values (*e.g.,* plaintext attestation) follow a protocol with multiple *vmcall* to consume the entire output. Register-based communication is preferred over direct memory access by the monitor to reduce security risks such as confused deputy [48] and cache attacks. The monitor accesses memory regions only when they are not accessible by any TD, either to enforce a `Clean` by zeroing its content or a `Hash` by reading it. Future work aims to eliminate even these accesses by offloading them to more restricted execution environments [31], ensuring full TD memory isolation from the monitor.

TD switches save the caller's VMCS and load the callee's. The backend manages general-purpose registers in software and handles other registers – including model-specific registers (MSRs) – using a combination of hardware and software, preventing leakage across domains. Virtual processor identifiers tag EPT TLB entries to reduce flushing during TD transitions. All control transfers, whether synchronous (`switch` monitor calls) or asynchronous (e.g., exceptions), pass through the monitor, allowing Tyche to track state on each core and enforce TD policies. After a switch, the child runs until it either receives an unhandled interrupt or explicitly returns. The x86_64 backend also supports a prototype switch quantum, enabling preemption of a child after a fixed number of cycles enforced directly in hardware by Intel VT-x [97], with deadlines tracked across TDs in the monitor.

TD interrupt policies are offloaded to hardware via VMCS configuration to enable direct delivery where possible. Intel VT-x supports fine-grained trapping for the first 32 vectors (exceptions) but only a single bit for external (higher-vector) interrupts. To selectively deliver high-vector interrupts when not all of them are allowed, the backend traps all external interrupts, inspects the vector, and either reinjects allowed ones or switches to an ancestor TD per the routing described in §4. Alternatively, the x86_64 backend supports hardware APIC virtualization and I/O-MMU interrupt remapping when available, reducing traps and enabling finer-grained control. It can also delegate APIC virtualization configuration by exposing relevant VMCS fields [97] as TD registers to the parent. This allows the parent to manage interrupt delivery but limits the child's 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) [85] 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 a TD configuration cannot be satisfied with the available PMP registers, a synthetic exception is injected and delegated to a parent TD 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 [4, 40] to defend against rogue DMA requests. Like x86_64, interrupts and exceptions can be either trapped or delivered directly to a TD 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 a TD 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 TDs must carefully manage memory to maximize the use of contiguous memory, like existing RISC-V security monitors [21, 31, 68,

108]. 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 TDs run unmodified workloads as sandboxes, enclaves, or CVMs in separate TDs. 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 one kernel driver (**Tyche-Capa**) to allow Linux environments running in a TD to interact with the monitor and create TDs. At the same time, code that does not use Tyche runs unmodified within the Linux TD, ensuring backward compatibility with existing software.

The Tyche-Capa kernel driver works across both x86_64 and RISC-V and interacts directly with Tyche. It abstracts capabilities and exposes a simplified interface through *ioctl* commands, file descriptors, and `mmap` for creating, managing, and running TDs. The driver allocates memory from kernel pools, marks pages as reserved (like ballooning [101]), and ensures revocation to prevent resource leakage, even in the event of user program crashes. To reduce fragmentation and accommodate RISC-V's limited PMP entries, the driver can optionally reserve a contiguous physical memory range at boot via kernel command-line parameters.

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 TDs.

VMs & CVMs with KVM on Intel x86_64: KVM-Tyche is a fork of the KVM-Intel [30] driver, modified to run VMs and CVMs as child TDs (*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 [2, 12, 25, 46].

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 TD. 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 [2] VMM enables confidential VM support by requesting exclusive memory from Tyche-Capa.

Sandboxes & enclaves on x86 with Gramine: Gramine [7], formerly Graphene [96], is an open-source library OS (libOS) for running unmodified applications inside Intel SGX [58] enclaves; it shields the application and mediates system calls to the untrusted OS. **Gramine-Tyche** is a fork of the Intel SGX platform abstraction layer of Gramine that runs enclaves and sandboxes as TDs in userspace, isolating programs in exclusive or shared memory without Intel SGX hardware. The porting effort involved modifying only 300 lines of code (LOC) out of 14.9k LOC, replacing Intel SGX logic with Tyche-Capa calls and populating page tables.

