Gramine-TDX: A Lightweight OS Kernel for Confidential VMs

Dmitrii Kuvaiskii[*]
Intel Labs
Neubiberg, Germany

Dimitrios Stavrakakis[*] [†]
The University of Edinburgh
Edinburgh, United Kingdom
Technical University of Munich
Munich, Germany

Kailun Qin
Intel Corporation
Shanghai, China
Shanghai Jiao Tong University
Shanghai, China

Cedric Xing
Intel Corporation
Santa Clara, United States

Pramod Bhatotia
Technical University of Munich
Munich, Germany

Mona Vij
Intel Labs
Hilsboro, United States

ABSTRACT

While Confidential Virtual Machines (CVMs) have emerged as a prominent way for hardware-assisted confidential computing, their primary usage is not suitable for small, specialized, security-critical workloads, i.e., legacy VMs with their conventional OS distributions result in a large trusted computing base.

In this paper, we present the Gramine-TDX OS kernel to execute slim, single-purpose, security-first, unmodified Linux workloads with a minimal attack surface. In comparison to a typical Linux kernel, Gramine-TDX's codebase is $\sim 50\times$ less in binary size and has a significantly smaller attack surface, which makes it a perfect match for emerging cloud-native confidential-computing workloads. Our evaluation on 11 workloads indicates that Gramine-TDX has 1-25% average overhead for CPU- and memory-intensive applications. Performance on network- and FS-intensive applications can drop to 6% of the native application's, as Gramine-TDX prioritizes security over optimizations in virtual hardware communication. We build our prototype using Intel®Trust Domain Extensions (TDX).

CCS CONCEPTS

• **Security and privacy → Systems security**; **Trusted computing**; • **Software and its engineering → Operating systems**.

KEYWORDS

Confidential computing; Security; Intel TDX; Gramine; OS

ACM Reference Format:
Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. 2024. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690323

[*]Both authors contributed equally to the paper.
[†]This work was done when interned at Intel.

1 INTRODUCTION

Confidential Computing solutions secure data at all stages of data processing [30]. Confidential Computing relies on Trusted Execution Environment (TEE) technologies, which can be divided into two groups: *(i)* process-based TEEs (e.g., Intel SGX [31, 64]) and *(ii)* virtual machine (VM)-based TEEs (e.g., Intel TDX [44], AMD SEV [22], ARM CCA [60]).

VM-based TEEs, also called Confidential VMs (CVMs), allow to deploy hardware-isolated encrypted VMs, protected against a wide range of insider attacks [78]. The main target for CVMs are legacy feature-rich VMs with monolithic OS kernels. In a nutshell, they protect entire VMs from privileged attackers and allow running workloads in a secure and isolated manner. However, this primary usage of CVMs is not suitable for small, specialized, security-critical applications: legacy VMs with their conventional OS distributions and a plethora of applications, tools, and files result in unnecessarily bloated TEEs with a large attack surface.

The attack surface of CVMs is comprised of several interfaces to interact with the untrusted host. A malicious host can snoop on these interfaces or inject unexpected/erroneous values through them. The CVM attack surface typically consists of *(i)* the shared memory between the CVM and the host, *(ii)* port and memory-mapped I/O, *(iii)* a set of hypercalls and *(iv)* interrupts/exceptions for event notifications [21, 29]. Different TEE technologies may introduce further attack vectors such as untrusted CPUID leaves.

Most VMs in cloud environments use Linux as their OS kernel, which is designed to trust the *hypervisor* on the host platform. However, there is a growing concern of privileged insider attacks [70], where the attacker controls the hypervisor and the rest of the infrastructure. Thus, *hardening* the Linux kernel, i.e., making the kernel secure and stable in the face of a malicious hypervisor and misbehaving virtual hardware, becomes a timely, non-trivial problem.

We highlight the complexity of hardening the Linux kernel by calculating the Trusted Computing Base (TCB) metrics in Table 1. The TCB size closely correlates with the potential vulnerabilities and the system's attack surface [48, 66]. We compare three kernel variants: (a) general-purpose HWE Linux kernel v5.19 shipped in Ubuntu 22.04, (b) Linux kernel v5.19 patched and configured for Intel TDX [11], and (c) Linux kernel v6.1 configured for Amazon Firecracker MicroVMs [6]. Columns 2-6 show different TCB metrics: the size of the uncompressed kernel binary, its code segment size, the number of features enabled in the kernel, the Lines of Code (LoC) that are actually compiled into the Linux kernel, and the

Table 1: TCB Comparison with Linux kernel variants.

Kernel	Binary (MB)	Code (MB)	#features	LoC (K)	#inputs
Ubuntu 22.04 v5.19	68	19.5	2,877	2,048	15,628
Intel TDX v5.19	56	19.5	2,836	2,046	1,098
Firecracker v6.1	27	8.6	789	1,067	911
Gramine-TDX	1.2	0.7	–	57	177

number of code points at which untrusted input is loaded from the host at runtime. For the last metric, we use the smatch static analyser with Intel TDX-specific patterns [10].

As it is evident from Table 1, the Linux kernel is massive and complex. Even a fine-tuned Firecracker version enables 789 features, resulting in a large TCB of 8.6MB of binary code and an attack surface with 911 entry points for untrusted inputs[1]. Importantly, several features provided by the Linux kernel are redundant for many cloud-native confidential-computing workloads and have the adverse effect of increasing their TCB and vulnerable input points.

A recent attack called Heckler highlights the pitfalls of using a bloated legacy OS kernel in CVMs [25]. Heckler relies on the legacy INT 0x80 software interrupt that can be injected into a CVM at any time. Upon receiving the interrupt, Linux mistakenly executes a system call, which may lead to a privilege escalation, allowing the attacker to gain complete control over the CVM. The Linux kernel was patched to fix this vulnerability [83]. However, given that the Heckler attack abused only one out of hundreds of potentially vulnerable inputs, we anticipate similar attacks in the future.

To this end, a dedicated solution is desirable for security-savvy, confined and lean workloads: a **minimal security-first kernel**, designed with confidential computing requirements in mind, with a small and well-defined attack surface, and a tiny set of essential functionalities. This kernel must provide a high level of protection for code and data inside the CVM, making it especially practical for cloud-native workloads and possibly-malicious environments, such as Function-as-a-Service offerings [16] and Edge clouds [3].

In this paper, we present Gramine-TDX: a library OS to execute slim, single-purpose, unmodified, security-first workloads in a CVM with a minimal attack surface and very low TCB. In comparison to the typical Linux kernel, the Gramine-TDX kernel is approximately ∼50x less in binary size and has a significantly smaller attack surface. We implement our prototype using Intel TDX [44], but it can also be adapted to use other VM-based TEEs.

Gramine-TDX is based on the Gramine Library OS (LibOS) [7, 88, 89], and aims to minimize the CVM attack surface. Gramine is a low-TCB modular runtime with a platform-agnostic component, the LibOS, and a set of platform-specific backends. We choose Gramine as the foundation of our prototype because it provides high-level protections transparent to the application, such as an encrypted file system, remote attestation support and an extensive validation of internal kernel state consistency. Extending Gramine with a TDX backend is a non-trivial task: many OS kernel primitives need to be implemented *from scratch*, while all the interfaces with the untrusted host have to additionally be hardened (§5).

Our security analysis shows that Gramine-TDX has the smallest possible attack surface consisting of 5 virtio queues, 3 memory-mapped I/O regions and 1 hardware interrupt[2]. Its TCB contains approximately 57K LoC. Around 40K LoC are reused from Gramine LibOS, and the new TDX backend constitutes the rest 17K LoC. Notably, only 5K LoC are potentially vulnerable and require security analysis and hardening (3K refer to generic VM functionality like PCIe bus probing and 2K are TDX-specific). Our evaluation highlights the applicability of Gramine-TDX. Our set of 11 *unmodified* workloads includes complex and diverse applications. Gramine-TDX achieves the best performance on CPU- and memory-intensive applications that do not require much communication with the virtual hardware, with an average performance overhead below 25%. Performance on network- and FS-intensive workloads is significantly worse; it can be as low as 6% of the native application's. This is mainly because Gramine-TDX concentrates on the security rather than on optimizations of virtual hardware communication. Overall, this paper makes the following contributions:

- We present the design (§4) and implementation (§5) of a minimal, *security-first* OS kernel for CVMs that minimizes the TCB and the attack surface.
- We provide a security analysis of the attack surface exposed by Gramine-TDX and explain the applied mitigations (§6).
- We highlight the efficiency of our *publicly-available* Gramine-TDX prototype through an extensive evaluation incorporating a diverse set of microbenchmarks and real-world applications (§7).

[1]This is an estimation collected using smatch. Some of them may be classified as trusted during manual review, and others may be missing because of false negatives.
[2]Gramine-TDX is not vulnerable to Heckler [25], as it ignores legacy INT 0x80.

2 BACKGROUND

2.1 Confidential Computing

Confidential Computing enables the protection of data in use by performing the computation in an attested [65] hardware-based TEE [30, 78]. A TEE provides integrity and confidentiality guarantees for both code and data running inside of it.

The majority of hardware vendors support TEEs, including Intel SGX, Intel TDX, AMD SEV, and ARM CCA [22, 44, 60, 64]. Intel SGX is the only industry-built process-based TEE, while the rest provide their security guarantees for entire VMs, coining the term *confidential VM* (CVM). Most cloud vendors, such as Microsoft, Google and IBM, currently offer CVMs [85–87]. Along with the hardware offerings, several software frameworks and Software Development Kits (SDKs) have emerged to ease the application deployment in confidential computing settings [24, 88].

2.2 Intel TDX: Architecture and System Model

Intel TDX components. Intel TDX [44] is a technology designed to isolate secure VMs, called Trust Domains (TD), from the hypervisor, other TDs, and any other software on the host platform [21, 28]. It is built using a combination of hardware and software components, namely Secure-Arbitration Mode (SEAM), Total Memory Encryption - Multi Key (TME-MK), and the Intel TDX Module.

SEAM extends the Virtual Machine Extensions (VMX) architecture. It defines two new VMX modes: *SEAM VMX root* and *SEAM VMX non-root*. The former is a restrictive mode where instructions can be fetched only from a special SEAM-memory range; it hosts the Intel TDX Module. The latter is a more permissive mode that adds *Shared Extended Page Tables (EPT)* support; it hosts user TDs.

