RESEARCH-ARTICLE

Erebor: A Drop-In Sandbox Solution for Private Data Processing in Untrusted Confidential Virtual Machines

CHUQI ZHANG, National University of Singapore, Singapore City, Singapore

RAHUL PRIOLKAR

YUANCHENG JIANG, National University of Singapore, Singapore City, Singapore

YUAN XIAO, ShanghaiTech University, Shanghai, China

MONA VIJ, Intel Corporation, Santa Clara, CA, United States

ZHENKAI LIANG, National University of Singapore, Singapore City, Singapore

View all

Open Access Support provided by:

National University of Singapore

Intel Corporation

ShanghaiTech University

PDF Download
3689031.3717464.pdf
08 February 2026
Total Citations: 3
Total Downloads: 2174

Published: 30 March 2025

Citation in BibTeX format

EuroSys '25: Twentieth European Conference on Computer Systems
March 30 - April 3, 2025
Rotterdam, Netherlands

Conference Sponsors:
SIGOPS

EuroSys '25: Proceedings of the Twentieth European Conference on Computer Systems (March 2025)
https://doi.org/10.1145/3689031.3717464
ISBN: 9798400711961

Erebor: A Drop-In Sandbox Solution for Private Data Processing in Untrusted Confidential Virtual Machines

Chuqi Zhang[§*], Rahul Priolkar[¶], Yuancheng Jiang[§], Yuan Xiao[△◇]
Mona Vij[†], Zhenkai Liang[§], Adil Ahmad[¶]
National University of Singapore[§], Arizona State University[¶], ShanghaiTech University[△], Intel Labs[†]

Abstract

Confidential virtual machines (CVMs) are designed to protect data in cloud machines, but they fail in this task in common Software-as-a-Service (SaaS) cloud environments. In such settings, the software stack within a CVM, including service programs and the operating system, that receives and processes data may intentionally disclose it to attackers. We present Erebor, a sandboxing architecture for CVMs that processes client data in secure containers, where restrictions apply to both (a) access by all untrusted outside components and (b) the sandbox's ability to communicate data through memory and software-controlled direct or covert exits. Erebor enables such restrictions through a security monitor design based on intra-kernel privilege isolation for CVM, fully compatible with emerging cloud deployments without requiring host modifications. Under realistic scenarios, such as large language model inference and private information retrieval, Erebor only adds a performance overhead of 4.5%–13.2%, demonstrating its practicality in terms of enabling strong data sandboxing in modern cloud machines.

CCS Concepts: • **Security and privacy** → **Virtualization and security**; **Trusted computing**; • **Software and its engineering** → **Operating systems**.

Keywords: CVM, Sandboxed Container, OS Design

1 Introduction

Hardware trusted execution environment (TEE) extensions, like Intel TDX [18], AMD SEV [61], and ARM CCA [5], allow users to protect their sensitive workload and data in confidential virtual machines (CVMs). These virtual machines are isolated from an untrusted host platform (e.g., a cloud machine on a remote server), and thus are designed to protect data even if the platform is compromised (§2).

Unfortunately, CVMs cannot protect user data in the common cloud *software-as-a-service* (SaaS) model. In this model, a third-party *service provider* deploys a service program to handle data processing requests from different clients (§3.1). Common examples of such services include private information retrieval [10] and cloud-based intrusion detection [9]. Even when deployed within a CVM, client data remains fully accessible to the CVM software controlled by the service provider, including the service program and the CVM OS kernel. Such software may, individually or in collusion with each other (and even with the untrusted host hypervisor), leak client data to providers (§3.2).

Protecting client data from the untrusted CVMs software requires enforcing additional data isolation within the virtual machine, and recent approaches [43, 89] have proposed techniques to achieve this. These systems rely on *CVM partitioning features*—specifically AMD Virtual Machine Privilege Levels (VMPL [44]). Using such features, they instantiate a security monitor in a privileged CVM partition and use it to create enclaves akin to SGX [71] in isolated partitions.

While promising for important cloud computing models, aforementioned systems only provide partial data protection in our model, and raise deployment hurdles (§3.3). In particular, enclaves are designed for one-way isolation from the OS to programs. Programs can directly or covertly disclose data through system calls or hypervisor calls. Additionally, leveraging CVM partitioning features for isolation requires non-trivial changes to the cloud provider's hypervisor and, in emerging deployments [6, 53], to their paravisor [4, 28, 97]. This raises deployment challenges from the perspective of a cloud tenant (i.e., the service provider in our model).

This paper presents Erebor, a new system architecture for confidential virtual machines that shields client data in the remote SaaS model. In contrast to prior systems, Erebor enforces a full *sandbox* where, in addition to typical enclave protections, intentional data disclosure by the service provider's program is prohibited. Erebor is also entirely *drop-in* since it does not require changes to the cloud provider's software infrastructure. We developed a Proof-of-Concept (PoC) for Intel TDX, and our evaluation indicates that Erebor introduces a modest performance overhead of 4.5%–13.2% compared to native CVM execution, demonstrating its practicality for strong client data protection in cloud servers.

The key to Erebor's drop-in status is a new CVM security monitor design based on the notion of *intra-kernel privilege isolation* [45, 49–51]. This design virtualizes the hardware kernel privilege into two modes using only guest-controlled

[*] Work done while being a visiting researcher at ASU with Dr. Adil Ahmad.
[◇] Work done while working at Intel Labs.

EuroSys '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717464

features that do not require hypervisor or paravisor changes (unlike VMPL). The CVM kernel runs in a lower privileged mode, managing most system functions while delegating only a small set of sensitive operations to the monitor (§5). Unlike prior privilege isolation solutions, Erebor's monitor is specifically designed for CVMs and leverages advanced hardware features (e.g., Protection Keys and Control Enforcement Technology) to ensure performance and security.

Leveraging its privileged security monitor, Erebor implements sandboxed containers, dubbed Erebor-Sandbox, where a service provider's program executes and provides a service on data from a single client. Erebor enforces the following three properties for each sandbox.

I. Resource-efficient memory isolation. To prevent the sandbox from leaking client data or outside software from accessing this data through memory interfaces, Erebor ensures that all *writable* regions of the sandbox (where data is processed) are inaccessible to outside software. Different sandbox runtimes are still allowed to share common memory regions (e.g., large databases) in a secure *read-only* manner to reduce memory consumption [41]. The monitor enforces the aforementioned isolation policies by controlling all configuration interfaces (e.g., page tables) that permit CPU and devices to access certain memory regions of a CVM (§6.1).

II. Sandbox runtime and exit protection. To prevent the sandbox from maliciously exiting to outside software and intentionally leaking data, the monitor interposes and appropriately handles all exits. Interposition is achieved by controlling the interrupt descriptor table, model-specific registers, and the guest-hypervisor communication interface. Using this interposition, the monitor protects the sandbox's context state (i.e., registers) at interrupts and exceptions, and disables system calls and hypercalls from userspace. Erebor integrates a Library OS (LibOS) to handle system call-related functionality within the sandbox boundary. This functionality includes memory management, file system, multi-threading, and task synchronization (§6.2).

III. Secure data communication. To prevent client data or processing results of this data from ever being accessible to attackers during network communication, Erebor provides end-to-end data shepherding. Specifically, the monitor safely obtains sensitive data from a remote client, installs it within a sandbox, and sends back processed results. This is achieved through an end-to-end secure communication channel between the client and the monitor, and established using CVM remote attestation-based authenticated key exchanges. The attestation interface is controlled by the monitor to avoid impersonation attacks from the untrusted OS (§6.3).

We implemented a prototype of Erebor for TDX guest that runs the Linux kernel (§7). Our prototype includes a LibOS and development toolkit for service providers to execute their programs in Erebor-Sandbox. The LibOS is based on the well-maintained Gramine [85, 87], and it supports native Linux applications with few changes. We also performed a detailed security analysis of Erebor to show that it enforces strong client data protection in CVMs (§8).

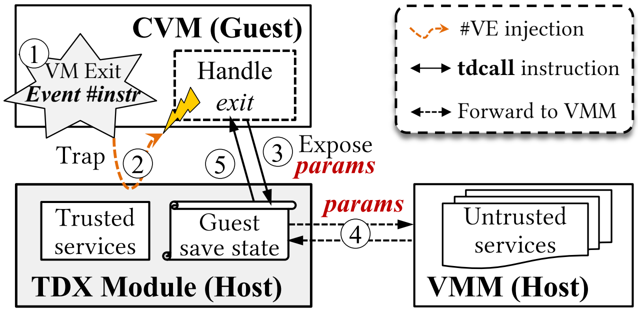


Figure 1. Illustration of TDX's Guest-Host Communication Interface (GHCI [13]) for synchronous CVM exits.

We evaluated Erebor on an Intel Xeon Platinum (Emerald Rapids) server using both microbenchmarks and real-world programs—including CPU-bound data processing services such as image/graph processing, private information retrieval, intrusion detection, and ML inference. As mentioned earlier, Erebor introduces a modest performance overhead, and reduces memory overheads by up to 89.1% compared to naive solutions, demonstrating its practicality (§9).

2 Background

2.1 Intel Trust Domain Extensions (TDX)

TDX [18] protects the integrity and confidentiality of guest CVMs. The host TDX module, a trusted Intel-signed software, operates in a protected CPU mode and works with the host hypervisor (VMM) to manage memory and context switches for CVMs. TDX introduces a privileged `tdcall` instruction, enabling the guest OS to request host TDX module or VMM operations, such as request memory for device I/O, passing data to the VMM for VM exit handling, and remote attestation. This section elaborates on these aspects.

CVM memory protection. CVM guest memory is managed by a secure Extended Page Table (sEPT) controlled exclusively by the TDX module. Guest memory is divided into *private* and *shared* parts. At runtime, private memory is protected from untrusted software and device access. Specifically, external software, including the host hypervisor and BIOS, cannot access or modify the guest's private memory. The host hypervisor can only allocate and reclaim private pages for memory management purposes. Device interactions—MMIO and DMA—are restricted to shared memory only. DMA is restricted by checks of the host IOMMU [17]. To convert its memory between private and shared, a CVM must issue a `tdcall` to the TDX module.

CVM exit state handling. A CVM may exit either synchronously or asynchronously to the host hypervisor. At all exits, the guest's context state (e.g., general-purpose regis-

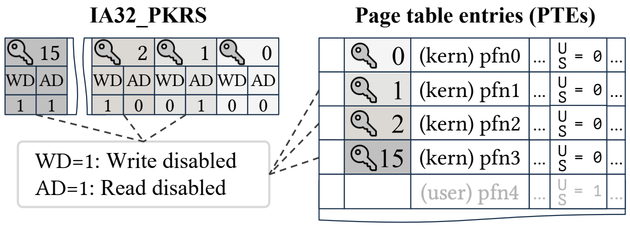

IA32_PKRS
15
WD AD
1 1
2
1
0
WD AD WD AD WD AD
1 0 0 1 0 0
WD=1: Write disabled
AD=1: Read disabled
Page table entries (PTEs)
0 (kern) pfn0 ... U/S = 0 ...
1 (kern) pfn1 ... U/S = 0 ...
2 (kern) pfn2 ... U/S = 0 ...
15 (kern) pfn3 ... U/S = 0 ...
(user) pfn4 ... U/S = 1 ...