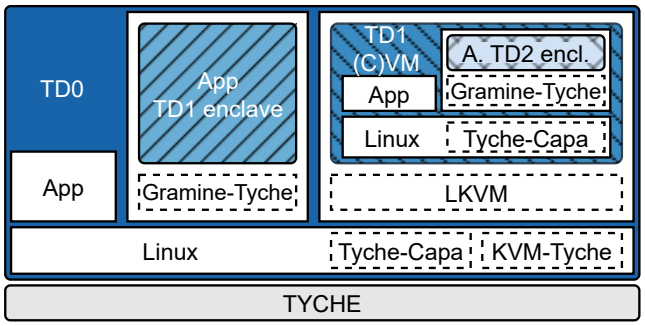

Figure 7. Superposed system and TD views of deployments in Figure 11. TDs are blue rounded boxes. System abstractions (processes, kernels, and VMs) are full rectangles, and libraries and drivers dotted ones.

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 TDs.

6.3 Preserving runtime behavior

Backward compatibility is a key aspect of Tyche's design. The Tyche SDK preserves the semantics and runtime behavior of existing abstractions, enabling fair performance comparisons in §7. Future work could optimize the software stack to better leverage Tyche's capabilities. For example, Gramine's libOS could run as privileged software to reduce enclave exits, as in its recent TDX port [57, 65]. An Tyche-aware VMM could use TDs for efficient device passthrough or trusted zero-copy I/O, and optimize interrupt handling via safe delegation to reduce VM exits. For instance, timer interrupts currently route to the hypervisor (`TD0` Linux), but modifying KVM's scheduling to use TD interrupt policies could eliminate this overhead.

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 unmodified real-world applications isolated with various threat models (§7.2), and details the confidential LLaMa

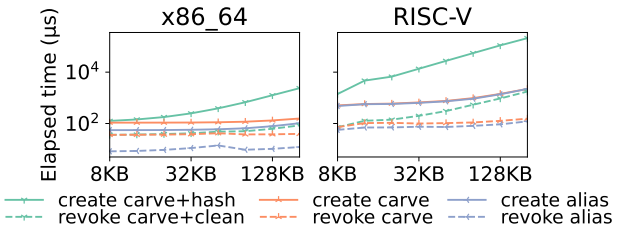

Figure 8. Create & revoke latencies as a function of size for carve, with and without hash & clean, and alias TD memory.

CPU-based inference for mutually distrustful model and data owners (§7.3).

Experimental setup: On x86_64, Tyche runs on an Intel i7-10700 CPU with 16 cores and TD0 is a stock Linux Kernel v6.2. The CPU supports Intel SGX [58] v1 with a usable enclave page cache (EPC) of 94MB. On RISC-V, Tyche runs on a StarFive VisionFive2 board [92] with the JH7110 SoC and 8 PMP entries per hart and TD0 is a stock Linux Kernel v5.15. VMs and CVMs run a stock Linux Kernel v6.2 with 4GiB of RAM, 8 cores, and VIRTIO [66] devices. CVMs further enable SWIOTLB [9] bounce buffers from the kernel command line to copy I/O into a memory region shared with TD0. For network experiments, a client [1, 44] runs on a separate machine connected to the local Ethernet network using 12 cores and 400 connections. Results report the average of the best 9 out of 10 runs, except for CoreMark-Pro which averages internally [38].

Naming conventions: Bare-metal Linux is referred to as "native", while KVM-Intel Linux VMs on bare-metal Linux are "native VMs". An enclave deployed with Gramine-SGX on bare-metal Linux is an "SGX enclave". TD0 is the first trust domain created by Tyche at boot and runs a Linux OS. A confidential TD created by TD0 with Gramine-Tyche is a "TD1 enclave". VMs and CVMs created by TD0 with KVM-Tyche are called "TD1 VM" and "TD1 CVM", respectively. A "TD2 enclave" is a Gramine-Tyche enclave created by a TD1 CVM, protected from both TD0 and the TD1 CVM. Finally, sandbox TDs running with Gramine-Tyche are not reported as they achieve the same performance as enclave TDs. Figure 7 illustrates these deployments.