TME-MK is an encryption engine sitting on the memory bus to encrypt/decrypt the traffic to/from main memory. TME-MK encryption keys can only be used in SEAM and enable the Intel TDX Module to provide per-TD memory encryption.

The *Intel TDX Module* is responsible for creating, measuring, executing and attesting TDs. New instructions are provided to transition a CPU between the VMX modes. The SEAMCALL is used to transition from legacy VMX root mode into SEAM VMX root, and the TDCALL performs the transition from SEAM VMX non-root into SEAM VMX root mode. The hypervisor uses the SEAMCALL to request services from the Intel TDX Module, e.g., adding pages to a TD. The TDCALL is used to request services from within a TD, such as generating an attestation report.

Threat model & runtime protections. Intel TDX threat model considers an active, powerful adversary that can control the entire system software stack, including the OS/hypervisor, and perform physical attacks on the platform (e.g., memory probing, cold boot attacks [97]). Intel TDX does not protect against replay attacks in the same TD. We consider attacks on resources' availability management as well as side-channel attacks as out of scope.

The memory of a TD is confidentiality-protected by using a per-TD encryption key, which is randomly generated by the CPU and is not accessible by the software. The integrity of guest-to-host page mappings is provided by *Secure EPT*, which resides in the private range of the TD memory. The CPU context is also stored in the private range of TD memory when a virtual CPU is interrupted. To enable direct communication with the host, Intel TDX allows the TD to define regions of shared memory in its *Shared EPT*. The shared memory is not protected by Intel TDX. Thus, the TDs must operate on values in shared memory with utmost care.

Attack surface. Software inside a TD interacts with the untrusted host through several interfaces that constitute the attack surface:

- **Shared memory**: used as data-path buffers (e.g., virtio queues) for emulated devices.
- **CPUID leaves & Model Specific Registers (MSRs)**: some CPUID leaves and MSRs are controlled by the hypervisor.
- **Port I/O & Memory-Mapped I/O (MMIO)**: used for control-path commands of emulated devices.
- **Specific hypercalls**: used for hypervisor-specific services, such as requesting a Intel TDX Quote.
- **Interrupts/exceptions**: host injects an interrupt/exception to notify about events (e.g., new data in virtio devices).

Measurement & attestation. At TD creation, the Intel TDX Module initializes the measurement registers. The *MRTD* register contains a hash over the initial state of the TD (i.e., metadata, content of TD memory). The Intel TDX Module also provides runtime measurement registers (*RTMRs*) that can be set during runtime. On top of that, the Intel TDX Module generates a TD-identifying object (TDREPORT), that includes the measurement registers as well as a 64-byte TD-supplied data field. The TDREPORT can be verified and signed by an Intel SGX enclave, the *TD Quoting Enclave*. The signed TDREPORT is called an *Intel TDX Quote* and allows a remote user to gain trust in a TD running on a remote platform [79].

Software. User-space software can be executed inside a TD without modifications. However, there are two kernel-space components that must be Intel TDX-enabled: the OS kernel and the BIOS. There exist two main BIOS implementations: *Intel TDX Virtual Firmware (TDVF)* and *TD-Shim*. The former is a modification of the OVMF project, which enables UEFI support for VMs. The latter is a minimal virtual BIOS designed specifically for Intel TDX.

2.3 OS Kernels for Confidential VMs

Linux kernel has become the norm for virtualization and is the only one that currently supports CVMs [85, 87]. Recent Linux kernels consist of more than 20 million LoC with more than 15,000 configuration options [62]. Therefore, it is impossible to perform an audit or a formal verification of the whole Linux codebase [19]. Moreover, the abundance of features and the scattered vulnerable code parts necessitate a lot of manual effort and specialized tools to achieve a level of confidence in a hardened Linux version [12].

Ongoing research efforts to analyze hardening requirements in Linux [10, 38] are mainly based on three pillars: *(i)* attack surface minimization, *(ii)* code audit and *(iii)* exhaustive fuzzing. To *minimize the attack surface*, the most prominent approach is to manually disable a set of features, such as device drivers and I/O ports. To *audit the Linux code*, static analyzers are employed, which are indecisive since they cannot cover every attack pattern and can result in potential false positive reports. Finally, *fuzzing the Linux kernel* is time-consuming, best-effort and inconclusive [38, 80].

2.4 Gramine Library OS

Gramine-TDX is based on the Gramine Library OS [7, 88, 89]. Gramine allows to run a single Linux application inside a TEE, without source-code modification or recompilation. The deployment model of Gramine is simple: a user takes the original application along with its dependencies, writes a companion configuration file (called the *Gramine manifest*) and runs this bundle inside the TEE environment. Gramine also provides tools for TEE attestation, secret provisioning and transparent file encryption.

Gramine has a modular design. It consists of two tightly interacting components: the LibOS and the backend. The LibOS is platform-agnostic and calls into the backend whenever it needs to perform host-specific operations (e.g., memory management, networking, and filesystem operations). To allow for switching between different backends, Gramine specifies a standard API between the LibOS and the backend.

Importantly, Gramine implements a minimal set of functionalities required for the seamless protected execution of applications. It outsources the non security-critical functionalities to the host. Consequently, Gramine does not have to implement a filesystem or a network stack [43, 72]. Further, Gramine is designed to have a well-defined narrow interface with the host and includes extensive validations of the correctness and consistency of host replies.

3 OVERVIEW

Gramine-TDX is a lightweight library OS (LibOS), designed to run unmodified Linux applications in CVMs using Intel TDX. Figure 1 presents an overview of Gramine-TDX components and its deployment workflow. Precisely, Gramine-TDX builds on Gramine LibOS and implements an Intel TDX backend. Thus, by the virtue of virtualization, it protects the host OS from a malicious CVM, and based on Gramine's design, it protects the CVM from a malicious host OS. This is a core difference with Gramine-SGX which does not protect the host from a possibly malicious SGX enclave. Additionally, Intel TDX offers more HW features (e.g., Intel CET) within the CVM, providing additional defense in depth for the Gramine-TDX kernel. In contrast, such features are forbidden inside SGX enclaves.

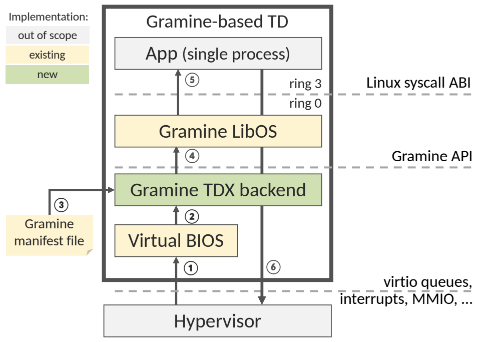


Figure 1: Gramine-TDX is a security-specialized LibOS that executes arbitrary single-process Linux applications. The LibOS component is platform-agnostic, whereas the Intel TDX backend implements hardened interfaces between the TD and the untrusted host hypervisor. The Gramine manifest file holds the security posture of the application.

Further, Gramine-TDX runs entirely in kernel-space (ring 0) while the applications are placed in user-space (ring 3) of the Gramine-based TD, unlike Gramine-SGX that runs both in the same ring-3 address space without memory protections. Importantly, Gramine-TDX outsources as many subsystems (e.g., filesystem, network stack) as possible to the host, which reduces its TCB and, consequently, the attack surface. Lastly, the Gramine manifest file constitutes a backbone of Gramine-TDX, as it includes the configuration for the execution environment of the application (§ 2.4).

Gramine-TDX currently supports only single-process applications. Further, the applications should not require any special hardware features, e.g., GPU access. However, such restrictions are not a barrier for the adoption of Gramine-TDX, as many cloud-native workloads conform to them.

Gramine-TDX deviates from the traditional VM deployment model. First, the hypervisor initiates the TD and the virtual hardware through the Intel TDX-enabled virtual BIOS (1) and the control is transferred to the Gramine TDX backend (2). Then, the backend loads and parses the manifest (3). Based on its content, the Gramine LibOS is bootstrapped (4). Finally, the LibOS proceeds with the execution of the application in the TD user-space (5). At runtime, all requests (system calls) from the application are intercepted by the LibOS, and, if needed, are forwarded through the backend to the hypervisor (6). All requests to the untrusted hypervisor are funneled through a narrow interface of the Intel TDX backend.

3.1 System Model

Threat model. Gramine-TDX inherits the standard Intel TDX threat model [21, 28]. It protects against powerful adversaries that can control the host system and the hypervisor. On top of that, Gramine-TDX comes with tools for file encryption, thus, being able to provide confidentiality and integrity guarantees for separate files. The direct communication with the host through shared memory is beyond our threat model. Additionally, Gramine-TDX does not provide any explicit protection for the network communication, which is assumed to be protected at the application level leveraging cryptographically secure libraries (SSL/TLS) [72]. Gramine-TDX offers tools for Intel TDX remote attestation which ease the authentication and secret provisioning process [79]. Lastly, similar to Linux-based VMs, the console lies outside its protection boundaries. Gramine-TDX considers all intra-TD interactions as trusted, i.e., malicious or exploitable applications are out of scope.

Usage & deployment model. Gramine-TDX is designed to execute generic, security-critical applications in Intel TDX CVMs, while minimizing the attack surface. Its deployment model heavily differentiates itself from the traditional VM deployment. It requires support from the hypervisor, as it offloads its core I/O functionalities to the host (e.g., file system). The essential files for the application execution are loaded from the host and the network communication is achieved directly through the host sockets. In this way, Gramine-TDX eliminates the need for a special VM image and the often tedious handcrafted network configuration. Note that, to ensure integrity and confidentiality, both the stored files and the network traffic need to be encrypted and integrity-protected; the former are transparently encrypted and integrity-protected by Gramine-TDX whereas the latter must be handled by the application itself.

Programming model. Gramine-TDX aims to support all cloud-native applications, i.e., applications that use POSIX and Linux interfaces and do not mandate the use of special hardware primitives (e.g., raw access) or perform admin operations. Currently, Gramine-TDX does not support the `fork` system call or the `posix_spawn` function, thus constraining its usage to single-process applications.

3.2 Design Goals

Security first principle. Gramine-TDX strives to minimize the attack surface of an application running inside a TD. To this end, Gramine-TDX offloads the core functionalities of I/O subsystems (e.g., network, file system) to the host and limits the number of its I/O endpoints (e.g., virtio queues, MSRs). Notably, it includes *minimalistic* implementations of *only* three drivers. Additionally, Gramine-TDX does not provide support for complex resource management or sophisticated scheduling. Gramine-TDX relies on the LibOS that contains a wide set of validations on the correctness of the application state and uses practical knowledge from the Intel SGX backend. Cumulatively, this results in Gramine-TDX having a minimal well-tested TCB and a significantly reduced attack surface (Figure 2) while providing clear security boundaries.