Figure 2. Illustration of Intel PKS [24]. At PTEs, each physical page frame (pfn) is tagged with a protection key (ranging from 0 to 15). PKS only applies to kernel mode (supervisor) pages (PTEs that have the `U/S` flag set to 0).

ters) is securely saved and restored by the TDX module.

1) *Asynchronous exits* occur due to hardware events that don't require input from the CVM (e.g., external device interrupts). The TDX module traps the event, saves the guest context, and returns control to the host.

2) *Synchronous exits* are caused by guest software events, like explicit hypercalls or a few privileged instructions (e.g., `wrmsr`) that the host may emulate. In such cases, TDX defines the Guest-Host Communication Interface (GHCI [13]) for CVMs to selectively expose data to host VMM. These exits (①, Fig. 1) are trapped by the TDX module, which injects a virtualization exception (#VE) into the guest (②). The guest handles #VE, prepares the parameters, and executes a `tdcall` with the leaf instruction `vmcall` to exit to host (③–⑤). Since `tdcall` is a privileged instruction, executing it from userspace triggers a general protection fault (#GP).

Remote attestation. A guest may request the TDX module for a CPU-signed CVM attestation digest using `tdcall`. The digest contains (a) measurement of the guest's initial contents (e.g., firmware and images) and (b) custom data provided by the CVM. This digest is then sent to a remote challenger for verification. The custom data is used to establish a secure channel between the challenger and the CVM using authenticated key exchange protocols.

2.2 Control Enforcement Technology (CET)

CET [7, 80] enables hardware-assisted control flow integrity (HW-CFI) for kernel and user modes, preventing adversaries from manipulating control flow in both forward and backward directions. For forward control-flow protection, CET introduces indirect branch tracking (**IBT**) using an instruction, `endbr64`. At indirect call or jump targets, the hardware inspects that the first instruction is `endbr64`. If `endbr64` is missing, a control protection fault (#CP) is raised to the kernel. To support backward control-flow integrity, CET introduces hardware-assisted shadow stacks (**SST**), which operate on a *per-logical-core and per-task* basis. The shadow stack pointer (SSP) for the kernel mode is managed by the model-specific register `IA32_PL0_SSP`.

Kernel mode shadow stacks must reside in *write-protected* memory, with each stack possessing a unique token to ensure only one logical processor can activate it at a time. At calls, exceptions, or interrupt flows, the CPU pushes call site information onto the shadow stack. Before interrupts (that use separate kernel execution stacks) or context switches to new kernel task threads, the OS switches the SSP to the target shadow stack. Upon returns (via `ret` or `iret`), the hardware verifies the return address against the saved information in the shadow stack. If verification fails, a #CP is triggered.

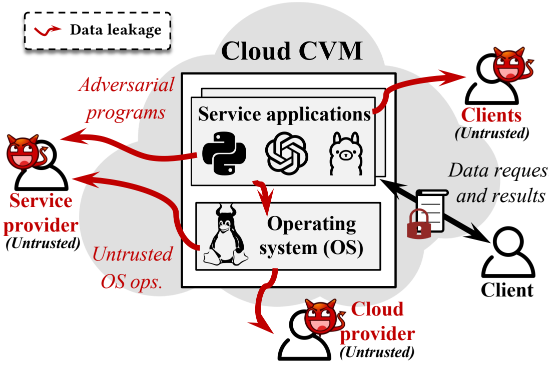


Figure 3. Erebor's system model and trust model (from a client's point of view).

2.3 Protection Keys for Supervisor (PKS)

PKS [24] allows *thread-local* efficient memory access control at a page granularity within kernel memory regions. As illustrated by Fig. 2, when PKS is activated on a system (by setting the control register `CR4.PKS` bit), each kernel virtual page is associated with a protection key (ranging from 0 to 15) specified in its page table entry (PTE). A model-specific register, `IA32_PKRS`, manages the memory access permissions for each protection key on its corresponding CPU. Depending on the set permissions, a thread's read or write access to a page will raise a fault.

3 Motivation

3.1 SaaS Data Processing in CVM

This work considers software-as-a-service (SaaS) data processing services inside CVM. This section describes the service *system model* and *application model.*

System model. This paper considers cloud interactions among different parties—clients, service provider, and cloud provider—on a cloud confidential computing platform using CVMs. Fig. 3 illustrates the system model.

Clients possess sensitive data (e.g., healthcare or financial information) that they want to send as input to a remote data processing service. Clients do not trust the service provider, the cloud provider, or other clients. A client assumes that these parties may try to steal its sensitive data.

The **service provider** designs application software that offers a cloud data processing service to multiple clients. This service is deployed within a CVM to appease client privacy concerns related to processing sensitive data on untrusted remote machines and/or fulfilling governmental regulatory requirements (e.g., HIPAA [15], GDPR [11]). Meanwhile, service providers rely on CVM to protect their own intellectual property, given that they may not want to disclose their close-sourced proprietary algorithms/source code to untrusted hosts. For cost-efficient deployment, the service provider serves multiple clients using one CVM.

We assume the service provider is *honest but curious*—they deliver functionally correct services, but collect client data for monetary gain, such as selling it to advertising or analytics agencies [3]. Honesty can be verified by clients querying multiple service providers and comparing results, and providing incorrect services would damage the provider's reputation. However, data leakage is stealthy and hard for clients to detect, as they do not control the software stack (§3.2).

Finally, the service provider rents out a CVM from the **cloud provider** (e.g., Azure). We assume the cloud provider is also *honest but curious*, and they may collude with the service provider to collect client data.

Service application model. We consider request-response service applications, where a client submits a request alongside the data and receives the corresponding response(s). For each client request, we assume that the service application is self-contained on the service provider's CVM (not distributed across multiple service providers). Furthermore, our focus is on CPU-based applications, excluding those that rely on heterogeneous accelerators (e.g., FPGAs). Supporting such external devices would require the integration of trusted I/O mechanisms for CVMs, which constitutes an orthogonal and complementary area of research [16, 19].

Real-world example scenarios. Cloud services like image processing and graph processing [12] accept multi-modal data input for various purposes, such as OCR scanning [26], image segmentation [36], and graph computations. The input data—such as private photos, documents, and user social network graphs from corporate clients [47]—must be protected against leakage. Moreover, CPU-powered AI inference [32] enables cost-effective machine learning (ML) models on cloud machines to serve client queries. Clients need to protect query data, which may include sensitive personal or financial information. Other examples of services include information retrieval (e.g., from healthcare databases [10]) and cloud-based intrusion detection [9], using corporate data like employee network traffic and logs.

3.2 Threat Model and Attacker Capabilities

We assume the attackers, including both service providers and cloud providers, control the CVM's OS kernel and unprivileged software (service programs), as well as the host hypervisor. In addition to attacks that the traditional CVM model considers (i.e., software or device-based attacks from the hypervisor), we consider the following attack vectors:

Table 1. Comparison between Erebor (this work) and existing CVM data protection solutions.

Protection system	Approach	Data Protection			No cloud infra. change	
		AV1	AV2	AV3	Paravisor[1]	Hypervisor
Veil [43]	Enclave	✓	✗	✗	✗	✗
NestedSGX [89]	Enclave	✓	✗	✗	✗	✗
Erebor	Sandbox	✓	✓	✓	✓	✓

[1] Paravisor [22, 28] denotes the privileged component deployed into the CVM by cloud providers, using CVM partitioning features (e.g., VMPL, §3.3).

- **AV1: OS data retrieval.** The OS[*] may direct the CPU to read data from the program's memory regions or convert these regions to shared CVM memory (§2.1) and retrieve them using device DMA. The OS may also access data by reading the program's state (e.g., register values) while handling interrupts or exceptions.
- **AV2: Program *direct* data leakage.** The service program may leak data to the OS or hypervisor by using system or hypervisor call functionalities, such as writing to the disk file system or sending data over the network.
- **AV3: Program *covert* data leakage.** The service program may covertly transmit data to the OS or hypervisor by encoding it into parameters (i.e., arguments) and frequency of system and hypervisor calls [41, 60]. Additionally, a program may trigger and send user-mode interrupts [35] to leak data to other attacker-controlled processes without exiting to the privileged software.

Out-of-scope. We do not consider data leakage through digital timing-based or micro-architectural side channels. We discuss their mitigation in §12. Also, we exclude physical side channels due to their noisy and expensive nature [66].

3.3 CVM Data Protection Solutions and Limitations

Protecting data in our model from aforementioned attack vectors (**AV1–AV3**) requires enforcing additional isolation within CVMs against untrusted software. In this regard, the approach taken by recent systems [43, 89] is to leverage CVM partitioning features, particularly AMD Virtual Machine Privilege Levels (VMPL) [44]. Specifically, Veil [43] and NestedSGX [89] leverage VMPL to instantiate a privileged security monitor within an AMD SEV-SNP CVM. Using the monitor and its partitioning capabilities, they further isolate sensitive data into an *enclave* partition. Like SGX [71], these enclaves provide *one-way isolation*, disallowing the untrusted OS to access program data [41, 60].

There are two problems with employing the aforementioned solutions to protect data in our system model (Tab. 1).

Partial data protection capabilities. Like SGX, Veil and NestedSGX consider the software inside an enclave to be trusted. Thus, the enclave is allowed to communicate with the operating system using system calls and hypervisor us-

[*] Unless otherwise specified, the term "OS" refers to the CVM guest OS.

ing hypercalls. Malicious programs can use this interface and disclose data to the attacker (**AV2-AV3**). Therefore, full data sandboxing with such enclave solutions would require additional Software Fault Isolation (SFI) techniques [41, 60, 88].

Cloud infrastructure support required. CVM partitioning enclave solutions require support from a cloud provider in terms of making changes to the hypervisor and, in emerging cloud deployments, the paravisor [4, 28, 97]. Specifically, changes to the host hypervisor are needed to support the transition between enclaves and OS for scheduling, interrupt, and exception handling [43]. Additionally, in emerging deployments, cloud providers leverage CVM partitioning features to deploy a trustworthy paravisor inside CVMs and provide cloud infrastructure services like live migration and firewall management to cloud tenants [22]. In such deployments, features like VMPL are under the control of the paravisor, and implementing solutions like Veil or NestedSGX would require support from the cloud provider. This creates a non-trivial barrier for service providers (who are simply cloud tenants in our model) to deploy such solutions.

4 Erebor Overview

Erebor is a sandboxing architecture for CVMs that enforces full data protection (against **AV1-AV3**), and can be deployed as a drop-in solution without changes to the cloud software infrastructure (i.e., hypervisor or paravisor).

This section describes the assumptions behind our work, and Erebor's architectural components. Since Erebor's prototype is designed on TDX, we describe the following sections using TDX. Moreover, for simplicity, we only describe a native CVM without a paravisor deployment. We discuss our analysis and experiments to show Erebor's compatibility with other CVMs and paravisor deployments in §10.