7.1 Microbenchmarks

We report the average latencies (over 1000 operations) of Tyche operations on both platforms, allowing us to compare hardware mechanisms (§7.1.1). We then measure Tyche's CPU and I/O overheads (§7.1.2) on x86_64 for different TD configurations (Figure 7) compared to native, native VM, and SGX enclaves when applicable.

7.1.1 Tyche operations on x86_64 & RISC-V

Creation/Revocation: Figure 8 shows carves are more expensive than aliases as they remove memory from TD0 and notify other cores via IPIs (§5.2). This difference is especially noticeable on x86_64, where carves trigger a walk on TD0's EPT. On both platforms, *hash* and *clean* increase latencies with the TD's size, as they require reading and writing memory, respectively.

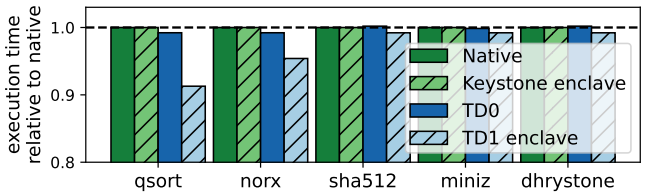

Figure 9. RV8 RISC-V CPU microbenchmarks comparison between native, unmodified Keystone, and Tyche.

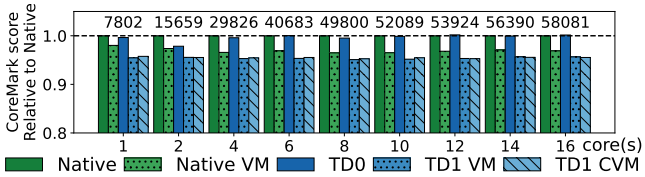

Figure 10. CoreMark-Pro performance relative to bare-metal Linux (Native) for different TD deployments and varying number of cores on x86_64 (raw native score at the top).

Switches: In Table 2, TD 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 TD 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 [13, 57, 58, 68, 96].

RISC-V prototype: On Figure 9, Tyche on RISC-V competes with native and unmodified Keystone on most RV8 microbenchmarks, with a <10% slowdown for TD1 enclaves on short-lived programs (qsort, norx) due to the extra indirection to the Tyche-Capa driver.

7.1.2 Tyche CPU & I/O overheads on x86_64

CPU overheads: Figure 10 shows CoreMark-PRO CPU benchmark results varying core counts from 1 to 16. Linux TD0 matches bare-metal performance, with slight bumps (at most ~2%) at 2, 4, and 8 cores due to virtualization amplifying cache effects from hyperthread placement. Some benchmarks benefit from shared caches, while others experience pressure on the shared TLB. TD1 VM and CVM have

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

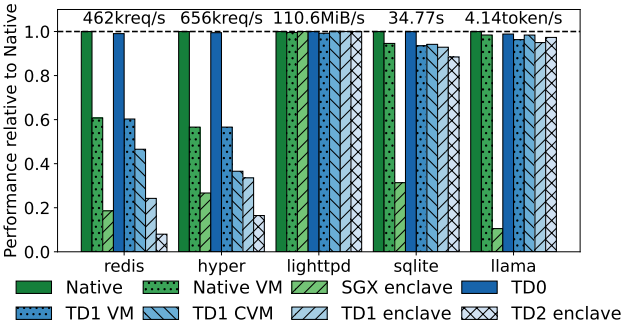

Figure 11. Performance relative to native execution for real-world applications isolated as TDs on x86_64 (blue). Bare-metal Linux deployments are included for comparison (green).

a slight overhead: ~1% compared to a native VM and ~4% compared to native, due to transitions from TD1 to TD0 on timer interrupts. Notably, CVMs incur no additional cost compared to TD1 VMs.