Generality. Gramine-TDX aims to support a wide range of applications and frameworks. Despite its minimal design, Gramine-TDX provides the essential functionalities, in co-operation with the host, to execute complex and diverse cloud-native applications, assuming that they consist of a single process and do not demand specialized hardware access. Undeterred by these limitations, Gramine-TDX can run unmodified applications written in several languages and runtimes, including C, C++, Rust, Python, Java, Go.

Ease of use and deployment. To promote applicability, Gramine-TDX strives to simplify its use and deployment process. To this end, Gramine-TDX adopts a hypervisor-agnostic design. It utilizes a set of APIs that is supported by the most widely-used hypervisors (e.g., QEMU/KVM, Cloud Hypervisor). Further, Gramine-TDX relies on extensively tested, robust host subsystems which are present in all existing infrastructures. Lastly, the only required effort to run an application under Gramine-TDX is to synthesize a manifest file that defines the configuration options to properly setup the execution environment for an application.

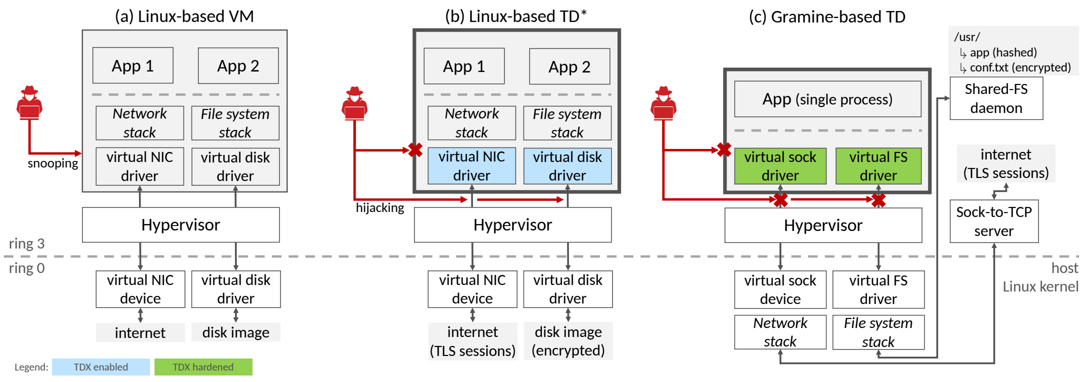


Figure 2: Comparison of deployment models and attack surfaces of classic Linux-based VMs, confidential Linux-based TDs, and our proposed Gramine-based TDs. (a) Linux-based VM is susceptible to snooping and hijacking attacks from a privileged adversary. (b) Linux-based TD is not susceptible to snooping attacks because all TD state and communication is encrypted, however, Linux-based TD is susceptible to hijacking attacks on VM<->hypervisor interfaces. (c) Gramine-based TD is not susceptible to snooping and hijacking attacks because all TD state and communication is encrypted and hardened. Note how Gramine-based TD outsources all non-security-critical stacks to the host platform.
***Current Linux upstream (as of July 2024) does not have comprehensive Intel TDX hardening, though some attack vectors are being addressed [83, 90].**

Trust establishment. GRAMINE-TDX, as a confidential computing system, must provide a way for remote users to establish trust with a running application. To achieve this, GRAMINE-TDX offers mechanisms to generate a chain of trust and the necessary evidence to attest that an application executes the expected software on the expected Intel TDX hardware. Importantly, this evidence, among others, includes measurements of the GRAMINE-TDX backend and the GRAMINE manifest file. Thus, a client can request the Intel TDX attestation evidence and proceed with its verification, before performing any security sensitive action (e.g., secret provisioning).

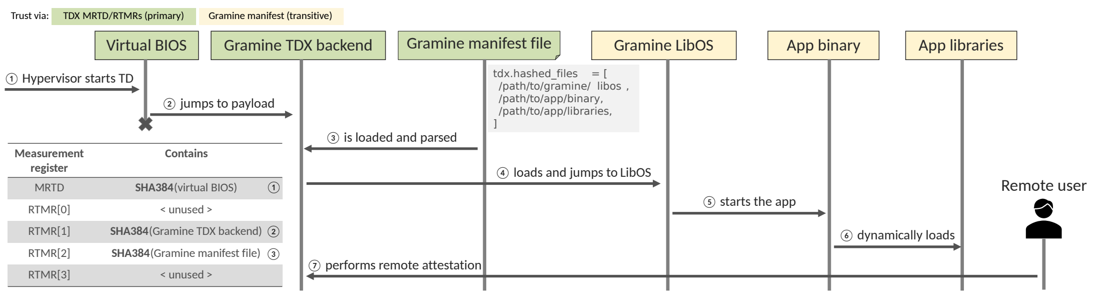
4 DESIGN

We highlight GRAMINE-TDX's design choices by comparing it against traditional Linux-based VMs and TDs (Figure 2). Figure 2a depicts a normal non-confidential VM deployment: a VM runs on top of the host bare-metal system (for simplicity, we assume that the host OS is Linux-based). The hypervisor manages the VM lifecycle and connects virtual drivers in the VM with the corresponding virtual devices on the host; in particular, with a virtual Network Interface Card (NIC) and a virtual hard disk pseudo-device [77]. Arbitrary applications may run inside a VM, supported by the guest OS (guest Linux). The guest Linux is fully-featured and VM-aware. Network I/O requests from the applications go through the network stack and are ultimately forwarded to the host through the virtual NIC driver. Similarly, file I/O requests go through the FS stack and are forwarded to the host via the virtual disk driver. This VM deployment is insecure – all VM memory is stored in RAM in plaintext, and an attacker can snoop on this VM and steal private data.

Figure 2b depicts a typical CVM deployment with Intel TDX: an Intel TDX-enabled Linux kernel runs in a protected VM (TD). An Intel TDX-enabled kernel means that it is aware that it runs inside a TD and changes its drivers (and other subsystems) accordingly, to allow interactions between a TD and the host. In particular, all TD memory is marked as TD-private and is transparently encrypted and integrity-protected by Intel TDX. The only exceptions are related to the drivers, e.g., the virtual NIC driver must put network packets in shared memory so that the host virtual NIC device can consume them, and similarly in the other direction. The same applies to the virtual disk driver. In this deployment, the VM memory is encrypted and the drivers are expected to encrypt network packets/file blocks before putting them into shared memory, so that the attacker cannot snoop any data. However, an attacker is still left with a large attack surface in the form of untrusted inputs (e.g., via CPUID, MSR, MMIO) from the malicious host, which can be used to construct a hijacking attack and subvert the execution of VM applications or leak their private data. To date, the Linux kernel is only beginning to be hardened against such attacks [83, 90].

Figure 2c shows the GRAMINE-TDX deployment: GRAMINE-TDX replaces the guest Linux kernel. GRAMINE-TDX lacks the network and FS stacks. Instead, GRAMINE-TDX uses the corresponding stacks on the host. This up-levelling is achieved through a different set of drivers: a virtual socket (vsock) driver and a virtual FS driver. These drivers work at the level of POSIX sockets I/O and FUSE-based file I/O. They are standardized in VIRTIO specification; thus, the hypervisor/host is guaranteed to implement their corresponding devices [15]. We write these drivers *from scratch* with an Intel TDX-specific security model in mind. Since the host operates on virtual-sockets and virtual-FS primitives, GRAMINE-TDX requires two daemons to run on the host: one shared-FS daemon that translates FUSE-based I/O to normal disk I/O and one network server that translates vsock-based I/O to normal TCP/IP networking. In this deployment, the attacker has no attack surface to exploit as GRAMINE-TDX sanitizes all untrusted inputs and rejects malicious ones.

Small TCB. Minimality of GRAMINE-TDX stems from four factors:
• GRAMINE-TDX does not implement the network stack and the file system stack. Instead, the GRAMINE-TDX kernel implements two paravirtualized device drivers: *virtio-vsock* and *virtio-fs* [15]. The former driver forwards network packets from the guest VM to the host and back via a POSIX-style socket interface, without any packet pre-processing. The latter driver forwards FS requests and data blocks from the guest VM to the host and back via a FUSE-style



```
tdx.hashed_files    = [
  /path/to/gramine/  libos ,
  /path/to/app/binary,
  /path/to/app/libraries,
]
```

Measurement register	Contains	
MRTD	**SHA384**(virtual BIOS)	①
RTMR[0]	< unused >	
RTMR[1]	**SHA384**(Gramine TDX backend)	②
RTMR[2]	**SHA384**(Gramine manifest file)	③
RTMR[3]	< unused >	

Figure 3: Gramine creates a chain of trust to remotely attest that the Gramine TD runs the expected software stack. First Intel TDX hardware measures virtual BIOS in the MRTD register, then virtual BIOS measures the Gramine TDX backend in RTMR[1], and finally Gramine TDX backend measures the manifest file in RTMR[2]. The rest of the software stack has corresponding hash values in the manifest, and is thus transitively reflected in RTMR[2]. Remote user fetches the set of MRTD/RTMR registers during remote attestation and compares against the expected values.

file system interface, also without any FS processing. We implement these two drivers in a minimalistic fashion from scratch.

• Gramine-TDX does not have any legacy or advanced code. There is no support for 16-/32-bit execution. There are no kernel drivers other than the two core virtio drivers, along with a trivial *virtio-console* for I/O on the terminal. There is no support for exotic features, no choice of scheduling algorithms and no architectures other than x86-64. The whole Intel TDX backend for Gramine is written from scratch and intentionally in a simplistic way.

• Gramine-TDX supports only single-process applications. There is no support for UNIX-style fork and, therefore, no logic for resource sharing. In particular, paging is trivial: virtual-to-physical mapping is 1:1 and is set up at boot time. Additionally, there are no Inter-Process Communication (IPC) mechanisms (e.g., IPC signals). Allowing only a single process in a TD is a significant limitation but also a big contributing factor to the simplicity of Gramine-TDX.

• The Intel TDX backend required no modifications to the core Gramine, thanks to Gramine's modularity. Thus, we can reuse the robust Gramine core functionalities (e.g., LibOS) in Gramine-TDX.