4.1 Assumptions

We trust the CVM hardware and assume that it is free of or patched against hardware defects [62, 69, 73] and malicious interrupt injections [78, 79]. In addition, we trust the correctness of the CVM remote attestation process, including the hardware mechanisms and software protocols. Moreover, we trust the CVM boot firmware, like Open Virtual Machine Firmware [27], provided by the hardware vendor. Last, in the case of a paravisor-enhanced CVM, we assume the paravisor is trusted, as they are independently built (e.g., Coconut-SVSM [4]), open-sourced, and can be formally verified [97].

4.2 System Architecture and Components

Instead of using CVM partitioning features (§3.3), Erebor instantiates a lightweight privileged CVM security monitor, dubbed **Erebor-Monitor**, using *intra-kernel privilege isolation* [45, 50, 51] techniques (Fig. 4).

The core idea is to isolate the hardware kernel privilege mode (i.e., ring-0 in TDX) into two virtual modes—***privileged*** and ***normal***. In privileged mode, the monitor manages critical system functions. The kernel, in normal mode, handles the remaining functions, keeping the monitor's Trusted Computing Base small. Using this core idea, Erebor implements a new security monitor that is specially crafted for CVM sandboxes and leverages novel hardware features (e.g., Protection Keys and Control Enforcement Technology) to ensure performance and security (§5).

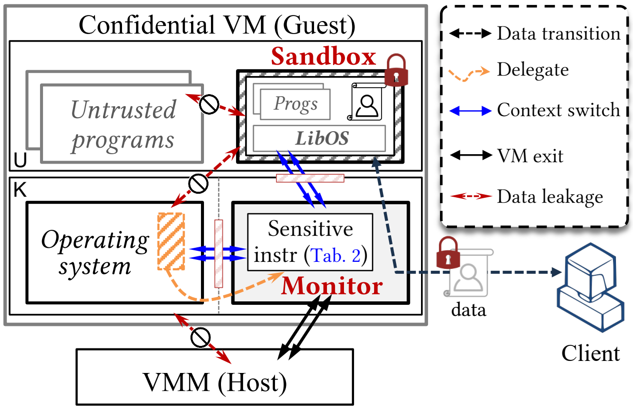


Figure 4. Erebor overview. The system achieves sandboxing through an intra-kernel security monitor design, preventing the untrusted OS from executing a subset of sensitive privileged instructions by delegating them to the monitor (▨).

There are key benefits of Erebor-Monitor's implementation in terms of providing sandboxing capabilities in a *drop-in* manner. For sandboxing, executing in the (privileged) CVM kernel mode, the monitor can (a) naturally intercept all interactions between user programs and the operating system kernel, as well as the outside hypervisor (§2.1) and (b) control all guest memory configuration interfaces (e.g., page tables) to further enforce isolation. In terms of a drop-in solution, the monitor can be designed using techniques and features that are available natively within a guest virtual machine boundary and do not require interactions with the hypervisor (e.g., like VMPL does for switching partitions [43]).

The monitor creates special sandboxed containers, called **Erebor-Sandbox**, with each container dedicated to processing a single client's sensitive data and restricted from any direct or covert communication with external components (§6). The container executes a service provider's developed programs with a Library Operating System (LibOS) [81, 85, 87]. All programs within the sandbox container share the same address space—each process task is converted into a thread. From Erebor's perspective, this approach simplifies the task management and secure communication enforcement amongst tasks within the same container. At a high-level, Erebor enforces the following three data protection policies on its sandbox runtimes:

- To prevent data retrieval or disclosure through memory, Erebor ensures that all *writable* regions belonging to sand-

Table 2. Sensitive privileged instructions and descriptions.

Type	Instruction	Sensitive instruction usage description
CR	`mov %r, %CR`	Write CR0/3/4 to control MMU page table and enable hardware kernel protection features.
MSR	`wrmsr v, %MSR`	Configure guest-controlled hardware kernel protection CPU features (e.g., PKS and CET).
		Control system call context switch interface.
SMAP	`stac`	Temporarily grant the kernel mode with read and write permissions to user memory.
IDT	`lidt v`	Control #INT/excpt. context switches.
GHCI	`tdcall`	Request TDX module to convert CVM shared and private memory for device access.
		VMexit to the VMM to handle general events.
		Request TDX module for attestation digest.

boxes (where data is processed) are isolated from all outside software. For efficiency, different sandbox instances are allowed to share common memory regions (e.g., large in-memory databases) in a *read-only* manner (§6.1).

- To prevent data leakage during exits, Erebor interposes all software-controlled exits and ensures secure handling. Once client data is received in a sandbox, the monitor disables all system calls and synchronous exits (including hypercalls) from the sandbox. To protect the program state from being exposed at external interrupts, the monitor securely saves, masks, and restores it (§6.2).
- To prevent data leakage during communication, Erebor relays data between clients and sandboxes through an end-to-end secure communication channel (§6.3).

5 Monitor Privileged Mode Enforcement

This section describes how Erebor virtualizes a *privileged* mode in ring-0 to host Erebor-Monitor, while depriviledging the untrusted kernel to the *normal* mode. At a high-level, the goal is to allow Erebor-Monitor the exclusive ability to execute a set of sensitive privileged instructions (Tab. 2) that are critical for sandboxing enforcement (explained in §6) in a controlled manner. This is achieved using offline instrumentation (§5.1), runtime controlled memory configuration interfaces (§5.2), and gated secure monitor calls (§5.3).

5.1 Kernel Instrumentation and Verified Boot

Erebor instruments sensitive privileged instructions in kernel source code, replacing them with secure calls to the monitor (§5.3). This allows the monitor to enforce isolation policies before executing these instructions. In the future, we expect that the instrumentation could be directly integrated into mainstream kernels, similar to how Xen-based paravirtualization was patched into Linux [29]. Instrumentation correctness is verified by a *two-stage* boot process.

In the first stage, only the trusted boot firmware (§3.2) and the Erebor-Monitor binary are loaded into the CVM. At this time, both components are measured by the CVM's attestation mechanisms, and included within any requested attestation digest. Note that the firmware is open-sourced [27], and the monitor will be open-sourced; thus, a remote client can attest that these components are correctly loaded.

In the second stage, the monitor receives the instrumented kernel image, scans it to verify proper instrumentation, and then loads the kernel. Unlike typical software fault isolation (SFI) systems [41, 60, 81, 93], which require complex binary disassembly to verify instrumentation at the native instruction level, Erebor only needs to ensure that no instruction byte sequences form sensitive instructions. Thus, it only performs byte-level scanning of the executable sections [51] using an ELF loader. If verification passes, the monitor loads the kernel and performs relocations, thereafter allowing the (deprivileged) kernel to execute.

5.2 Virtual Privilege Memory Interface Control

Erebor-Monitor controls memory interfaces to restrict CPU and device memory views and enforce isolation policies between the kernel and itself. Specifically, the kernel is restricted from bypassing instrumentation to execute sensitive instructions or compromising the monitor's memory to break such privilege separations.

Controlled MMU and DMA configuration interfaces. For CPU access, the monitor exclusively manages the CVM's physical MMU interfaces, controlling memory mappings and page attributes. This is by following the Nested Kernel principles [51]—only the monitor can execute instructions that directly or indirectly modify the MMU state and page table pages and entries (PTPs/PTEs). This is achieved by (a) sensitive instruction instrumentation and (b) using the kernel protection key feature (PKS, §2.3) to restrict memory access restrictions (see §10 for alternatives on other platforms).

To control the MMU state, the monitor instruments MMU-related control and model-specific registers (`CR`/`MSR` in Tab. 2). To control PTPs, the monitor marks PTP memory as writable only by the monitor (*read-only* for normal mode). This is achieved by assigning a protection key to PTPs, granting the key *non-writable* permissions in the normal mode.

For device access, the monitor exclusively controls GHCI interface (Tab. 2) to restrict allowed DMA mappings (CVM shared memory) for devices.

Kernel and monitor isolation policy enforcement. The monitor enforces that the isolated kernel cannot execute sensitive instructions within its own or userspace memory, nor access the monitor's memory.

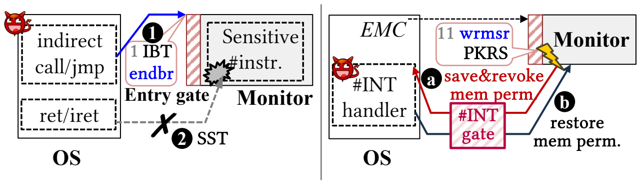
The monitor enforces Write-xor-eXecute (W⊕X) permissions when executing the kernel. Thus, the kernel cannot update its memory to execute sensitive instructions. To do so, the monitor assigns a *non-writable* protection key to all kernel code pages. For the remaining kernel writable pages, the monitor sets those PTEs' Non-eXecutable (NX) flag. There are cases in which the kernel may update its executable code

```
1  EntryGate: (.att_syntax)
2    endbr64 ;Track indirect branch
3    ;Save scratch registers
4    mov %rax, -8(%rsp)
5    mov %rdx, -16(%rsp)
6    mov %rcx, -24(%rsp)
7    ;Grant monitor access perm.
8    mov IA32_PKRS, %rcx
9    mov $GRANT_ALL, %rax
10   wrmsr IA32_PKRS
11   ;Switch to monitor stack
12   mov %rsp, %rcx
13   mov perCPU(SecureStack), %rsp
14   push %rcx ;Save OS stack ptr
15   ;Restore scratch registers
16   mov -8(%rcx), %rax
17   mov -16(%rcx), %rdx
18   mov -24(%rcx), %rcx
```

(a) Entry gate.

```
1  ExitGate:
2    ;Finished monitor execution
3    ;Switch back to OS stack
4    mov 0(%rsp), %rsp
5    ;Save scratch registers
6    push %rax
7    push %rcx
8    push %rdx
9    ;Revoke OS access perm.
10   mov IA32_PKRS, %rcx
11   mov $REVOKE_OS_R_W, %rax
12   wrmsr IA32_PKRS
13   ;Restore scratch registers
14   pop %rdx
15   pop %rcx
16   pop %rax
17   ;Return to the OS
18   ret
```

(b) Exit gate.



(c) HW-CFI (left) / Interrupt gate (right) guarded EMC execution.

Figure 5. Entry/exit gates for Erebor-Monitor-Call (EMC) gates and how they are enforced at runtime.

pages, such as loadable kernel modules and eBPF bytecode. In these cases, the kernel requests the monitor to scan and verify the code before loading it [43, 51].

To prevent the kernel from executing sensitive instructions in userspace pages, the monitor always enables kernel-user execution separation, i.e., Supervisor Mode Execution Prevention (SMEP) [20]. The monitor achieves it by setting CR4.SMEP bit (Tab. 2). SMEP enforces the kernel mode cannot execute user-level pages (i.e., PTEs with U/S bit as 1).

To ensure the monitor's integrity and data confidentiality, the kernel is not allowed to access the monitor's memory regions (including the code, data, and execution stacks). To do so, the monitor assigns a protection key to its own pages. That key grants the normal mode kernel *non-accessible and non-writable* permissions to those monitor pages.

Finally, to restrict untrusted device access, the monitor ensures that any CVM memory converted to shared (by tdcall to the TDX module) will always reside in the system-reserved region for devices. Thus, the kernel and monitor are private and inaccessible by devices (§2.1).