I/O overheads: Figure 11 and Figure 12 show Redis [3] throughput and latency under different TD configurations, measured by a memtier [1] remote client. TD0 and TD1 VMs have <1% throughput overhead compared to native and native VM, while TD1 CVM degrade latency and drop throughput by ~20% compared to native VMs, due to I/O bounce buffer copies. Enclaves perform poorly compared to VMs due to the libOS copies to and from the enclave. The TD1 enclave outperforms SGX, while TD2 degrades throughput and latency further due to accumulated copies and indirections – VIRTIO drivers, SWIOTLB bounce buffers to the TD1 CVM, and Gramine-Tyche copies to the TD2 enclave. We confirm this by measuring lighttpd [64] throughput and observe that as the payload size increases, the cost of copies is amortized, and all configurations reach native and TD0 throughput for a 10K bytes payload.

7.2 Enclaves, VMs, CVMs, & composable isolation

Figure 11 reports Tyche's performance isolating unmodified real-world applications on x86_64 relative to native and includes native VMs and SGX enclaves for comparison.

Applications: We use two web servers (lighttpd [64] and rust-hyper [54]), a database (SQLite [49]), an in-memory key-value store (Redis [3]), and a CPU-based machine learning inference program (llama-cpp [43]) processing prompts. We use wrk [44] to measure HTTP servers and memtier for Redis. SQLite runs the full in-source speedtest1 [49] with a size argument of 1000, measuring various SQL operations including 500K inserts. LLaMa reports its own throughput [43] after generating 1000 tokens. Applications overlap with or resemble those found in related work [13, 96], allowing performance overhead comparisons across platforms.

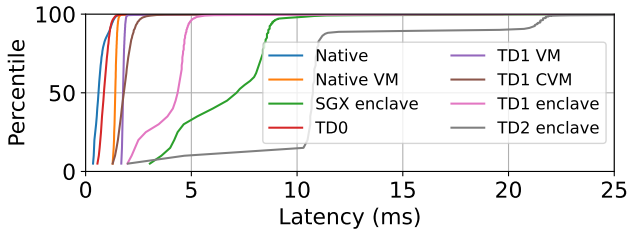

Figure 12. Redis GET latency distribution as measured by memtier (max throughput) during Figure 11's experiment.

Configuration: All binaries are unmodified and come from Gramine's default examples, except for LLaMa, which we added (§7.3). The Gramine manifests disable debugging as well as exitless [6] due to CVE-2022-21233 and CVE-2022-21166. All enclaves are multi-threaded, with 3 Gramine runtime threads [8], and their sizes range from 256MB to 4GB.

TD0: TD0 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 TD0 has direct access to devices and the APIC.

VMs & CVMs: TD1 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), TD1 CVMs experience additional overheads due to extra copies through bounce buffers for I/O (see §7.1.2). They otherwise perform similarly to native VMs.

Enclaves: TD1 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 a TD2 enclave. This is likely due to the limited EPC (94MB) size compared to the high memory usage (4GiB) for the model [52, 77] and llama-cpp contexts [43].

Discussion: The latency of network operations is the primary issue but not a direct consequence of Tyche's design, as discussed in §7.1.2, and could be alleviated [40, 69, 70, 83], *e.g.,* by providing safe device passthrough. Tyche otherwise demonstrates that existing frameworks can be ported to run TDs, achieving performance comparable to equivalent native deployments and outperforming Intel SGX, even in nested cases, reinforcing our claim of backward compatibility.

7.3 Confidential LLM inference with mutual distrust

This case study deploys the scenario from Figure 2 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 TD0 hypervisor with KVM-Tyche. The user runs a TD1 CVM and creates a TD2 encapsulated enclave for LLM inference on the CPU, running unmodified llama-cpp [43] with Gramine-Tyche and 4 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, sharing memory only with the user-controlled TD1 CVM. The CVM receives prompts through an SSH connection and forwards them to the LLM enclave. User keys can further be secured in a separate enclave.