Deployment model. *Classic VM deployment* typically requires a VM disk image – a binary blob that represents a Virtual Hard Disk (VHD) for the VM. The disk image contains the files found in a normal OS distribution: configuration files, tools and utilities, pre-installed programs and shared libraries. Next, a VM deployment requires a kernel image – the executable that is copied in the VM memory by the hypervisor and is handed control after VM startup, so that the OS kernel starts to boot. A VM deployment also needs a virtual BIOS that prepares virtual hardware information for the guest kernel, such as the PCI bus configuration and contents of ACPI tables. It further requires several virtual devices, such as a virtual NIC so that the VM can establish network connections. In the end, the classic VM deployment strives to provide a complete set of virtual hardware, such that the kernel and applications running inside a VM are hardly aware that they do not run on bare metal.

Typical TD deployment implies the hypervisor to be instructed to create a TD instead of a VM. Additionally, the guest BIOS and kernel must be Intel TDX-enabled, and the mounted VHD must contain encrypted data (to preserve the confidentiality of its content). In all other aspects, TD deployment mimics classic VM deployment.

Gramine-TDX deployment, in contrast, is aware that it runs inside a TD and relies heavily on the support from the hypervisor. In particular, Gramine-TDX does not use a VHD but directly accesses the host file system. To guarantee confidentiality, the files on the host that are used by Gramine-TDX must be encrypted; Gramine-TDX decrypts them inside the TD before it passes their content to the application. Therefore, Gramine-TDX does not need a prepared VM image. In this deployment, the application and its dependencies (e.g., shared libraries) and other files (e.g., configuration, database, logs) are read directly from the host file system. The host is responsible for obtaining the input files from the remote user/service and sending the output files back to the remote user/service. As for networking, Gramine-TDX does not use a virtual NIC but instead directly communicates with the sockets created on the host. As a consequence, Gramine-TDX requires no network configuration.

Chain of trust and remote attestation. As described in §2.2, a remote user, must gain trust in the remotely executing TD by examining the Intel TDX attestation evidence – an Intel TDX Quote. Only after successful verification of the quote, the user can trust the TD and provide the application with secrets. The user awaits specific reference values in the TD's measurement registers, which are embedded in the received quote. Currently, Gramine-TDX uses a subset of the available Intel TDX measurement registers; these registers build a chain of trust. The user is expected to verify each of the chain items against reference values.

Figure 3 shows how the chain of trust is built. The measurement registers contain hashes of the three components whose integrity is critical during boot: the virtual BIOS, the Gramine-TDX backend binary, and the application-specific Gramine manifest file. There is no need to extend Intel TDX measurement registers with the hashes of subsequently loaded binaries and files, like the Gramine LibOS binary and application files, because the manifest already contains all these hashes. Thus, by gaining trust in the manifest, the user transitively gains trust in all files specified in the manifest.

Out of the five measurement registers, Gramine-TDX uses MRTD, RTMR[1] and RTMR[2]. This choice is dictated by the Intel TDX virtual firmware specification: MRTD reflects the firmware code (the virtual BIOS), RTMR[0] reflects the firmware configuration (our virtual BIOS doesn't allow different configurations), RTMR[1] reflects the OS kernel (Gramine-TDX), RTMR[2] reflects the application (Gramine manifest file has all the information for the application) and RTMR[3] is reserved for special uses.

Gramine-TDX largely reuses remote attestation flows from Gramine-SGX. Particularly, it uses the RA-TLS and Secret Provisioning libraries, initially developed for Gramine-SGX [47].

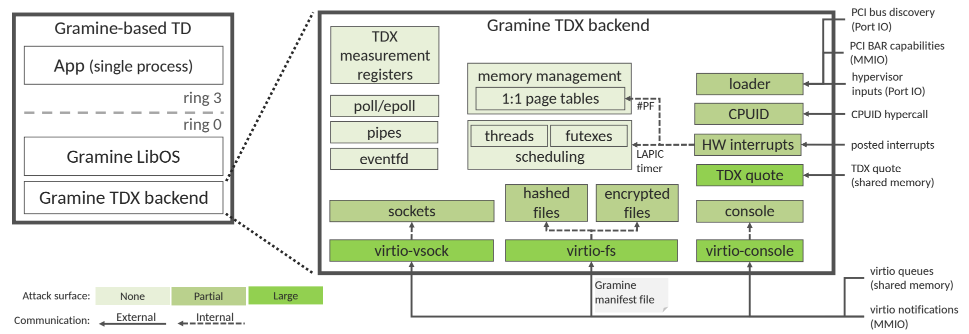


Figure 4: Architecture of Gramine-TDX, in particular of the newly developed Intel TDX backend. The backend implements OS kernel primitives from scratch, in a security-first manner. Primitives that do not have external communication with the host are not susceptible to attacks. Primitives that *do* have such communication are susceptible to attacks and must be hardened. Some primitives have transitive communication and/or are marginally susceptible to attacks and must be lightly hardened.

5 IMPLEMENTATION

Host/hypervisor. We consider the traditional Linux-based hypervisor stack on the host platform. We use the Intel TDX-enabled host Linux kernel v6.8 (with the Intel TDX-enabled KVM kernel module) and the Intel TDX-enabled QEMU hypervisor v8.2. The host stack natively supports the virtual socket (*virtio-vsock*) and virtual FS (*virtio-fs*) devices. To translate FUSE-based file I/O requests from Gramine-TDX into normal host-file I/O, we use the *virtiofsd* tool with a share-all-files configuration [91], which is safe because file accesses are limited by host's permissions on files. To translate vsock-based network I/O requests from Gramine-TDX in TCP/IP packets, we use *socat* with the vsock-to-TCP configuration.

Starting Gramine-TDX implies starting the Intel TDX-protected VM (TD) under QEMU/KVM alongside two daemons (virtiofsd and socat) running in the background. QEMU must forward the TD's requests to these daemons. Additionally, QEMU must read the Gramine manifest file, parse the requested size and number of virtual CPUs and spawn the TD with this setup. Finally, QEMU must forward the specified command-line arguments and host environment variables into the TD, by using the *fw cfg* pseudo-device. We use the classic PCI-based "q35" machine type in QEMU.

We hide all the above actions under a simple Bash script. Therefore, the invocation of Gramine-TDX looks like this:

```
# prepare the manifest file for your app
$ vim helloworld.manifest.template
$ gramine-manifest helloworld.manifest.template helloworld.manifest

# run app in Gramine-TDX
$ gramine-tdx helloworld
Hello, world
```

Guest BIOS. When a VM starts, the virtual CPU jumps to a *reset vector* address that typically contains the BIOS code. The virtual BIOS is responsible for loading the OS kernel binary (in our case, the Intel TDX backend binary of Gramine-TDX), preparing the memory-map and ACPI tables for the kernel, detecting all virtual CPUs, performing the initial code/data measurement into MRTD/RTMR registers, and passing control to the OS kernel.

There are two virtual BIOS implementations for Intel TDX: *TDVF* and *TD-Shim*. In Gramine-TDX we choose TD-Shim because it is security-oriented and has a minimalistic philosophy (§2.2). The main benefit of using TD-Shim, compared to writing our own BIOS, is that it fills the MRTD/RTMR registers on startup. Precisely, TD-Shim measures itself in the MRTD register and the Gramine TDX backend in the RTMR[1] register (see Figure 3). This leaves Gramine-TDX with only the RTMR[2] register to fill.

Guest kernel. Gramine-TDX constitutes a new Intel TDX backend for Gramine, written completely *from scratch*. Thanks to the modular architecture of Gramine, we are able to use its LibOS component, the encrypted- and hashed-files format, the RA-TLS and Secret Provisioning libraries, the manifest syntax, and several Gramine tools without modifications. The new Intel TDX backend consists of 17K LoC and is written entirely in the C99 language. Out of these 17K, 15K LoC are generic VM functionality (e.g., context switching, interrupt handling, PCIe bus probing), and 2K LoC are TDX-specific (TDX hypercalls, #VE handling). Given that the LibOS component has 40K LoC and the newly written Intel TDX backend has 17K LoC, the ratio of reused code in Gramine-TDX is ~70%. If we additionally take into account all the reused libraries and tools, then the ratio of reused code reaches ~90%. The only third-party dependencies are mbedTLS crypto and the tomlc99 parser. Both these dependencies are statically linked into Gramine-TDX. Aside those, Gramine-TDX is self-contained and does not use libc. This benefits no-glibc workloads (e.g., Go apps), which have significant performance penalties on Gramine-SGX.

Figure 4 shows the subsystems that our backend implements. They include the traditional OS kernel primitives such as memory management, thread scheduling and futex-based synchronization, implementations of pipes, eventfd objects, sockets and files, poll/epoll mechanisms. Precisely, we use 1:1 virtual-to-physical pages' mapping, simplistic round-robin Single Queue Multiprocessor Scheduler (SQMS), a minimal PCI bus discovery mechanism and a limited ELF loader to load the LibOS binary. Gramine-TDX also contains Intel TDX-specific subsystems, such as extending measurement registers and fetching the Intel TDX quote.

Our three virtio drivers are implemented intentionally in a simplistic manner. We consider the virtio-fs driver as not performance critical. We implement it with a single virtio queue, under a global lock, in a synchronous "one request at a time" manner, using busy polling. Similarly, we consider the virtio-console driver as non-critical for performance. Therefore, we do not apply any optimizations to its implementation. Our virtio-vsock driver is more sophisticated: we implement optimized receive and transmit queues with notification suppression, two separate locks, and a helper thread to operate on the receive queue and cleanup the transmit queue,

Table 2: Interfaces with untrusted host.

Attack surface	# at boot time	# at runtime
Virtio queues	0	5
HW interrupts	0	1
MMIO regions	10	3
Port I/O ports	2	0
MSRs	0	0
CPUID leaves	0	0
Intel TDX quote region	1	0

analogous to the Linux threaded interrupt handler [76, 77]. Our virtio-vsock implementation is still rather slow, e.g., credit-based flow control is only minimally implemented.

Gramine-TDX maps the IRQs (interrupt requests) of our three drivers to a single interrupt vector number and only on one vCPU. The first simplification is required to side-step the need to find and parse the untrusted DSDT ACPI table, at a price of additional MMIO accesses for each interrupt. The second simplification is required to avoid the locking during interrupt processing, as only one CPU accesses interrupt-related data structures. Interrupt handling is divided in the top half and bottom half, similar to other OS kernels. This implementation is a performance bottleneck in I/O-heavy workloads, but is easy to reason about in security reviews.