5.3 Gated Secure Monitor Calls

Erebor implements a secure interface—Erebor-Monitor-Call (EMC)—for the OS to request sensitive instructions. Each EMC is bounded by an *entry* and *exit* gate, clearly defining the boundaries for privilege transitions. The OS can only honestly perform EMCs to request privileged instructions.

Fig. 5-(a,b) illustrate the entry and exit gates. The entry gate grants full access to the monitor's memory and switches to a protected per-core execution stack. To grant memory permissions, the entry gate modifies the IA32_PKRS MSR of the current core to permit *read-write* access to the monitor's memory (line 10-12). Subsequently, the execution stack is pointed to the monitor's per-core stack (line 14-15). Throughout the entry gate, the OS's execution state is securely preserved (line 4,16), ensuring a safe and reversible context switch. Upon completing the request, the exit gate reverses the operations and returns to the OS (Fig. 5b).

HW-CFI guarded deterministic EMC execution flow. Erebor leverages HW-CFI enabled by CET (§2.2) to ensure that the OS can only deterministically jump to the start of the entry gate to enter the monitor and perform an EMC. Thus, the privilege transition is always safe, and the monitor always enforces proper isolation policies (§5.2) when executing sensitive instructions (by setting target values).

Fig. 5c-left illustrates the workflow. Erebor ensures that only the start of the entry gate has an endbr64 (line 2) and no other part of the monitor does. As a result, forward indirect jmp or call can only jump to the entry gate start (❶); any attempt to target arbitrary monitor code triggers a #CP. Additionally, the shadow stack prevents backward function or exception returns (ret/iret) from arbitrarily altering control flow into the monitor's code (❷).

To ensure HW-CFI is always active, the monitor configures its control interfaces (IA32_S_CET MSR and CR4.CET bit). For backward integrity, the monitor maintains a shadow stack memory by configuring IA32_PL0_SSP MSR. Per CET specification [80], kernel shadow stack pages are always *non-writable-but-dirty* (by clearing PTE R/W bits, enabling dirty bit and CR0.WP). Unlike task switches, which require switching corresponding shadow stacks (§2.2), executing EMC does not. This is because, from the kernel's perspective, an EMC behaves like a normal function call.

Interrupt gate guarded EMC-exclusive permissions. Once the entry gate is invoked, Erebor ensures that only the CPU under EMC execution is granted memory permissions until completes at the exit gate. In other words, during EMC, the OS cannot preempt the execution while retaining the granted permissions to access monitor memory.

The OS may inject Inter-Processor Interrupts (IPIs), or a hypervisor may inject device interrupts to preempt EMC. After a CPU gains monitor access permissions at the entry gate (line 11, Fig. 5a), such interrupts may be triggered to redirect execution to the OS. The monitor ensures that these interrupts are handled safely by routing them through a special and protected *interrupt gate* (#INT gate in Fig. 5c-right). To do so, the monitor controls the exception vector handlers to wrap interrupts with #INT gate (IDT in Tab. 2). At the interrupt entry, the #INT gate (a) saves all general-purpose registers, (b) saves the current memory permissions (i.e., IA32_PKRS value) onto the secure stack and revokes the permissions to access monitor memory (ⓐ), and (c) restores

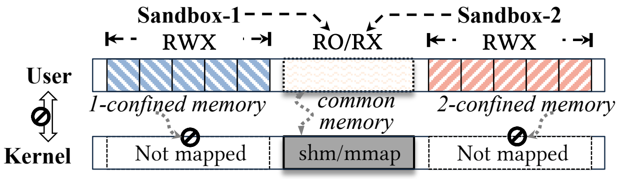


Figure 6. Erebor-Sandbox memory and permissions.

general-purpose registers and jumps to the OS interrupt handler. At interrupt returns, the gate performs the same operations to restore memory permissions (❺).

6 Sandbox Data Protection Enforcement

Using its privileged monitor (§5), Erebor efficiently enforces data protection within Erebor-Sandbox through memory isolation (§6.1) and sandbox exit protections (§6.2), while also establishing secure communication channels for data transfer between the sandbox and clients (§6.3).

6.1 Resource-Efficient Sandbox Isolation

Erebor ensures that all memory regions of the sandbox process must be declared before usage, and enforces strict memory permissions based on the declared type.

Confined and common (shared) memory declaration. Sandbox memory has two types—confined and common, and Fig. 6 describes those types and permissions. Confined memory is used for sandbox process code, data, stacks, runtime heap, and also for holding client data. Thus, the sandbox *exclusively* owns all permissions to its confined memory. On the other hand, common memory is *read-only* to a sandbox, thereby allowing shared instances (e.g., ML model, databases, or shared libraries) across sandboxes.

During initialization, the sandbox sends a request to the monitor and declares all required confined memory regions (§7). This is transparently done by the LibOS, which leverages a special device driver to send EMC. Note that the service provider is responsible for setting up the memory budget (hard limit) for a container's confined memory.

Erebor-Monitor allocates confined sandbox pages from a reserved memory region (§7) and *pins* them (i.e., no swapping to disk). Pinning of confined pages is primarily to avoid leaking secrets by page faults, and in the future, a secure page fault handling mechanism can be implemented [76]. In contrast, common pages are large and do not contain secrets, thus they are not pinned. Rather, the monitor allocates them using its default filesystem (shm) backend or using anonymous shared memory during *mmap* (MAP_ANONYMOUS).

Access permission restrictions on sandbox memory. Erebor-Monitor controls the MMU interface to enforce CPU access permissions on sandbox memory (Fig. 6). Device DMA access permissions to the sandboxes are further restricted by controlling the GHCI interface (§5.2).

To prevent other processes from accessing a sandbox's confined pages, the monitor keeps a *single mapping* policy for these pages. Once a page is declared as confined and mapped to a sandbox, the monitor refuses to map it further to other processes (or even the kernel) page tables. This prevents double-mapping attacks to access confined pages.

To further prevent the kernel from accessing sandbox confined pages within the same task context (e.g., at interrupts), the monitor enables kernel-user memory access separation, specifically Supervisor Mode Access Prevention (SMAP) [20]. The monitor always sets the `CR4.SMAP` bit (`CR`, Tab. 2), thereby enforcing that the kernel cannot access user-level pages by walking the sandbox's user page table.

Note that SMAP can also be temporarily ignored in the kernel by executing a `stac` instruction. Thus, Erebor treats it as a sensitive instruction and removes it from the kernel (`SMAP` in Tab. 2). One side-effect is that the kernel cannot access user regions even for native (non-sandboxed) programs, which is required for native system call handling such as `read/write` with userspace buffer parameters. To support that, the monitor interposes the *user copy* [14] interfaces (i.e., copy_from/to_user()) from the OS. When such a request happens, the monitor emulates the operations on behalf of the OS and then revokes the temporal user-level page access permission (using the `clac` instruction) afterward.

Erebor-Monitor enforces sandbox *read-only* permissions on common memory at runtime, by controlling page table attributes. Before receiving client secret data (§6.3), sandboxes can normally write to common memory to initialize shared instances. Once client data is loaded, the monitor clears the Write (W/R) bit in the relevant page table entries (PTEs), revoking sandbox write access permission.

Finally, the monitor achieves device access prevention by controlling `tdcall` (GHCI), always marking all sandbox memory as CVM private memory (§5.2).

6.2 Sandbox Runtime and Exit Protection

Once Erebor-Sandbox receives client data, the sandbox required critical runtime services [46, 86] are emulated by the LibOS. To further prevent untrusted programs from leaking data by compromising the LibOS and inducing system calls or VM exits, Erebor-Monitor intercepts sandbox software-controlled exits alongside asynchronous exits (e.g., interrupts) to the OS, ensuring secure handling.

Runtime system service emulation within the LibOS. The LibOS emulates the following four runtime services.

1) Heap memory management. During initialization, the LibOS pre-allocates all memory and declares them as confined memory for sandbox runtime heap allocation (*brk/mmap*).

2) In-memory stateless filesystem. The LibOS *preloads* all required files (e.g., libraries) into the sandbox and creates in-memory mountpoints before receiving client data. Once the client data is received, the sandbox operates statelessly for the remainder of its execution by creating temporary in-memory files (maintained in confined memory).

3) Multi-tasking and synchronization. The LibOS supports

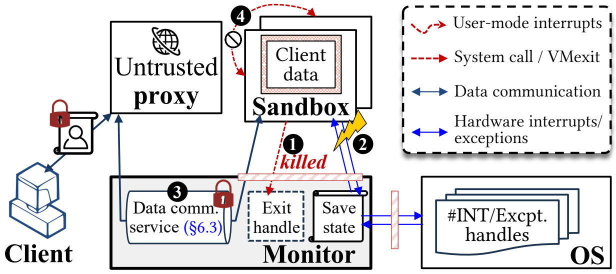


Figure 7. Erebor-Sandbox exit interposition procedure during runtime, after receiving and installing client data.

sandbox multitasking using threads. The maximum number of threads is predetermined, and all threads are created during initialization (by *clone*). Synchronization between normal threads is enabled using system calls like *futex*, but this is not possible in sandboxes as they are disabled during data processing. Following the practice of SGX SDK [39], Erebor's LibOS manages userspace synchronization internally using its own spinlock. This increases resource utilization due to busy-waiting, but avoids covert channel leaks.

4) Client data communication services. Erebor does not allow the sandbox to establish network connections. Thus, the sandbox relies on the monitor to communicate data with the client (i.e., receiving client input data and sending output data to the client, further explained in §6.3).

Sandbox exit interposition and protection. All exits from the sandbox are intercepted by the monitor, which inspects the exit reason and handles it accordingly.

To interpose exits, the monitor controls the sandbox's context switch interfaces (`MSR`, `IDT`, and `GHCI` in Tab. 2). Specifically, user-and-kernel system call is interposed by loading a special entry (through `IA32_LSTAR` MSR), while exceptions and interrupts are interposed by a special exception vector table entry. Those special entries redirect execution to the monitor's handler first. On the other hand, the monitor setting user-target table (`IA32_UINTR_TT` MSR) to determine whether to disable user-mode interrupts (explained below).

Once client data is loaded, the monitor kills the sandbox if it exits through system calls, or due to VM exits (identified by #VE, ❶ in Fig. 7). It is acceptable to disable VM exits—from our observation, a benign program only performs hypercalls for `cpuid` [34]. To support `cpuid`, the monitor emulates it by requesting to the hypervisor once and caching the results. For normal interrupts (e.g., scheduler events), the monitor saves the sandbox state and then masks it, before jumping to the OS handlers (❷). After the OS handles them, the control flow is redirected back to the monitor (to resume the sandbox). Recall that the LibOS requests the monitor for client data communication. The monitor internally handles it in such cases (❸, further explained in §6.3). Finally, the monitor disables sandboxes to send user-mode interrupts by clearing `IA32_UINTR_TT.valid` (❹) before entering them.

6.3 Secure Data Communication

Erebor-Monitor establishes a secure channel with a remote client to exchange data into Erebor-Sandbox's confined memory and return results. The monitor does not have direct network access; rather, it uses an untrusted *proxy* in CVM (Fig. 7) to send and retrieve network packets.