As shown in Figure 11, the TD2 enclave achieves near-native performance (~2% overhead), outperforms SGX enclaves (~10x), and goes beyond prior industrial solutions [18, 37] by also protecting the model from the user. Future KVM-Tyche integration with Tyche could explore enabling secure GPU passthrough.

8 Related Work

Tyche combines a security monitor design with a capability-based API inspired by micro-kernels to deliver a unified isolation solution.

Security monitors: They serve as trusted intermediaries [88] to enhance systems with new isolation and security guarantees [28, 41, 106] and are easier to inspect, update and extend than hardware-based solutions. Arm CCA uses a monitor to partition memory between realm, normal, and secure worlds [72] and Komodo [41] shows how a division of labor between software and hardware [23] enables enclaves in Arm TrustZone [19]. Security monitors are popular in confidential computing to restrict privileged software, *e.g.,* Inktag [51], Overshadow [28], HyperEnclave [61], and Keystone [68] protect enclaves from untrusted OSes, Cloudvisor [106] isolates VMs from hypervisors, and Blackbox [50] secures containers. They can offer intra-VM isolation, *e.g.,* Veil [13] implements enclaves in AMD CVMs [15] while Erebor [105] focuses on sandboxes in Intel TDX [57] ones, or they are used to harden kernel integrity [20, 32, 89].

Tyche takes a distinct approach by providing a unified low-level isolation abstraction, *i.e.,* trust domains, as a common foundation for higher-level ones. Unlike traditional monitors that extend or support well-known system abstractions, such as enclaves or CVMs, Tyche trust domains are orthogonal to system abstractions, their semantics, or existing privilege levels, allowing them to not only replace previous monitors but also to compose their isolation boundaries. This approach is inspired by micro-kernel design.

Micro-kernels: Micro-kernels [63, 73, 94] 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 outside the kernel* [73] and supporting custom isolation abstractions

Tyche leverages key micro-kernel concepts to deliver modern isolation guarantees. Its memory operations resemble L4 [73]'s *grant*, *map*, and *flush*, which are foundational to memory management. 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 [42], which enables recursive VM isolation through "nested processes" without the overhead of naive nested virtualization. While Fluke focuses on modular VM isolation, Tyche generalizes this approach by providing a foundational mechanism to compose compartmentalization and confidential computing isolation.

Despite its similarities to micro-kernel APIs, Tyche is *not* a kernel replacement, does not create directly useable system abstractions such as processes, and requires an untrusted kernel in TD0 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 functions more like an exokernel [39] or a trusted state machine, enforcing and attesting the partitioning and sharing of resources across trust domains. Tyche shares some similarities with hypervisors that adopt micro-kernel principles to improve security [71, 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 [22, 99], which grant extra privileges to their initial domain (e.g., DOM0 in Xen [22] or the root partition in Hyper-V [99]), TD0 in Tyche has no special privileges. In fact, Tyche could enable designs with multiple "root" TDs in the capability derivation tree, by partitioning hardware at boot time, *e.g.,* to create a secure coprocessor running a TD for security-sensitive tasks like key management [107]. As future work, we explore safe bare-metal interrupt access inspired by Directvisor [29].

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 [14, 62, 86, 102]. Tyche could use alternative architectures [62, 86], like NoHype [62], that eliminate virtualization and provide simpler primitives to partition I/O, memory, and cores. CHERI [102] capabilities offer a promising alternative to page-based mechanisms, already supporting enclaves [98] and secure embedded devices [14]. 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 [72], or realm world [72]. Alternatively, it could run as firmware in EL3, akin to the RISC-V backend, and leverage Granule Protection Table [72] to isolate memory. Similarly, Tyche could run as an Intel TDX [57] module, providing attestable policies for resource management and private shared memory.

9 Conclusion

Tyche's trust domain abstraction unifies isolation enforcement across privilege layers and system abstractions without accumulating security extensions. We showed that the Tyche security monitor is both general, by supporting and composing enclaves, sandboxes, and confidential virtual machines, and practical by running unmodified applications with near-native performance. A case study of confidential inference in the cloud involving mutually distrustful CSPs, users and model owners illustrates its applicability to modern, complex threat models in the cloud.