Gramine-TDX carves out several memory regions at boot, with predefined ranges. This hard-coding of the memory layout eases out-of-bounds checks and page protections.

Multi-core support is implemented in a classic way: at boot time, the first (BSP) virtual CPU initiates the Intel TDX-specific MP Wakeup Mailbox protocol to wake up the other (AP) virtual CPUs. At runtime, each vCPU periodically receives timer interrupts and invokes the scheduler. All shared state between vCPUs is protected by coarse-grained locks.

We implement PCI device discovery instead of a simpler MMIO device discovery due to its ubiquity in hypervisors, e.g., Cloud Hypervisor supports only the PCI transport. Gramine-TDX expects all virtio devices to be reported on PCI bus 0 (i.e., no PCI bridges). The three supported virtio devices can be configured only in one way, in particular, to use *split* virtio queues [15].

Gramine-TDX supports multi-threaded applications but not multi-process applications. For example, embarrassingly parallel workloads like PyTorch and TensorFlow are supported, but child-spawning workloads like Bash are not. At the implementation level this means that Gramine-TDX does not support `fork` and `vfork` system calls and does not support the `clone` system call without the `CLONE_VM` flag. It should be noted that Gramine-TDX supports `execve` as this system call does not create a new process.

6 SECURITY ANALYSIS

Gramine-TDX must guarantee the following properties: *(i)* the code of the application and Gramine-TDX must be integrity-protected, *(ii)* the data of the application and the internal state of Gramine-TDX must be confidentiality- and integrity-protected, *(iii)* network I/O must be confidentiality- and integrity-protected, *(iv)* file I/O must be integrity-protected for hashed files and confidentiality-protected for encrypted files, *(v)* Gramine-TDX must not be vulnerable to privileged attacks, *(vi)* tampering with the Gramine manifest file must be detectable, *(vii)* the chain of trust must reflect all loaded components and be immutable.

Statistics on attack surface. A core security argument in favor of Gramine-TDX is that it has a small TCB, as shown in Table 1. Additionally, Gramine-TDX does not have a variety of features that can be disabled at build time; it has only one configuration. This design choice reduces the risk of having a specific subset of enabled features that is rarely tested. Finally, Gramine-TDX has a small number of entry points through which untrusted inputs from a malicious host can be propagated into the TD.

Table 2 shows statistics on the attack surface of Gramine-TDX. As attack surface, we consider the number of interfaces with the untrusted host (hypervisor). We do *not* count the interfaces that are write-only by Gramine-TDX– such cases are benign because the host may ignore the writes, leading to DoS. Only reads are important, as the host can inject malicious values. We further distinguish between interfaces used at boot time and at runtime; we harden both, but the main focus lies on the runtime interfaces. These interfaces are more important because they are used after Intel TDX remote attestation and it is easier to mount attacks on an already-running application. Unfortunately, collecting similar statistics on the Linux kernel was infeasible: one would need to analyze the whole code base of Linux, starting from the vulnerable input points and reconstructing the call chains to learn which components must ultimately be hardened. As shown in §1, even a fine-tuned Linux kernel has over 1 million lines of code and over 900 input points.

Rollback attacks. Rollback attacks are a general issue for Intel TDX, which Gramine-TDX does *not* mitigate. There is no trusted *absolute* time in Intel TDX. However, Intel TDX guarantees secure *relative* time; timeouts never trigger prematurely. There are proposals for secure clocks that can be incorporated in Gramine-TDX [34].

Gramine-TDX uses encrypted files that are vulnerable to rollback attacks between invocations, similarly to Linux. It partially mitigates this by maintaining freshness metadata during application lifetime, but a complete defense would require an independent trusted metadata store (e.g., monotonic counter) [23, 63].

Applied mitigations. Table 3 shows in detail how Gramine-TDX achieves its integrity and confidentiality properties, by listing all attack vectors and explaining the applied mitigations. The descriptions below explain each mitigation mentioned in the table. Note that we do not aim to highlight novel defenses but instead we strive to provide a *systematic study* of possible attacks on VM-based TEEs and their mitigations using Gramine-TDX.

1. Encrypt data. For network I/O, it is the responsibility of the application to use TLS connections or similar secure protocols. For file system I/O, the application developer must mark sensitive files as "encrypted" in the Gramine manifest; Gramine automatically encrypts/decrypts these files using its crypto FS protocol [43].

2. Do not expose data. For console I/O, the application is responsible to minimize and not expose confidential data on the terminal. Additionally, Gramine sanitizes its own log to avoid the leakage of any sensitive information (e.g., addresses of loaded binaries). Finally, the application developer must specify all files that will be accessed by Gramine explicitly; Gramine refuses access to unspecified files, which decreases the risk of accidental exposure of data from FS I/O.

3. Disable inputs. Applications modify their behaviors based on provided command-line arguments and environment variables. This flexibility can be abused by the attacker, e.g., the attacker may provide a `verbose` argument and leak private data. Gramine manifest by default disables all command-line arguments and environment variables coming from the untrusted host.

Table 3: Attack surface exposed by Gramine-TDX, with examples of attacks and corresponding mitigation measures.

Attack vector	Attack methods	Mitigations
Shared virtio queues: control metadata fields and data buffers	Writes: leak data via eavesdropping on contents of data buffers.	§6.1 Encrypt data. §6.2 Do not expose data.
	Reads: inject malicious data; subvert control flow via malformed metadata or malicious data; leak data via attacker-controlled pointer / offset.	§6.5 Verify inputs against known hashes. §6.6 Minimize number of untrusted inputs. §6.8 Separate into read-only and write-only. §6.9 Read once and cache. §6.10 Check against erroneous values. §6.11 Cross-check against known values. §6.12 Check adherence to protocol.
Model-Specific Registers (MSRs)	Reads: subvert control flow via maliciously crafted values in MSRs.	§6.7 Remove need for untrusted inputs. §6.13 Use trusted HW primitives.
CPUID leaves	Reads: subvert control flow / weaken security-critical CPU features via maliciously crafted values in CPUID leaves.	§6.7 Remove need for untrusted inputs. §6.13 Use trusted HW primitives.
ACPI tables	Reads: subvert control flow / weaken security-critical CPU features via maliciously crafted values in host-provided ACPI tables.	§6.7 Remove need for untrusted inputs.
Port I/O: PCI bus discovery, reading VMM inputs	Reads: subvert control flow via maliciously crafted values during port I/O reads; inject malicious data.	§6.3 Disable inputs. §6.4 Hard-code / allow-list inputs. §6.6 Minimize number of untrusted inputs. §6.9 Read once and cache. §6.10 Check against erroneous values. §6.12 Check adherence to protocol.
Memory Mapped I/O (MMIO): PCI bus discovery, virtio drivers	Reads: subvert control flow / weaken security-critical virtio device features via maliciously crafted values during MMIO reads.	§6.4 Hard-code / allow-list inputs. §6.6 Minimize number of untrusted inputs. §6.9 Read once and cache. §6.10 Check against erroneous values. §6.11 Cross-check against known values. §6.12 Check adherence to protocol.
HW interrupts	Inject malicious interrupts to confuse interrupt-handling code.	§6.6 Minimize number of untrusted inputs. §6.13 Use trusted HW primitives.
Crypto primitives	Force Gramine to use insecure source of entropy.	§6.13 Use trusted HW primitives.
Shared buffer for Intel TDX quote	Reads/writes: subvert control flow via malformed metadata or malicious data; leak data via attacker-controlled pointer / offset / length.	§6.10 Check against erroneous values. §6.12 Check adherence to protocol.
Reading Gramine manifest	Reads: weaken security-critical Gramine features via maliciously crafted values in manifest file; defy the chain of trust.	§6.5 Verify inputs against known hashes. §6.12 Check adherence to protocol. §6.14 Perform remote attestation.

4. Hard-code / allow-list inputs. It is frequently unrealistic to disable all command-line arguments and environment variables. Gramine manifest provides a syntax to hard-code a specific set of arguments and envvars. Also, Gramine-TDX hard-codes a single possible secure PCI configuration with which the TD is supposed to be started. If the host provides any different PCI configuration than the expected one, Gramine-TDX fails loudly.

5. Verify inputs against known hashes. Gramine manifest requires to specify a set of "hashed" files. The hashes of these files are calculated and placed in the manifest at build time. Gramine verifies all opened files in this set against their reference hashes.

6. Minimize the number of untrusted inputs. Gramine-TDX uses a bare minimum number of virtio queues for the three supported devices: (a) two queues for virtio-console: receive and transmit, (b) two queues for virtio-vsock: receive and transmit, and (c) one queue for virtio-fs: request [15]. The optional queues are not used; e.g., the "control receive" queue for virtio-console, the event queue for virtio-vsock and the notification queue for virtio-fs. Gramine-TDX also minimizes the number of shared-with-host metadata/control fields of the virtio queues. Gramine-TDX uses a limited number of untrusted inputs at runtime: no MSRs, no CPUID leaves, no port I/O, 3 MMIO addresses and 1 interrupt vector.

7. Remove need for untrusted inputs. Gramine-TDX strives to remove the need for untrusted inputs. As mentioned above, there are no untrusted MSR accesses and no untrusted CPUID invocations during boot or runtime. Also, there is no port I/O during runtime. Finally, Gramine-TDX does not use host-provided ACPI tables; the only table used is MADT (Multiple APIC Description Table) which is synthesized from trusted inputs by TD-Shim.

8. Separate into read-only and write-only. Gramine-TDX never uses read-write regions of shared memory and never uses read-write MSRs. To enforce this, Gramine-TDX supports only the *split virtio queue* format, which separates the queue into several parts, where each part is either write-only or read-only by the driver [15].

9. Read once and cache. Gramine-TDX reads as much information as possible at boot time and caches it to be used later at runtime. This information includes the PCI configuration and reading command-line arguments and environment variables (if needed) from the host. Importantly, control fields and notification statuses of virtio queues are read once per event and cached; this prevents Time-of-Check-to-Time-of-Use (TOCTOU) attacks on the queues.

10. Check against erroneous values. Gramine-TDX checks all possibly malicious inputs against (sets of) expected values or reasonable value limits. Precisely, all input pointers and offsets are checked for overflows, NULL pointers, pointers to private memory, off-by-one errors, etc. This is important for shared buffers in virtio queues, for the Intel TDX quote shared buffer, and for free-formed strings with command-line arguments and environment variables.