Client-monitor secure channel establishment. Prior to any client data communication, the monitor establishes a secure channel with the client using remote attestation (§2.1). In an Erebor-enabled CVM, only the monitor is allowed to establish attested secure channels—only it can execute a `tdcall` (Tab. 2) to request a signed attestation digest from the TDX hardware, which it then sends to the client. The client verifies the signature, attests to the boot state measurement in the digest, and uses the information provided to establish a secure channel based on authenticated key exchange.

As we discussed in the verified two-stage boot (§5.1), only the monitor and firmware are loaded into a CVM first. As a result, a remote user can always attest that the correct security monitor is loaded into the CVM and that they are communicating with it, before establishing a secure channel.

Secure sandbox data communication and cleanup. Data received through the secure channel is directly written into Erebor-Sandbox by the monitor, and the results are securely sent back through the monitor for transmission.

To support the data communication services for the sandbox (§6.2), the monitor exposes an *ioctl* system interface with a reserved file descriptor to the LibOS. Such *ioctl* calls are intercepted by the monitor (❸, Fig. 7), which checks the descriptor and securely (a) writes client data into sandbox memory or (b) reads results from sandbox. While output data is encrypted, its size may still covertly leak information. Thus, the monitor pads the output data to fixed-lengths before returning it to the client [60]. After all results are sent back and a client session is terminated, the monitor clears (zeroes) the sandbox's memory regions, including its in-memory file system and internal thread contexts.

7 Implementation

Erebor-Monitor and guest OS kernel. Our monitor is implemented on Linux v6.6.0, which is also used as the CVM guest operating system. Our instrumentation modifies the kernel with ~4.5k SLoC of C code and ~0.3k SLoC of assembly code. The current implementation disables huge pages (i.e., 2MB and 1GB), and the monitor's memory region is also aligned with 4KB pages. Disabling huge page is done to simplify PKS-based permission settings (§5.2) on kernel's direct mapping of all physical memory (and kernel text mapping). This avoids the hurdle of setting fine-grained permissions when kernel memory mappings are at coarse granularities, particularly when Erebor has to modify permissions for a subsection inside a huge page. In the future, forced page splitting can be implemented when setting protection keys within a huge page—by splitting the huge page into 4KB pages and setting the protection keys only for the required split pages. Erebor requires W⊕X permissions on kernel's code sections. To simplify implementation, we currently disable the loadable kernel module (LKM) and eBPF. However, certain kernel operations—such as self-updating its code using `text_poke`—require modifications to these sections. To support this, we instrument the corresponding poke functions (e.g., `text_poke_f`). Thus, the monitor can validate and update the OS code on behalf of the kernel.

Erebor-Sandbox Library OS (LibOS) toolchain. Our LibOS is an extension of Gramine [87]. We made ~2.9k SLoC modifications (§6.2). For the memory backend, we reserved a physical region based on Linux Contiguous Memory Allocator to serve sandbox confined memory. We created a driver under `/dev/` in the untrusted OS to help issue EMC and declare memory. We modified the LibOS loader, replacing its `mmap` backend with the driver, in which it (a) allocates confined memory, (b) performs a EMC to declare the memory, and (c) populates and pins the sandbox page table.

Derived from Gramine, Erebor's LibOS supports many applications with minor modifications. Specifically, Gramine supports POSIX APIs and over 170 Linux system calls, covering most commonly-used functionality, allowing it to natively run complex Linux applications. The small modifications to such applications required by Erebor are mainly so that they can operate within a confined environment—particularly concerning the secure data communication channel (§6.3). These modifications can be achieved with only tens of lines of code using the `ioctl` interface (§6.3).

To illustrate Erebor required program modifications concretely, we rely on a real-world application used by our evaluation—LLAMA.cpp (presented in details in §9). Such a service application receives client input prompt (data), performs large language model (LLM) inference subsequently, and returns inference results. Listing 1 illustrates the main modification to support Erebor-Sandbox.

Last, admittedly, LibOS compatibility still has challenges due to Erebor's single-address space model (§4.2). Nonetheless, we have seen advancements in the recent solution [81], which securely supports single address space for multi-tasks by replacing *fork* with corresponding *spawn*-like supports.

Host hypervisor (Linux KVM). We simply employed an Intel-maintained KVM hypervisor, which supports PKS virtualization and will be upstreamed to the mainline Linux [21]. ***Limitations:*** Linux's HW-CFI by CET (§2.2) currently supports only forward integrity (IBT), with kernel shadow stack still under development [8]. Thus, our prototype omits backward checks. However, these checks have minimal performance impact [80, 92], and are by default deployed in the latest kernel like Windows 11 [37]. Our monitor interposes all exits from userspace, slightly increasing non-sandbox execution overhead. This can be avoided by isolating IDTs and syscall entries for normal programs and sandboxes [59, 95].

Listing 1. Erebor-Sandbox software modification.

```
/* Application: LLAMA.cpp */
@@ Support Erebor's I/O @@
+ struct io_payload {
+    char *buf;
+    size_t size;
+ };
+ int dev_fd = open("/dev/erebor-psudeo-io-dev", O_RDWR);

@@ common.cpp ++ support Erebor's input channel @@
bool gpt_params_find_arg(int argc, char ** argv, ...) {...
-       std::copy(std::istreambuf_iterator<char>(file),
-                std::istreambuf_iterator<char>(),
-                back_inserter(params.prompt));
+       struct io_payload input_p = {
+           .buf = new char[MAX_SZ]; .size = 0; };
+       ioctl(dev_fd, INPUT, &input_p);
+       params.prompt = std::string(input_p.buf, input_p.size);
}

@@ main.cpp ++ support Erebor's output channel @@
int main(int argc, char **argv) {...
-   std::ostringstream output_ss;
+   std::string output_buf;
+   struct io_payload output_p;
    while ((n_remain != 0 && !is_antiprompt)) {
        for (auto id : embd) {
            std::string token_str = llama_token_to_piece(id..);
-           printf("%s", token_str.c_str());
-           output_ss << token_str;
+           output_buf.append(token_str);
        }
    }
-   write_logfile(output_ss.str(), ...);
+   output_p.buf = const_cast<char *>(output_buf.c_str());
+   output_p.size = output_buf.size();
+   ioctl(dev_fd, OUTPUT, &output_p); ...
}
```

To facilitate implementation and evaluation, we rely on an untrusted filesystem—DebugFS—to emulate Erebor's data communication channel. That is, currently, instead of using the network relay, we program Erebor-Sandbox to interact data through a DebugFS interface.

8 Security Claims and Analysis

Through a series of security claims, this section shows how we protect Erebor-Monitor's integrity and enforce sandbox isolation against attackers.

Enforcing Monitor virtual privileged mode execution. The attacker's goal is to elevate privilege, by arbitrarily executing *sensitive instructions* (Tab. 2) while bypassing the monitor-enforced restrictions (i.e., proper target values and enforcement) on such instructions.

C1: *Erebor-Monitor will execute within the CVM first and only load a deprivileged kernel without sensitive instructions.*

The client uses remote attestation mechanisms to ensure the monitor is loaded. Before booting the kernel, the monitor scans and analyzes the kernel image and refuses to boot if sensitive instruction sequences are found (§5.1).

C2: *The deprivileged kernel cannot insert sensitive instructions in accessible memory regions and execute them.*

The monitor controls the MMU interface and PKS to enforce the kernel's W⊕X permissions to kernel pages. Any dynamic code (e.g., kernel modules) is validated by the monitor before loading. Moreover, SMEP is enabled to prevent the kernel from executing sensitive instructions from user pages (§5.2). Finally, the monitor controls GHCI to prevent malicious device DMA from accessing kernel memory.

C3: *The deprivileged kernel cannot harm the integrity of Erebor-Monitor to break sensitive instruction restrictions.*

With controlled MMU translation interfaces, the monitor assigns PKS protection keys to its own pages, prohibiting the kernel from remapping or accessing the monitor memory. The monitor also controls GHCI to ensure the monitor memory is private and devices cannot perform malicious DMA access to the monitor memory (§5.2).

C4: *The deprivileged kernel cannot bypass the restrictions of Erebor-Monitor-enforced sensitive instructions, by arbitrarily executing those instructions in the monitor memory.*

All sensitive instruction requests will deterministically go through the EMC with secure gates, in which the isolation policies are always enforced (§5.3). The monitor deploys HW-CFI to prevent any control flow violations without going through EMC gates, Fig. 5c-left). Moreover, the interrupt gate prevents any permission violations during EMC execution, revoking privileged permissions at interrupts (Fig. 5c-right).

Claim: *In the Erebor-protected CVM, the kernel relies on Erebor-Monitor for all sensitive instructions* (**C1** – **C4**).

Enforcing Sandbox memory and data protection. The attacker's goal is to (a) leak client data by exploiting the network and the communication channels, (b) retrieve data in sandbox memory, or (c) leak data from the sandbox.

C5: *Client data and processing results will only be exchanged within an end-to-end secure channel to Erebor-Monitor.*

The monitor controls the GHCI (Tab. 2) to generate attestation digests exclusively—no untrusted software can impersonate the monitor to establish secure channels (§6.3). Additionally, as data is encrypted by the channel, untrusted software, including the proxy program, cannot access it.

C6: *Attacker-controlled software outside Erebor-Sandbox cannot read data located within sandbox memory.*

Once client data is received, only the monitor will decrypt and install it into confined memory (§6.3). By controlling MMU mappings, the monitor ensures that confined memory is exclusively accessed by the sandbox. Moreover, SMAP is always enforced by the monitor during kernel execution, ensuring the kernel cannot access sandbox user pages (§6.1).

C7: *Erebor-Sandbox is not allowed to write to outside attacker-accessible memory regions.*

By controlling the MMU mappings, the monitor ensures that a sandbox is granted write permission only to its confined memory. Any other memory, whether owned by other software or sandbox common memory, is either unmapped or non-writable for the sandbox (§6.1).

C8: *All software-driven exits from Erebor-Sandbox to outside software are intercepted and protected by Erebor-Monitor.*

Table 3. Overhead (CPU cycles) comparison between different privilege-level transitions, including empty EMC, empty *syscall*, and *hypercall* (`tdcall` in a native CVM and `vmcall` in normal KVM guest). All costs are from round-trip calls.

Priv. trans. call	#Cycle	Times	Priv. trans. call	#Cycle	Times
EMC	1224	1×	**SYSCALL**	684	0.56×
TDCALL	5276	4.31×	**VMCALL**	4031	3.29×

Upon receiving client data, the monitor prohibits any sandbox software-controlled exits, except for the monitor-handled communication channel to support I/O (§6.3). Such exits include system calls, synchronous VM exits, sending userspace interrupts, and software exceptions (e.g., exceptions caused by illegal instructions or divide-by-zero). This is enforced by controlling the system call entry (`IA32_LSTAR` MSR), GHCI (`tdcall` instruction), userspace interrupt interface (`UINTR` MSR), and context switch (`IDT`) interfaces from the monitor. On the other hand, at external interrupts (e.g., invoked by external devices), the monitor always saves and clears the sandbox registers' state before exiting to the OS.

Claim: *Client data is securely sent to and received from a Erebor-protected CVM, within a sandbox boundary* (**C5** – **C8**).