References

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

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

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

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

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

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

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

[8] Gramine: threading. https://gramine.readthedocs.io/en/stable/manifest-syntax.html?highlight=num, 2020.

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

[10] Tee device interface security protocol (tdisp). https://pcisig.com/tee-device-interface-security-protocol-tdisp, August 2022.

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] Andres Freund. backdoor in upstream xz/liblzma leading to ssh server compromise. https://www.openwall.com/lists/oss-security/2024/03/29/4, 2024.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[26] 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.

[27] 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.

[28] 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.

[29] Kevin Cheng, Spoorti Doddamani, Tzi cker Chiueh, Yongheng Li, and Kartik Gopalan. Directvisor: virtualization for bare-metal cloud. In *Proceedings of the 16th International Conference on Virtual Execution Environments (VEE)*, pages 45–58, 2020.

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

[31] 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.

[32] 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.

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

[34] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976.

[35] 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.

[36] Jules Drean, Fisher Jepsen, Edward Suh, Srini Devadas, Aamer Jaleel, and Gururaj Saileshwar. Teaching an Old Dog New Tricks: Verifiable FHE Using Commodity Hardware. *CoRR*, abs/2412.03550, 2024.

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

[38] EEMBC. *CoreMark PRO*, July 2019. v1.1.2743 https://www.eembc.org/coremark-pro.

[39] 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.

[40] 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 *ASPLOS (2)*, pages 1061–1076, 2024.

[41] 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.

[42] 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.

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

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

[45] Google. Google confidential space security overview. https://cloud.google.com/docs/security/confidential-space, June 2024.

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

[47] gVisor Team. gVisor ptrace Platform. https://gvisor.dev/docs/architecture_guide/platforms/#ptrace, 2024.

[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] JFrog. Examining Malicious Hugging Face ML Models with Silent Backdoor. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/.

[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] 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.

[71] 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.

[72] 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.

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

[74] Linux Security. Understanding and Mitigating CVE-2024-42070: nftables Type Confusion Vulnerability. https://linuxsecurity.com/news/security-vulnerabilities/understanding-and-mitigating-cve-2024-42070-nftables-vuln, 2024. [Online; accessed 4-April-2025].

[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. 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.

[79] Microsoft. Virtual secure mode & virtual trust level (vtl). https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/tlfs/vsm, 2025.

[80] National Vulnerability Database (NVD). CVE-2024-1086: Linux Kernel Netfilter Use-After-Free Vulnerability. https://nvd.nist.gov/vuln/detail/cve-2024-1086, 2024. [Online; accessed 4-April-2025].

[81] National Vulnerability Database (NVD). CVE-2024-46722: AMDGPU Driver Out-of-Bounds Read. https://nvd.nist.gov/vuln/detail/CVE-2024-46722, 2024. [Online; accessed 4-April-2025].

[82] Trail of Bits. Exploiting ML models with pickle file attacks: Part 1. https://blog.trailofbits.com/2024/06/11/exploiting-ml-models-with-pickle-file-attacks-part-1/, 2024.

[83] 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.

[84] Dark Reading. 'Sleepy Pickle' Exploit Subtly Poisons ML Models. https://www.darkreading.com/threat-intelligence/sleepy-pickle-exploit-subtly-poisons-ml-models.

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

[86] 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.

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

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

[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] 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.

[91] Splunk. Paws in the Pickle Jar: Risk & Vulnerability in the Model-sharing Ecosystem. https://www.splunk.com/en_us/blog/security/paws-in-the-pickle-jar-risk-vulnerability-in-the-model-sharing-ecosystem.html.

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

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

[96] 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.

[97] 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.

[98] 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.

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

[100] 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.

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

[102] 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.

[103] 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.

[104] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*, 2019.

[105] 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.

[106] 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.

[107] 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.

[108] 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.