11. Cross-check against known values. Gramine's LibOS component maintains the state of the application's requests (e.g., which files are opened, which network connections are in which state). The LibOS actively cross-checks the soundness of its state, and fails loudly if the state is detected to be internally inconsistent. As a particular case in the Intel TDX backend, Gramine-TDX cross-checks

Table 4: Benchmarking variants

Variant	Execution environment
Native	Bare-metal instance
Normal VM	Standard VM with Linux kernel
Intel TDX VM	Intel TDX VM with Linux kernel
Gramine-SGX	Intel SGX enclave with Gramine LibOS
Gramine-VM	Standard VM with Gramine-TDX's kernel
Gramine-TDX	Intel TDX VM with Gramine-TDX's kernel

that the posted interrupt on a virtio device indeed signals that there is data to be read/written on the corresponding virtio queue.

12. Check adherence to protocol. Gramine-TDX adheres strictly to the following protocols: (a) PCI bus discovery and PCI device querying [17], (b) virtio queue flows specification [15], (c) TOML language specification, and (d) Intel TDX quote retrieval protocol [44]. If the actions of the host start to deviate from the protocols, Gramine-TDX considers it an attempted attack and loudly fails.

13. Use trusted HW primitives. Gramine-TDX uses the RDRAND instruction as the sole source of entropy; RDRAND is guaranteed to be secure inside a TD. Gramine-TDX strives to use only TD-trusted CPUIDs and MSRs. In particular, the trusted CPUID table is securely maintained by the TDX firmware, and Gramine-TDX uses only secure CPUID leaves for its internal operations. For other, non-secured leaves, Gramine-TDX intercepts the #VE exception (in TDX, CPUID instructions result in #VE), and synthesizes its value from the information in secure CPUID leaves. MSRs are treated similarly in TDX. Finally, Gramine-TDX relies on the fact that the first 30 interrupt vectors are securely triggered inside a TD [44], and allows only two HW interrupts: INT 32 for the LAPIC timer and INT 64 for all virtio device notifications. The former one is benign (only DoS possible), while the latter is verified by double-checking against the control fields of the corresponding virtio queues.

14. Perform remote attestation. As explained in §4, the remote user must perform Intel TDX remote attestation to gain trust in the remotely executing Gramine-TDX. During remote attestation, an Intel TDX quote with MRTD/RTMR measurements is sent. It reflects the entire software stack of Gramine-TDX and the application running on top of it [79]. Therefore, a malformed Gramine manifest or a maliciously replaced Gramine binary can be detected since RTMRs will contain unexpected measurements.

7 EVALUATION

We highlight Gramine-TDX's applicability and measure its performance overheads through a set of system microbenchmarks and a series of case studies covering 11 *unmodified* applications, accompanied by their *Gramine manifest.*

7.1 Experimental Setup

Testbed. We conduct our experiments on a dual-socket Intel TDX-enabled server, equipped with Intel Xeon Platinum 8570 CPU with 56 cores and 1 TB (16 channels × 64 GB/DIMM) DRAM running Ubuntu 24.04 with an Intel TDX-enabled Linux kernel v6.8. For our experiments, we use an Intel TDX-patched QEMU v8.2. Our VMs run Ubuntu 24.04 with an Intel TDX-enabled guest Linux v6.8. We use Intel TDX Module v1.5 (build_num 698). In our experiments, we provide the VMs with vCPUs equal to the number of application threads. The reported values are the average of 3 runs.

Variants. We conduct the experiments with the variants of Table 4. The Native variant refers to an application running as bare-metal

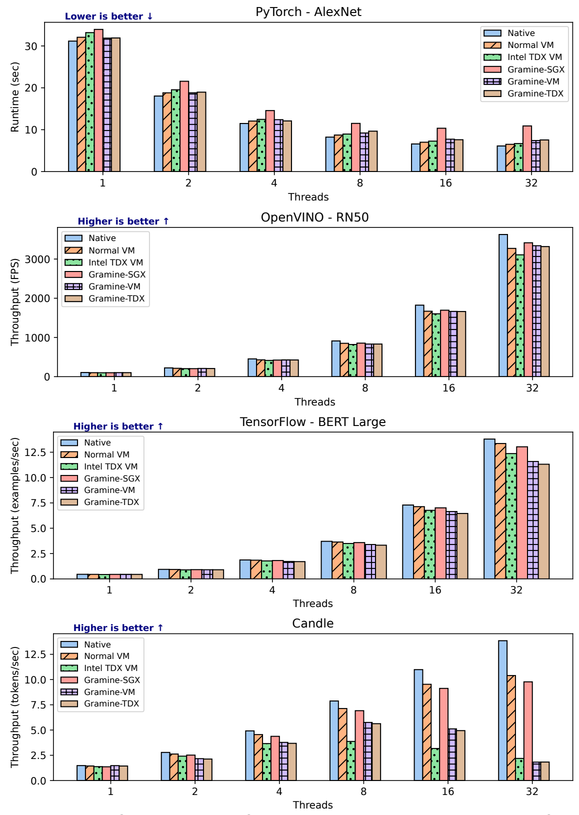


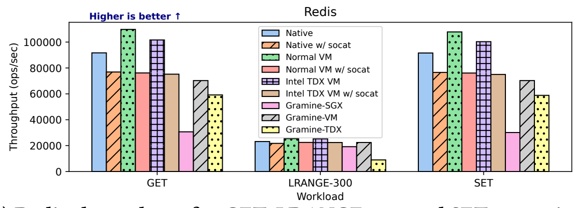
Figure 5: Performance of Gramine-TDX in AI/ML frameworks using different number of threads.

instance directly on our testbed server. The Normal VM and Intel TDX VM variants denote that an application is executed inside a virtual machine having Intel TDX disabled and enabled, respectively. In Gramine-SGX variant, the application is running inside an Intel SGX enclave using Gramine LibOS. Lastly, the Gramine-VM and Gramine-TDX variants indicate the execution of the application inside a virtual machine using the minimal Gramine-TDX's guest kernel with Intel TDX disabled and enabled, respectively.

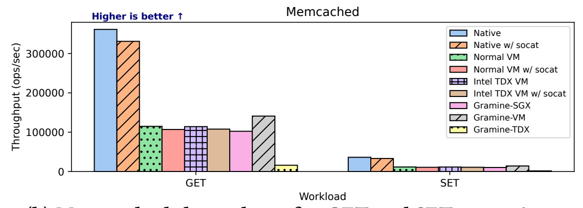
7.2 Microbenchmarks

Workloads. We evaluate the performance of Gramine-TDX using `UnixBench`, a benchmark to assess the performance of a Unix-like OS. It provides tests that examine different aspects of system performance, e.g., CPU, memory, file I/O. We execute all the benchmarks that Gramine-TDX supports, i.e., those that do not mandate spawning a new process. We run single-CPU workloads and use the default parameters of `UnixBench`.

Results. Table 5 presents the average scores of the examined variants in the exercised benchmarks. The results indicate that Gramine-TDX performs well in in-memory computations (e.g., arithmetic benchmarks) with minimal overheads (<5%). However, there is a significant overhead for I/O operations, seen in the file copy benchmarks, where Gramine-TDX achieves only 5-6% of the native performance, mostly due to Gramine-TDX's simplistic virtio-fs driver. Additionally, there is considerable slowdown for system calls, with around 35% overhead compared to the baseline, attributed to quirks in Gramine's context switch implementation. Lastly,



(a) Redis throughput for GET, LRANGE-300 and SET operations.



(b) Memcached throughput for GET and SET operations.

Figure 6: Throughput of in-memory databases (Redis, memcached) using Gramine-TDX

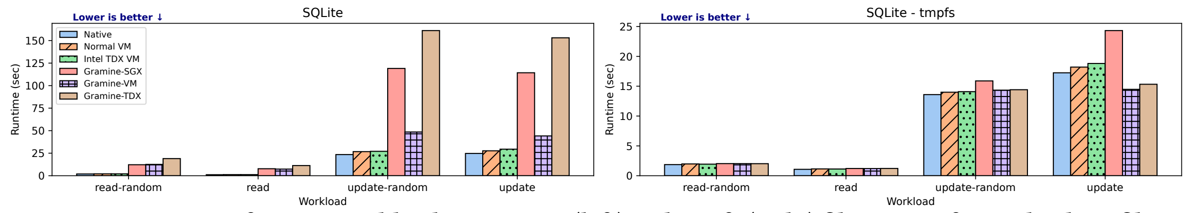


Figure 7: Runtime of SQLite workloads using ext4 (left) and tmpfs (right) filesystems for its database files.

Table 5: `UnixBench` **benchmarks scores. Values in parentheses identify the performance ratio with native as the baseline.**

Benchmark	Native	Gramine		
		SGX	VM	TDX
Dhrystone 2	332M	331M (0.99x)	336M (1.01x)	332M (1.00x)
Whetstone	5023	5006 (0.99x)	5022 (0.99x)	4987 (0.99x)
Copy 256, 500 maxblocks	185K	15K (0.08x)	13K (0.07x)	10K (0.05x)
Copy 1K, 2K maxblocks	698K	59K (0.08x)	51K (0.07x)	37K (0.05x)
Copy 4K, 8K maxblocks	2214K	228K (0.10x)	187K (0.08x)	137K (0.06x)
System Call Overhead	14.2M	12.7M (0.90x)	9.5M (0.67x)	9.3M (0.65x)
Pipe Throughput	1.57M	0.33M (0.02x)	1.71M (1.09x)	1.62M (1.03x)
Execl Throughput	79K	–	4K (0.05x)	0.8K (0.01x)
Recursion Test (hanoi)	8.2M	7.7M (0.94x)	8.1M (0.99x)	8.1M (0.99x)
Arithmetic Tests	782M	841M (1.08x)	816M (1.04x)	798M (1.02x)

pipe throughput is notably lower in Gramine-SGX due to the need for encryption of reused host-kernel pipes, whereas the VM/TDX variants avoid this by keeping pipes within the VM, eliminating the need for VM exits and encryption. Note that execl benchmark failed on Gramine-SGX due to a yet-to-be-analyzed bug.