9 Evaluation

System specifications. We used an Intel® Xeon® Platinum 8570 production server running Ubuntu 24.04 (Linux v6.8.0), with 56 physical (2.1 GHz) CPU cores, 1 TB memory, and 1 TB storage. We assigned 8 vCPU cores, 24 GB memory, and 100 GB virtualized (virtio) storage disk to the CVM. For the client-server experiments, the remaining resources of the host server machine were used to generate client workloads.

Evaluation settings. We first compared **Erebor** against **Native** (normal CVM). Then, for an ablation study, we evaluated real-world application performance by comparing **Erebor** against running applications in a normal CVM with LibOS (**Erebor-LibOS-only**). We also adopted two settings—**Erebor-LibOS-MMU** and **Erebor-LibOS-Exit**—to break down overheads of sandbox memory isolation (§6.1) and sandbox exit protection (§6.2), respectively.

9.1 Micro-Benchmarks

This section evaluates Erebor's overhead on delegated privileged instructions and general system events.

Privileged operation cost breakdown. Erebor delegates privileged instructions (Tab. 2) to Erebor-Monitor using an EMC (§6.1), our privilege transition call gates. It is important to understand the overhead of privilege transitions. We first evaluated the cost of an empty EMC and compared it with other privilege transitions, including system calls and hypercalls. Then, we evaluated the cost of individual privileged instructions when enabling Erebor's EMC, and compared it against native CVM execution.

Table 4. OS privileged instruction overheads (CPU cycles). Target **MSR**: `IA32_LSTAR`. Target **CR**: `CR0`. Target **GHCI**: `tdcall.tdreport` to generate attestation report. **MMU**: page table entry (PTE) update overhead. Native MMU update is measured by kernel's default `native_set_pte` for PTE write.

Privileged	#Cycle (Times)		Privileged	#Cycle (Times)	
Operation	Native	Erebor	Operation	Native	Erebor
MMU	23 (1×)	1345 (58.48×)	**IDT**	260 (1×)	1369 (5.27×)
CR	294 (1×)	1593 (5.42×)	**MSR**	364 (1×)	1613 (4.43×)
SMAP	62 (1×)	1291 (20.82×)	**GHCI**	126806 (1×)	128081 (1.01×)

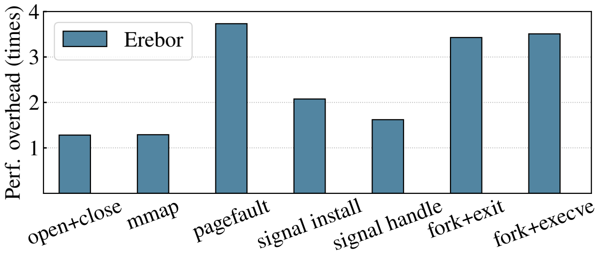


Figure 8. Erebor's overhead on LMBench (which ran as non-sandboxed normal workloads. From left to right, their EMC/second: 1.4M, 0.9M, 3.6M, 1.7M, 1.4M, 1.9M, 2.0M.

Tab. 3 shows the costs of different privilege transition calls, each executing an empty function. We found that EMC is twice as expensive as a system call, due to extra MSR operations during its entry and exit gates (Fig. 5). On the other hand, a hypercall in a TD guest (`tdcall`) incurs 3.3× more overhead compared to EMC, while in a normal (non-TD) guest, the hypercall (`vmcall`) incurs 2.3× more overhead. The increased hypercall latency in CVMs is because the TDX module has to protect the saved guest context state.

Tab. 4 illustrates the cost of privileged instructions. The primary overhead to all these operations is the EMC, but since the operations themselves take different times, their overall cost is different. Specifically, in terms of PTE updates (**MMU**), the native kernel performs few memory-write operations (only tens of cycles); thus adding an EMC increases the overhead significantly (58.5×). Nevertheless, PTE updates are generally infrequent (e.g., during dynamic allocation and process page table initialization). The impact of the extra cost along with the overhead (4.43 – 5.42×) from similarly infrequent updates to CRs, IDT, and MSRs is amortized during program execution (as shown in §9.2). In contrast, Erebor's overhead on tdcall is small. For instance, the majority of cycles in a TDREPORT are from report generation and integrity protection (i.e., HMAC key attachment) operations.

General system benchmarks (LMBench). We evaluated Erebor's performance overhead on system events using LMBench [72]. Note that LMBench runs non-sandboxed. The aim is to understand Erebor's overhead on system events, as Erebor's memory confinement and privilege instruction interposition are applied to the whole system.

Table 5. Real-world scenario and workload description.

Scenario (Prog.)	Workload Description
LLM inference (llama.cpp)	llama.cpp [23] with a **common** llama2-7b ~5GB model [25]. Default 8 threads and (**confined**) 256MB memory K-V cache. Prompted tasks of text translation and code generation.
Image processing (yolo)	NCNN, OpenCV, OpenMP framework with **common** model of Yolov5 [38]. Image segmentation with 100 input images.
Information retrieval (drugbank)	An ~400MB in-memory **common** database [31] populated by Drugbank [91]. Information retrieval by using 2.2M queries.
Graph processing (graphchi)	GraphChi [65] with 8 threads, and 2GB **confined** memory. Pagerank with an input graph Twitch-gamers (6.8M edges).
Intrusion detection (unicorn)	Unicorn [58] analyzer with 8 threads, 2GB **confined** memory cache and input of parsed 20MB log file.

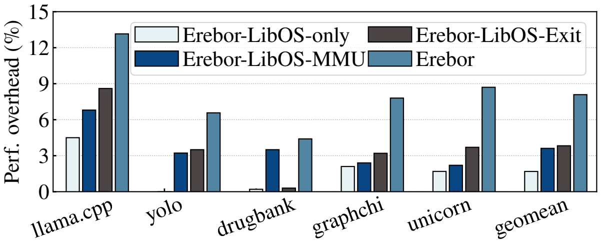


Figure 9. Erebor's overhead on real-world workloads.

Fig. 8 shows the results. Among all benchmarks, ***pagefault*** has the highest overhead (3.8× Native). During this benchmark, 3.5 million EMC were triggered per second. Also, it recorded ~6.4 million context switches, including 5.8 million system calls and 530 thousand exceptions or interruptions. On average, each context switch triggered 3.3 EMCs.

Similar to the *pagefault* benchmark, creating processes (i.e., the ***fork*** benchmark) also introduces high costs. This is because such operations involve extensive MMU updates (e.g., installing page table entries to handle page faults or copy process address space). From the statistics, the *fork* benchmarks include ∼2 million EMC per second, which incurs high costs. Such overhead could be lowered if batched MMU update is enabled [51]. Nevertheless, the impact of *fork* is one-time during process initialization, and Erebor's LibOS does not directly use forks (§6.2), while the other overheads are amortized during execution (illustrated in §9.2).

9.2 Target Programs

This section describes Erebor's performance in real-world service scenarios. Tab. 5 shows programs and workloads.

Results. Fig. 9 illustrates the runtime performance overhead incurred by these programs, while Tab. 6 provides detailed statistics related to exits incurred during execution, and memory and initialization overhead. The geometric mean overhead of Erebor over all programs is 8.1%. Considering breaking down the overhead into memory view isolation and container exit protection, the overheads amount to 3.6% and 3.9%, respectively, whereas the LibOS-only incurs a geomean overhead of 1.7%. Adopting the LibOS incurs a negligible overhead, as it emulates runtime system calls in userspace, avoiding considerable context switches to the kernel. Its

Table 6. Program execution statistics. **#PF**: page fault rate (per second). **#Timer**: APIC timer interrupt rate. **#VE**: virtualization exception rate. **EMC/s**: Erebor-Monitor-call rate. **Time**: data processing workload execution time with Erebor. **Conf.** and **Com.**: the amount of sandbox confined and common memory, respectively. **Init. Overhead**: program memory initialization overhead over Native execution.

Program	Sandbox-Exit (#exit/s)				EMC/s	Time.	Mem (MB)		Init.
	#PF	#Timer	#VE	Total		(sec)	Conf.	Com.	Overhead
llama.cpp	1.8k	0.9k	1.7k	4.4k	46.9k	52.85	501	4096	52.70%
yolo	1.2k	1.0k	1.3k	3.5k	77.6k	19.60	757	132	13.30%
drugbank	0.5k	0.5k	1.2k	2.2k	87.6k	12.89	814	400	28.50%
graphchi	0.8k	2.7k	0.7k	4.2k	40.9k	34.31	1340	–	36.80%
unicorn	0.7k	2.3k	0.9k	3.9k	39.5k	38.94	1254	–	31.20%

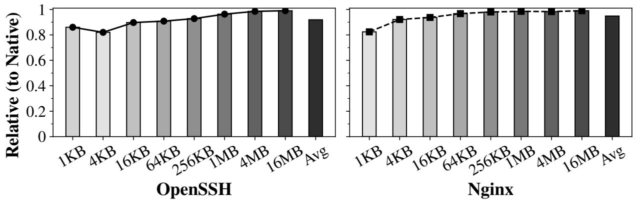


Figure 10. Relative throughput of background programs.

userspace synchronization incurs small overheads.

Among the programs, llama.cpp introduces the highest overhead at 13.15%, as it has a heavier workload. Its large memory usage induces more page faults for common memory. As a result, it has a considerable amount of runtime sandbox exits (4.5k in total exits per second due to different exceptions and interrupts) and EMCs (around 47k per second). Furthermore, with the setting of LibOS-only, Llama.cpp incurs an overhead of 4.5%, which is higher than that observed in other programs. This is because we observed frequent task synchronizations during its execution. Thus, the LibOS causes more overhead due to synchronization.

With common memory sharing, we observed a 0.15–9.2× reduction in memory usage, cutting consumption by up to 89.1% for a single sandbox. For instance, without memory sharing in llama.cpp, a 4GB model must be replicated across 8 containers, requiring ≈36GB memory. This was reduced to ≈8GB in our experiments. Larger shared regions would further enhance memory savings. Erebor increased initialization time by 11.5–52.7%, as pre-allocating container memory triggers many page faults. As shown in §9.1, page fault handling time increases due to EMCs. Nevertheless, this is a *one-time* cost, and containers can be pre-initialized in real settings (i.e., by adopting *warm-start* techniques [60, 68, 96]).

9.3 System-Intensive Background Programs

Erebor's memory confinement (§6.1) and privileged instruction interposition (Tab. 2) are enforced system-wide, incurring overhead for normal (non-sandboxed) programs. These programs manage VMs and serve as proxies (§6.3). We evaluated this overhead using several I/O-intensive workloads:

Table 7. Cross-CVM architectural features for Erebor.

Plat.	CVM general interfaces			Hardware protection features			
	Registers	Ctxt. switch	GHCI	Kernel-user separation	Prot. key	HW-CFI	
						Fwd.	Back.
TDX	CR/MSR	IDT	`tdcall`	SMEP/SMAP	PKS	IBT	SST
SEV	CR/MSR	IDT	`vmgexit`	SMEP/SMAP	*page table*	IBT	SST
CCA	EL1 Regs	VBAR	`smc` [5]	PXN/PAN	PIE [30]	BTI [2]	GCS [33]

- **OpenSSH** server under default settings; sending 1k file transfer requests from 8 client threads.
- **Nginx** server under default settings; using *ab* [1] to generate 1k client file requests (32 concurrency).

* *Clients and servers run on the same machine. All transferred file sizes varied from 1KB to 16MB.*

Fig. 10 shows the results. Across file I/O tasks, we found that the average throughput reductions of OpenSSH and Nginx are 8.2% and 5.1% (with maximum overheads 18% and 17.6% on small-sized files), respectively. For larger files, the performance is close to that of native with less than 5% throughput reduction. This is due to the increased frequency of system events during small file transfers, which leads to more frequent Erebor interpositions and a higher relative overhead. For larger files, the overhead is effectively amortized over the extended data transmission.

9.4 Key Takeaways

While Erebor increases system event latency by up to 3.8×, this overhead is amortized during execution. At initialization, programs incur a *one-time* overhead of 11.5% to 52.7% due to page faults. At runtime, the sandbox adds a modest geomean overhead of 8.1%, while saving up to 89.1% memory by secure sharing. Thus, we find Erebor practical for real-world use.

10 Platform Compatibility

Non-TDX architectural compatibility. The hardware features leveraged by Erebor to establish a security monitor and enable sandboxing are generally available in AMD SEV and ARM CCA (Tab. 7). The only exception we found was that SEV currently lacks PKS (even though it supports the user-mode protection keys or PKU). However, as the Nested Kernel [51] shows, similar memory protection to PKS can be enforced using private page table mappings and x86 write-protection features at a slightly higher cost.

Paravisor-enhanced CVM deployment. Although there is no current open-sourced hypervisor to support paravisor-enhanced TDX implementation [4], we find that paravisors are designed to shift native (non-CVM) guests into CVMs. Therefore, we deployed Erebor in a native guest to test compatibility, and were successful in executing our system in such a guest. This is expected since all the hardware features required by Erebor are not CVM-specific (unlike VMPL).

Note that we expect one change in Erebor's implementation on real paravisor deployments, specifically related to attestation and secure channel establishment (§6.3). In such a case, the attestation report via tdreport would reflect the paravisor's measurement instead of Erebor-Monitor. Thus, to establish a secure channel with a remote user, Erebor would need to retrieve the attestation report either using a virtual TPM (vTPM) deployed by the paravisor or the TDX runtime measurement registers (RTMRs) [63]. We leave a full paravisor deployment study to future work.

11 Discussion

Xen paravirtualization (PV). Erebor's intra-kernel security monitor concepts (§5) are analogous to Xen's paravirtualization interfaces [29]. Particularly, in Xen PV model, all privileged guest instructions are replaced with paravirtualized equivalents—namely, hypercalls. However, Erebor differs for its sandboxing model (§3.2), which limits the set of instructions managed within the monitor's TCB, rather than replicating the full suite of kernel functionalities (e.g., device and network management). Nonetheless, Erebor's monitor interfaces can be directly derived from PV. We defer the integration of PV interfaces with Erebor to future work.

Unikernel-based approach comparison. An alternative sandboxing approach is to deploy a *trusted Unikernel* as the OS for a CVM—converting a CVM into a dedicated service instance for a single client. Erebor has benefits in the following aspects. First, Erebor has a smaller TCB (e.g., less than 5k LoC versus 57k LoC in Gramine-TDX [63] kernel), as Erebor delegates most functions to the kernel and just validates them, instead of maintaining OS functions. Second, Erebor is cost-efficient in terms of handling concurrent clients with common memory sharing. Under a Unikernel design, each client has to be assigned its own CVM (with fully replicated program common memory), and a single host server may only support limited concurrent CVMs (e.g., 64 on our Xeon 5 server machine specified in §9). One potential avenue is to extend Erebor to support intra-Unikernel isolation [75, 83]. We leave this as future work.

Digital side/covert channel mitigations. Erebor currently does not consider digital processing timing/intervals and micro-architectural covert channels.

To mitigate process timing and input-output interval-based covert channels, Erebor can adopt fixed output intervals based on one-shot request execution, or employ leakage-free quantized communication intervals [60].

Even though CVM has robust side-channel protections against the untrusted hypervisor [17], the CVM kernel shares hardware resources like caches [55–57] and branch predictor units [52, 67] with sandboxes. Thus, untrusted components may leverage micro-architectural side-channels (e.g., cache/page table access pattern) to infer data. Admittedly, an efficient and provable defense against these channels requires hardware modifications [48, 82]. However, software heuristic-based protections can be adopted by Erebor, including core-isolation, rate limiting for sandbox exits, and cache/TLB eviction-enforced exiting [74], as well as sandbox software noise injection/obfuscation [40, 42, 77, 90].

12 Related Work

Intra-privilege kernel separation. Prior systems have leveraged isolation within kernel to implement security monitors. Our security monitor design is inspired by such approaches, particularly by Nooks [84], SVA [50], Nested Kernel [51], and SKEE [45] which also virtualize the MMU interface using instrumentation. Nevertheless, prior monitors are designed for kernel hardening (e.g., kernel code integrity) but not data protection. Implementing the latter requires additional memory enforcement (§6.1) and exit interposition (§6.2). Moreover, our monitor uses efficient hardware-enforced CFI mechanisms, while prior techniques rely on private mappings or expensive compiler techniques (e.g., SAFECode). Besides MMU virtualization, IskiOS [54] and KDPM [64] maintain shadow stacks or protect kernel sensitive data by using MPK, respectively. DOPE [70] leverages PKS to protect kernel objects from data-oriented attacks.

Client data sandbox for SGX enclaves. Ryoan [60] leverages software fault isolation (SFI) (NaCl [94]) to support distributed sandboxes, where different service providers can securely process client data without leaking it. Chancel [41] proposes multi-client SFI, allowing multiple isolated threads in a single sandbox to efficiently share sandbox memory. Erebor requires a fundamentally different solution since (a) it deals with CVMs instead of enclaves and (b) SGX enclaves are not supported within TDX-based CVMs. Moreover, Erebor also avoids compiler-enforced SFI in userspace code, which can incur high overheads during data processing [60].

13 Conclusion

Erebor is a sandbox architecture design for CVMs that enforces full data protection against leakage to untrusted software. It employs a novel security monitor for designed CVM, using intra-privilege isolation and hardware features like Control Enforcement Technology and Protection Keys, with end-to-end sandbox exit interposition and protection. Our evaluation shows that Erebor is capable of executing diverse cloud workloads with 4.5%–13.2% runtime overhead.

Acknowledgments

We thank our shepherd, Pierre Olivier, and all reviewers for their constructive feedback on improving the paper. This research is supported by the National Research Foundation, Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative, the US Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0204, and a gift from Intel. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, or AFOSR.

References

[1] The apache software foundation, "ab - apache http server benchmark tool". https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Branch Target Identification. https://developer.arm.com/documentation/109576/0100/Branch-Target-Identification.

[3] Cambridge analytica and facebook: The scandal and the fallout so far. https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html.

[4] COCONUT-SVSM on KVM: Progress, Plans and Challenges. https://kvm-forum.qemu.org/2024/COCONUT-SVSM_on_KVM__Progress_Plans_and_Chal_KUUSUF7.pdf.

[5] Confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[6] Confidential VMs on Azure. https://techcommunity.microsoft.com/t5/windows-os-platform-blog/confidential-vms-on-azure/ba-p/3836282.

[7] Control Flow Enforcement Technology (CET). https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf.

[8] Control-flow Enforcement Technology (CET) Shadow Stack. https://docs.kernel.org/next/x86/shstk.html.

[9] Datadog: Cloud Monitoring as a Service. https://www.datadoghq.com/.

[10] Datavant | the leader in data logistics for healthcare. https://www.datavant.com/.

[11] General data protection regulation. https://gdpr-info.eu/.

[12] Google AI—making AI helpful for everyone. https://ai.google/.

[13] Guest-host-communication interface (ghci) for intel® trust domain extensions (intel® tdx). https://www.intel.com/content/www/us/en/content-details/726790/guest-host-communication-interface-ghci-for-intel-trust-domain-extensions-intel-tdx.html.

[14] Hardened usercopy. https://lwn.net/Articles/695991/.

[15] Health insurance portability and accountability act of 1996 (hipaa). https://www.cdc.gov/phlp/php/resources/health-insurance-portability-and-accountability-act-of-1996-hipaa.html.

[16] How Azure is ensuring the future of GPUs is confidential. https://azure.microsoft.com/en-us/blog/how-azure-is-ensuring-the-future-of-gpus-is-confidential/.

[17] Intel trust domain extensions (tdx) security review. https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf.

[18] Intel trusted domain extensions. https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf.

[19] Intel® tdx connect architecture specification. https://cdrdv2-public.intel.com/773614/intel-tdx-connect-architecture-specification.pdf.

[20] Kernel self-protection. https://www.kernel.org/doc/html/v5.0/security/self-protection.html.

[21] KVM: PKS Virtualization support. https://lwn.net/Articles/892541/.

[22] Linux svsm (secure vm service module) for secure x86 virtualization in rust. https://github.com/AMDESE/linux-svsm.

[23] Llama.cpp tutorial: A complete guide to efficient llm inference and implementation. https://www.datacamp.com/tutorial/llama-cpp-tutorial.

[24] Memory protection keys for the kernel. https://lwn.net/Articles/826554/.

[25] Meta llama. https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/.

[26] OCR With Google AI. https://cloud.google.com/use-cases/ocr.

[27] Open virtual machine firmware (ovmf) status report. https://www.linux-kvm.org/downloads/lersek/ovmf-whitepaper-c770f8c.txt.

[28] OpenHCL: Evolving Azure's virtualization model. https://lpc.events/event/18/contributions/1862/.

[29] Paravirtualization (pv). https://wiki.xenproject.org/wiki/Paravirtualization_(PV).

[30] Permission indirection and permission overlay extensions. https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions.

[31] petewarden/c_hashmap.

[32] Run LLMs on CPU with Amazon SageMaker Real-time Inference. https://community.aws/content/2eazHYzSfcY9flCGKsuGjpwqq1B/run-llms-on-cpu-with-amazon-sagemaker-real-time-inference.

[33] Shadow stacks for 64-bit arm systems. https://lwn.net/SubscriberLink/940403/c4561635ec6d8881/.

[34] tdx.c. https://elixir.bootlin.com/linux/v6.6/source/arch/x86/coco/tdx/tdx.c#L614.

[35] User Interrupts – A faster way to signal. https://lpc.events/event/11/contributions/985/attachments/756/1417/User_Interrupts_LPC_2021.pdf.

[36] What is image segmentation? https://www.ibm.com/topics/image-segmentation.

[37] Windows 11 Security Book: Powerful security by design. https://www.microsoft.com/content/dam/microsoft/final/en-us/microsoft-brand/documents/MSFT-Windows11-Security-book_Sept2023.pdf.

[38] Yolov5. https://pytorch.org/hub/ultralytics_yolov5/.

[39] 01org. Intel(r) software guard extensions for linux* os (source code). https://github.com/01org/linux-sgx, 2016.

[40] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A Commodity Obfuscation Engine for Intel SGX. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2019.