7.3 AI/ML Frameworks

Workloads. We evaluate the performance for three widely-used AI/ML frameworks, namely PyTorch, OpenVINO, TensorFlow and candle, a minimalist ML framework for Rust. For PyTorch, we use an AlexNet pre-trained model for image classification. Our workload opens an image from a file and runs a classifier. For OpenVINO and TensorFlow, we use ResNet-50 (RN50) convolutional neural network and Bidirectional Encoder Representations from Transformers (BERT Large) models respectively, and perform the inference process on public datasets. For candle, we use the quantized version of the LLaMA model with a sample length of 200.

Results. Figure 5 presents the performance of Gramine-TDX in the described AI/ML frameworks. Precisely, the subplot at the top showcases the execution time of PyTorch image classification for every variant. The second subplot highlights the performance of OpenVINO for the different models across the execution environments of Table 4. The third subplot demonstrates the throughput of TensorFlow for our examined variants. Lastly, the subplot at the bottom indicates the achieved throughput of candle.

We observe that in most cases for all the frameworks, Gramine-TDX achieves performance close to the native execution. The average performance overhead is below 10%. Additionally, Figure 5 indicates that executing an application with Gramine-TDX does not hamper scalability; the scaling factor remains comparable to the baseline. However, we noted that the performance overhead slightly increases in the case of PyTorch and candle with the increased number of threads, where it reaches up to 24% and 87%, respectively. Nonetheless, specifically for candle, Gramine-TDX performs on par (or better) with the Intel TDX VM. There's an interesting issue with Linux-TDX on candle, as its performance is surprisingly low on 32 threads (under investigation).

7.4 Databases

Workloads. We conduct experiments using two in-memory databases (Redis, memcached) and two variants of a SQL database engine (SQLite) operating on different filesystems (ext4, tmpfs). For Redis and memcached, we use the redis and memtier benchmarks, respectively. We place the server in the execution environments described in Table 4 and run the redis-benchmark and memtier binary with their default settings natively (on the same machine). The "*w/ socat*" suffix in the labels of these experiments indicates the use of socat as a proxy layer for the network traffic, which is mandatory for some variants (e.g., Gramine-TDX) as explained in §5. In SQLite, we use the kvtest benchmark. Initially, we load a database with 500k blobs ranging from 2KB to 6KB with an average size of 4KB. Then, we execute the workloads: *(i)* random reads, *(ii)* sequential reads, *(iii)* random updates and *(iv)* sequential updates.

Redis. Figure 6a shows the throughput of Redis server for three operations (GET, LRANGE-300, SET). The overhead of Gramine-TDX, with the native execution as baseline, ranges from 35% to 61%. While this is significant, we observe that a big contributing factor is the use of socat. Importantly, socat introduces up to 26% overhead even in the native variant of the application. This fact justifies that Gramine-TDX performs comparably to a full-fledged VM (with and without Intel TDX enabled) when socat is used, as it can be seen from Figure 6a. Notably, the VM variants outperform the native execution. This is likely due to the VM's use of the virtio-net driver, and potential differences in host and guest kernel implementations or QEMU/KVM optimizations that affect Redis performance.

Memcached. Figure 6b highlights the throughput of Memcached server. Similarly to Redis, Gramine-TDX introduces significant

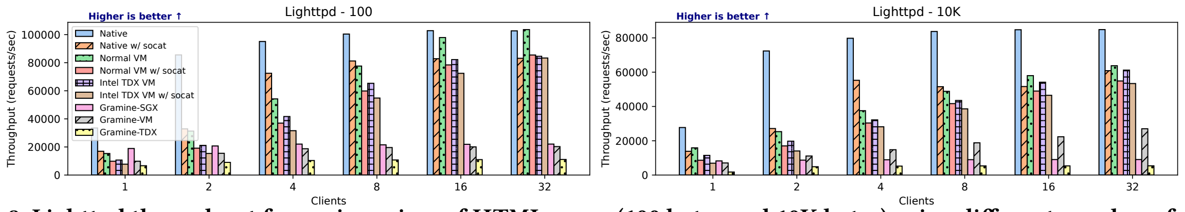


Figure 8: Lighttpd throughput for various sizes of HTML pages (100 bytes and 10K bytes) using different number of clients.

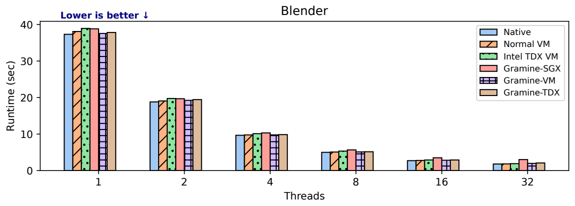


Figure 9: Runtime of Blender application.

Table 6: Runtime (seconds) of image processing applications.

Variant	Go application	Java application
Native	2.89	3.73
Normal VM	2.96	3.50
Intel TDX VM	3.16	3.62
Gramine-SGX	4.26	4.17
Gramine-VM	2.84	3.81
Gramine-TDX	2.95	4.06

Table 7: Boot time (seconds).

Variant	VCPUs	1GB	4GB	16GB	64GB
Normal VM	4	5.2802	5.2966	5.2802	5.6855
Intel TDX VM	4	6.7030	6.8027	7.2663	7.8827
Gramine-TDX	4	2.7993	3.7043	7.2426	17.6503
Normal VM	16	5.6209	5.6341	5.6412	5.8678
Intel TDX VM	16	7.1062	7.2943	7.5145	8.5848
Gramine-TDX	16	2.9773	3.5707	6.0349	11.7705

performance overheads, achieving ~5% of the throughput of the native execution. Interestingly, for Memcached, VM-based variants fare significantly worse as for Redis. Upon further investigation, we established that this is due to VMs with Memcached saturating the CPUs even in the default configuration. We attribute it to the fact that Memcached is multi-threaded (with 4 threads by default), which leads to additional CPU cycles for thread synchronization.

SQLite. Figure 7 presents the runtime of SQLite operating on an ext4 filesystem and also shows the SQLite runtime when the files are stored in tmpfs. When using the ext4 filesystem, Gramine-TDX's runtime is 9.78 – 10.03× higher than the bare-metal SQLite execution for the read and 6.19 – 6.86× for the update workloads. This overhead mainly stems from the expensive file I/O that is performed through Gramine-TDX's virtio-fs driver. When SQLite is tuned to use tmpfs and places its files in memory, avoiding the file I/O, the overheads of Gramine-TDX are diminished below 14%.

7.5 Web-server

Workload. We execute a Lighttpd web-server application. We use wrk, an HTTP benchmarking tool. We run our web-server with the variants of Table 4. Similar to the databases experiments, the "*w/ socat*" suffix denotes the use of socat as a proxy for the network communication. We set wrk to generate workloads that fetch a single HTML page using different numbers of clients. Each experiment runs for 30 seconds. The clients use a single thread and are located on the same machine as the server. We conduct our experiments with two HTML page sizes (100B, 10KB).

lighttpd. Figure 8 presents the throughput of lighttpd using a different number of clients and HTML page sizes. Overall, we observe that Gramine-TDX's throughput is 11 – 20% of the bare-metal web server's throughput when using the 100B pages. The respective numbers for the 10K pages are 6 – 7%. While this is a significant performance drop, we believe that fine-tuning our naive virtio-vsock driver implementation could significantly improve performance; in particular, we plan to improve batching and the sizes of transmit/receive queues and network packets.

7.6 3D Rendering

Workload. We measure the performance of Gramine-TDX using the Blender 3D framework. We execute the Blender application with one scene as input. The application performs the scene rendering using the PNG format with different thread counts.

Blender. We present the runtime of our Blender benchmark in Figure 9. Gramine-TDX does not introduce any significant overheads for the Blender application (< 7%) in most cases. This is due to the fact that the rendering is performed entirely in memory and does not require any expensive VM exits. On top of that, Figure 9 further highlights Gramine-TDX's scalability, by observing the decrease of the execution time while increasing the number of worker threads.

7.7 Image Processing

Workloads. To further highlight the applicability of Gramine-TDX, we run two image processing applications. The first one is written in Go and the second one in Java. These programs perform image processing operations, such as Gaussian Blur, edge detection, image sharpening and resizing. We use the infamous Lenna image as our input. We did not optimize these applications for parallel execution. Therefore we execute them with a single thread.

Results. Table 6 presents the performance of our examined variants for the image processing applications. We observe that Gramine-TDX increases the execution time of these applications only slightly (< 10%), as the processing is performed in-memory. This experiment further showcases that Gramine-TDX is a versatile framework that can seamlessly support multiple languages and runtimes.

7.8 Boot Time

Workloads. We compare the startup time of Gramine-TDX against normal VMs and Intel TDX VMs. For Gramine-TDX, we run a hello-world application and measure its end-to-end execution time (to err on the side of overapproximation). For the other VM variants, we use the `systemd-analyze` tool and present the resulting startup time. The presented values are an average of 10 runs.

Results. Table 7 presents the boot time of our examined variants for different VM configurations. For normal VMs, the startup time slightly increases with the assigned VM memory. The Intel TDX

VMs exhibit a similar trend but with marginally higher startup times. Notably, Gramine-TDX outperforms the baselines for small VM sizes, but its performance significantly declines as the allocated VM memory increases due to Gramine-TDX's memory initialization.

8 RELATED WORK

To build trust in confidential computing technologies, TEE vendors adopt an approach that involves collaborative analysis and thorough reviews with third-party experts. Both Intel TDX and AMD SEV have collaborated with Google's security teams to examine threat models and identify potential vulnerabilities [21, 29]. Despite acknowledging that CVMs running on a host can be under attacker control and may contain undiscovered issues in their interactions with untrusted components, a detailed security analysis on the attack surface of CVMs, as conducted in this paper, is *not* their primary focus. Besides, researchers identified vulnerabilities in VM-based TEEs that could lead to data extraction [57–59, 67, 68, 94], control-flow hijacking [37], code execution [38, 69, 75, 95, 96], and controlling the root of trust [27].

Currently, the most related project to Gramine-TDX is the COCONUT SVSM (Secure VM Service Module) [4]. It implements a secure module for CVMs that runs in a special privileged level below the guest VM. COCONUT can be considered as a special firmware that is accessed by the guest OS via a standardized interface. It is supposed to run security-critical modules (e.g., virtual TPMs), in a minimal isolated environment, separated even from the trusted guest OS. Similar to Gramine-TDX, COCONUT implements its own low-level primitives (e.g., scheduling, page tables), in a secure and hardened manner. In contrast to Gramine-TDX, COCONUT does not target unmodified Linux applications but requires them to be written from scratch or be manually ported to its APIs.