[41] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: Efficient Multi-client Isolation Under Adversarial Programs. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[42] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious file system for intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[43] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, 2024.

[44] AMD. AMD SEV-SNP: Strengthening SEV with Integrity Protections and More. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[45] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *Network and Distributed System Security Symposium*, 2016.

[46] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[47] Francesca Cerruto, Stefano Cirillo, Domenico Desiato, Simone Michele Gambardella, and Giuseppe Polese. Social network data analysis to highlight privacy threats in sharing data. *Journal of Big Data*, 9, 2022.

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

[49] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual ghost: Protecting applications from Hostile Operating Systems. *ACM SIGARCH Computer Architecture News*, 2014.

[50] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007.

[51] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[52] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[53] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Los Angeles, CA, USA, 2022.

[54] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast Intra-Kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2021.

[55] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[56] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, 2016.

[57] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[58] Xueyuan Han, Thomas Pasquier, Adam Bates, James Mickens, and Margo Seltzer. Unicorn: Runtime provenance-based detector for advanced persistent threats. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020*, 2020.

[59] Alexander Van't Hof and Jason Nieh. BlackBox: A Container Security Monitor for Protecting Containers on Untrusted Operating Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, July 2022.

[60] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[61] David Kaplan. AMD x86 memory encryption technologies. Austin, TX, August 2016. USENIX Association.

[62] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[63] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, 2024.

[64] Hiroki Kuzuno and Toshihiro Yamauchi. Kdpm: Kernel data protection mechanism using a memory protection key. In *Advances in Information and Computer Security*, 2022.

[65] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale graph computation on just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012.

[66] Dayeol Lee, Dongha Jung, Ian T. Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, Aug 2020.

[67] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug 2017.

[68] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[69] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[70] Lukas Maar, Martin Schwarzl, Fabian Rauscher, Daniel Gruss, and Stefan Mangard. Dope: Domain protection enforcement with pks. In *Proceedings of the 39th Annual Computer Security Applications Conference*, 2023.

[71] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2nd International Workshop on HASP*, 2013.

[72] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference (USENIX ATC 96)*, 1996.

[73] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[74] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical Side-Channel attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[75] Pierre Olivier, Antonio Barbalace, and Binoy Ravindran. The Case for Intra-Unikernel Isolation. In *The 10th Workshop on Systems for Post-Moore Architectures*, 2020.

[76] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing Controlled Channels with Self-Paging Enclaves. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, 2020.

[77] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, August 2015.

[78] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. Heckler: Breaking confidential vms with malicious interrupts. *CoRR*, abs/2404.03387, 2024.

[79] B. Schlüter, S. Sridhara, A. Bertschi, and S. Shinde. Wesee: Using malicious #vc interrupts to break amd sev-snp. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[80] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[81] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[82] YongHo Song, Byeongsu Woo, Youngkwang Han, and Brent ByungHoon Kang. Interstellar: Fully partitioned and efficient security monitoring hardware near a processor core for protecting systems against attacks on privileged software. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[83] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.

[84] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[85] 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 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, April 2014.

[86] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[87] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, June 2017.

[88] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[89] Wenhao Wang, Linke Song, Benshan Mei, Shuang Liu, Shijun Zhao, Shoumeng Yan, XiaoFeng Wang, Dan Meng, and Rui Hou. The road to trust: Building enclaves within confidential vms, 2024.

[90] J. Wichelmann, A. Rabich, A. Pätschke, and T. Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[91] David S Wishart, Craig Knox, An Chi Guo, Savita Shrivastava, Murtaza Hassanali, Paul Stothard, Zhan Chang, and Jennifer Woolsey. Drugbank: A comprehensive resource for in silico drug discovery and exploration. *Nucleic acids research*, 34(suppl 1):D668–D672, 2006.

[92] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[93] 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 (Oakland)*, Oakland, CA, May 2009.

[94] 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 *2009 30th IEEE Symposium on Security and Privacy*, 2009.

[95] Chuqi Zhang, Jun Zeng, Yiming Zhang, Adil Ahmad, Fengwei Zhang, Hai Jin, and Zhenkai Liang. The hitchhiker's guide to high-assurance system observability protection with efficient permission switches. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[96] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[97] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

A Artifact Appendix

A.1 Abstract

Erebor's source code is publically available, and its environment can be reproduced. In this appendix, we show the steps to set up Erebor and test its functionalities.

A.2 Description & Requirements

Erebor requires an **Intel physical machine** with some modern hardware features (§A.2.2).

A.2.1 How to access. We maintain an open GitHub repository to enable permanent accessibility. Meanwhile, we maintain a Zenodo repository for the archive.

- GitHub repository: https://github.com/ASTERISC-Release/Erebor.
- Zenodo DOI: 10.5281/zenodo.14943102, URL: https://doi.org/10.5281/zenodo.14943102.

Note: Please always use the GitHub repository to reproduce the environment. Zenodo is just for public archive.

A.2.2 Hardware dependencies. Erebor's host physical machine requires the following hardware dependencies:

- Intel Trust Domain Extensions (TDX). This is optional (not required for personal development).
- Intel Protection Key Supervisor (PKS).
- Intel Control-flow Enforcement Technology (CET).

A.2.3 Software dependencies. While Erebor's execution environment is in the VM, it has the following base requirements for the host physical machine and the guest:

- Ubuntu 22.04/24.04 Linux/KVM (host)
- QEMU version above 7.1.0 (host)
- (Extended) Gramine LibOS (for guest VM)

A.2.4 Benchmarks. We provide two workloads to test Erebor's main functionalities:

- *LMbench system benchmark.* This shows Erebor's capability to support general system events.
- *Helloword demo sandbox program.* This minimal working example shows Erebor's capability to support sandboxed programs and protect data.
- *LLAMA.cpp sandbox program.* This working example shows Erebor's capability to support a real-world scenario—LLM inference.

A.3 Set-up

System settings. Erebor has two settings:

- (**Default**) *Normal VM setting:* In this setting, the guest will run inside a normal VM, with Erebor's security monitor enabled. This setting is to test Erebor's main functionalities, and using a normal VM is sufficient.
- *TDX CVM setting:* In this setting, the guest will run in a CVM, aligned with the paper's system/threat model.

For artifact evaluation/personal development, we provide access only to the **(Default) setting**. As mentioned in the paper, the same code can run in both settings.

Environment setup. Please follow the README file in the GitHub repository (§A.2.1) to set up the environment.

Login into the guest VM. Once setting up the environment, please use the following commands to log in to the guest VM. Both the **username and password of the VM are pks**:

```
eurosys-aec-review@..:~/Erebor/scripts$ pwd
/mnt/sdb/Erebor/scripts # this is the scripts dir
$ sudo ./run-kvm-vm.sh

# Now the terminal will login into the VM session
pks@ubuntu-vm:~$

# Shutdown the VM by typing Ctrl - ]
pks@ubuntu-vm:~$ ^]

# Now returned to the host
eurosys-aec-review@..:~/Erebor/scripts$
```

Type **`ctrl` and `]`** (`^-]`) to shutdown the VM and return to the host terminal. Please shut down and restart the VM if it's finished or crashed (see §A.5).

A.4 Evaluation workflow

A.4.1 Major Claims

- *(C1): Erebor's source code is publically available and its environment can be correctly set up (§A.3).*
- *(C2): Erebor is able to support its functionalities (experiments E1 - E3 below).*

A.4.2 Experiments. We leverage our benchmarks (§A.2.4) for the following evaluations.

Login to the guest VM: Please always use the script/-commands shown in §A.3.

Experiment (E1): LMBench [EST 5-10 mins].

[Preparation]. We already installed the benchmarks within the server's VM disk image, so youo do not need to set up the benchmark again.

[Execution]. Please run in the guest VM. After finishing the scripts, please kill the guest VM.

```
pks@ubuntu-vm:~$ pwd
/home/pks

$ cd microbench/lmbench/
$ ./mmap.sh
$ ./pagefault.sh
$ ./syscall.sh
$ ./signal.sh
$ ./proc.sh

# Finish, shutdown the VM (type ^-])
pks@ubuntu-vm:~$ ^]
```

[Results]. Results will be printed to the terminal.

Experiment (E2): Helloworld sandbox [EST 5-10 mins].

[Preparation]. The Helloworld demo sandbox program is downloaded alongside the LibOS. Please build/install it:

```
pks@ubuntu-vm:~$ pwd
/home/pks

$ cd ENCOS-LIBOS/gramine/CI-Examples/helloworld
$ make clean && make

# Finish, shutdown the VM (type ^-])
pks@ubuntu-vm:~$ ^]
```

For this demo sandbox program, it does not require any input, and it will provide the output data `0x4141..41` (“AA..A”). You can find a detailed description in the file: ENCOS-LIBOS/gramine/CI-examples/helloworld/README.md.

[Execution]. Execute the program:

```
pks@..:...$ pwd
/home/pks/ENCOS-LIBOS/gramine/CI-Examples/helloworld

$ gramine-encos helloworld
```

[Results] Erebor’s monitor will forward the output data. We use an untrusted debugfs file channel to see the plaintext output data:

```
$ sudo cat /sys/kernel/debug/encos-IO-emulate/out
AAAAAAAA

# Finish, shutdown the VM (type ^-])
pks@ubuntu-vm:~$ ^]
```

Experiment (E3): LLAMA.cpp sandbox [EST 10-15 mins].

[Preparation]. The LLAMA.cpp program is also downloaded alongside the LibOS. Please build/install it:

```
pks@ubuntu-vm:~$ pwd
/home/pks

$ cd ENCOS-LIBOS/gramine/CI-Examples/llama
$ ./pre-req.sh
$ make
```

[Execution].

First, we run a native llama.cpp (non-sandboxed environment). Please just execute:

```
pks@ubuntu-vm:...$ pwd
/home/pks/ENCOS-LIBOS/gramine/CI-Examples/llama

pks@ubuntu-vm:...$ ./run-tests-native.sh
```

This script will simply load the local file demo_prompt.txt as the prompt and execute llama.cpp. During the program execution, you will see the inference results (generated text) on the terminal output (i.e., STDOUT).

Then, we run and see how Erebor sandbox would work. Please read and execute this script:

```
pks@ubuntu-vm:...$ ./run-tests-erebor-demo.sh
```

As demo_prompt.txt reflects secret data, it cannot be directly read into the program. Thus, the script puts the prompt into the simulated input communication channel (at /sys/kernel/debug/encos-IO-emulate/in).

[Results] As the program llama.cpp is now executed in Erebor 's confined sandbox environment, output data is not printed onto the terminal, but put into the emulated communication output channel. Fetch the output by:

```
pks@ubuntu-vm:...$ sudo cat
      /sys/kernel/debug/encos-IO-emulate/out
```

(This emulates the monitor returns result data to the client by a network proxy).

A.5 General Notes

1. As a research prototype, Erebor may contain implementation issues. During testing/development, in rare cases, you may encounter VM panic (kernel hangs/panic). Though this is rare, in such cases, please simply destroy the VM:

- By typing **ctrl and]**, i.e., ^-].
- And restart the VM again (§A.3).

2. We are actively updating Erebor. Please refer to our GitHub repository.