Another related project is Hecate [35]. Hecate builds an L1 hypervisor that runs inside a CVM and transparently shields the nested legacy VM from the untrusted host (L0) hypervisor. Thus, the effort of hardening shifts to the software running in L1. Hecate chooses a custom-configured Linux kernel as L1 hypervisor and discusses *some* hardening, but does not provide a systematic study on required mitigations. As we highlight in §2.3, Linux is notoriously hard to harden. Additionally, Hecate uses VMPLs present in AMD SEV-SNP architecture but missing in Intel TDX.

Linux-based CC solutions like Confidential Containers [5] and Oak Containers [33] facilitate the execution of cloud-native workloads in CVMs at the cost of involving a huge and potentially vulnerable guest OS. Driven by such concerns with monolithic OSes, prior works including Oak's restricted kernel alternative [33], Keystone [53], M^3 [93], and Enarx [1] advocate for the integration of a smaller, verifiable microkernel like Sel4 [46] into TEEs. Gramine-TDX makes an alternative design choice in favor of a unikernel/Library OS approach, which also has a small TCB and is amenable to formal verification. There exist Library OSes designed for process-based TEEs, such as Occlum [81], SGX-LKL [74], and MystikOS [14]. However, to our knowledge, none of these support VM-based hardware TEEs as Gramine-TDX does.

vSGX [99] and NestedSGX [92] allow legacy Intel SGX enclaves to run within AMD SEV CVMs. Contrary to Gramine-TDX, minimization of the TCB is *not* the goal of vSGX: it puts the whole Linux kernel, modified for vSGX purposes, inside the CVM. NestedSGX extends the SVSM framework to provide isolated enclave execution using distinct Virtual Machine Privilege Levels (VMPLs). Although it may be possible to adapt Gramine-SGX to Intel TDX CVMs following a similar setup, this requires non-trivial porting effort. Instead, Gramine-TDX is built from a clean slate, targeting CVM-specific attack surfaces with a minimal TCB.

Apart from Linux hardening (§2.3), there is *kernel debloating*, a practical approach that mitigates the OS kernel's security vulnerabilities by reducing its attack surface. This strategy permits only necessary kernel code execution as required by specific applications, by either removing superfluous code statically [41, 48, 51, 61, 84] or rendering it inaccessible during runtime [18, 49, 98]. However, the primary objective of almost all kernel debloating is to curtail the attack surface between ring-3 and ring-0, *rather than between ring-0 and the hardware*. Even though the ring-0/hardware attack surface might witness a reduction as a consequence of kernel minimization, such an effect is largely incidental and warrants further investigation on a case-by-case basis. The recent undertaking proposed in [41] approaches kernel debloating from the perspective of focusing on a machine's specific hardware device inventory. However, its original intent remains to decrease the ring-3/ring-0 attack surface. Concerning other mechanisms devised to decrease the kernel attack surface such as syscall filtering [32, 36, 45, 73], these are also *not* tailored to harden the ring-0/hardware attack surface.

Lupine [50] and X-Containers [82] demonstrate the potential to customize a general-purpose OS (Linux) via application-specific configurations at build time. However, they require non-trivial modifications to Linux, which likely hinders their acceptance upstream. Gramine-TDX is self-contained and has a smaller TCB (i.e., the uncompressed kernel-image binary of Lupine is over 20MB [39, 40]). Language-safe OSes, such as MirageOS [42], Tock [55], Theseus [26], and RustyHermit [52] promise security and lightweight properties. Yet, they typically lack POSIX support and are not designed for TEEs with attack surface minimization in mind. Gramine-TDX, implemented in C, enhances memory safety through its static (Coverity) and dynamic (sanitizers) analysis. Similar to Gramine-TDX, Enarx [1] targets a minimal TCB within TEEs and supports heterogeneous TEEs (Intel SGX, AMD SEV-SNP) by using WebAssembly and a microkernel. It cannot run unmodified binaries, necessitating the recompilation of applications into WebAssembly.

Gramine-TDX chooses to outsource the FS and network stacks to the host. The benefits and caveats of this choice are discussed in [54]. NetKernel [71] provides insights about the security-performance tradeoffs on outsourcing the network stack, whereas Obliviate [20] provides similar insights on outsourcing the FS stack. Finally, Bifrost [56] introduces performance optimizations to the network stack while complying with the CVM threat model; many of Bifrost's insights and techniques can be applied to Gramine-TDX.

9 CONCLUSION AND DISCUSSION

This paper stresses the need for a small *security-first* OS kernel targeted for confidential VMs.

Niche usages. We envision single-process cloud-native applications as primary targets for Gramine-TDX. However, we believe that Gramine-TDX can also find its niche in other security-critical areas. One example is TEE-specific utilities like the *TD migration* tool. Another example is running a software-based TPM under Gramine-TDX. Yet another example is running Gramine-TDX as the minimal L1 hypervisor in Intel TDX Partitioning [9].

Docker and Kubernetes integration. Given the process-like deployment model of Gramine-TDX, it is trivial to use it with Docker and Kubernetes (or other orchestration systems). To this end, tools like Gramine Shielded Containers (GSC) may come handy that take the original Docker image, add the Gramine environment to it, and replace the entry point with the Gramine-TDX binary [8]. The resulting "graminized" image can be stored in the Docker image registry and used in Kubernetes pods.

Choice of hypervisor. For prototyping, we choose the QEMU/KVM hypervisor. However, Gramine-TDX uses standard virtio devices and VM techniques (e.g, PCI bus discovery) that should be present on any reasonable hypervisor. In particular, we believe that Gramine-TDX should be trivial to run under Cloud Hypervisor. Moreover, we would like to explore library hypervisors like *libkrun* [13].

Choice of VM-based TEE. We expect VM-based TEEs to be more widely accessible than process-based TEEs. Gramine-TDX mainly consists of TEE-agnostic code. There are only ∼2K LoC that are Intel TDX-specific. Therefore we believe that porting Gramine-TDX to, e.g., AMD SEV would be a feasible task [22].

Cloud-native deployments. Currently, Gramine-TDX prototype is at odds with typical cloud deployments, where a user rents a VM from the cloud provider and *cannot* start a new VM/TD in a nested way. Unfortunately, nested virtualization is unsupported in Intel TDX. Therefore, the only way to use Gramine-TDX in the cloud to date is to rent a bare-metal instance. We are currently investigating a potential for *co-located VMs*, where a "parent" VM is allowed to spawn a "child" Gramine-TDX VM on the same host and share the network/FS; this is the model adopted by AWS Nitro Enclaves [2].

Artifact availability. Gramine-TDX is publicly available along with its entire experimental setup.

Acknowledgements. We thank our shepherd and the anonymous reviewers for their helpful comments, as well as Dan Middleton, Scott Raynor, Benny Fuhry, Michael Beale, Elena Reshetova, Vidhya Krishnamurthy, Anjo Vahldiek-Oberwagner, Yuan Xiao, Isaku Yamahata, Michał Kowalczyk and Borys Popławski. This work was partially supported by an ERC Starting Grant (ID: 101077577).

REFERENCES

[1] 2023. Enarx: Confidential Computing with WebAssembly. https://enarx.dev/
[2] 2024. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.
[3] 2024. Cloud vs. Edge. https://www.redhat.com/en/topics/cloud-computing/cloud-vs-edge.
[4] 2024. COCONUT SVSM. https://github.com/coconut-svsm/svsm.
[5] 2024. Confidential Containers. https://github.com/confidential-containers.
[6] 2024. Firecracker: Linux guest configs. https://github.com/firecracker-microvm/firecracker/tree/main/resources/guest_configs.
[7] 2024. Gramine: A library OS for Linux multi-process applications, with Intel SGX support. https://github.com/gramineproject/gramine.
[8] 2024. Gramine Shielded Containers. https://github.com/gramineproject/gsc/.
[9] 2024. Intel TDX Module v1.5 TD Partitioning Architecture Specification. https://www.intel.com/content/www/us/en/content-details/773039/intel-tdx-module-v1-5-td-partitioning-architecture-specification.html.
[10] 2024. Intel Trust Domain Extension Guest Linux Kernel Hardening Strategy. https://intel.github.io/ccc-linux-guest-hardening-docs/tdx-guest-hardening.html.
[11] 2024. Intel Trust Domain Extensions Ready For Linux 5.19 (Intel TDX). https://www.phoronix.com/news/Intel-TDX-For-Linux-5.19.
[12] 2024. Intel® Trust Domain Extension Linux Guest Kernel Security Specification. https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html.
[13] 2024. libkrun: A dynamic library providing Virtualization-based process isolation capabilities. https://github.com/containers/libkrun.
[14] 2024. Mystikos: Tools and runtime for launching unmodified container images in Trusted Execution Environments. https://github.com/deislabs/mystikos.
[15] 2024. Virtual I/O Device (VIRTIO). https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html.
[16] 2024. What is Function-as-a-Service (FaaS)? https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas.
[17] Doug Abbott. 2004. *PCI bus demystified.* Elsevier.
[18] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *USENIX Security Symposium.* 2435–2452.
[19] Mathieu Acher, Hugo Martin, Juliana Alves Pereira, Arnaud Blouin, Djamel Eddine Khelladi, and Jean-Marc Jézéquel. 2019. *Learning From Thousands of Build Failures of Linux Kernel Configurations.* Technical Report.
[20] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. Obliviate: A Data Oblivious Filesystem for Intel SGX. In *NDSS'18.*
[21] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. 2024. Intel Trust Domain Extensions (TDX) Security Review.
[22] AMD. 2024. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more.
[23] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *OSDI'23.* 193–208.
[24] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *OSDI'16.*
[25] Mark Kuhne Andrin Bertschi Shweta Shinde Benedict Schlüter, Supraja Sridhara. 2024. Heckler: Breaking Confidential VMs with Malicious Interrupts. In *USENIX Security'24.*
[26] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an experiment in operating system structure and state management. In *OSDI'20.*
[27] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. 2021. One glitch to rule them all: Fault injection attacks against amd's secure encrypted virtualization. In *ACM CCS'21.*
[28] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. 2023. Intel TDX Demystified: A Top-Down Approach.
[29] Cfir Cohen, James Forshaw, Jann Horn, and Mark Brand. 2022. *AMD Secure Processor for Confidential Computing Security Review.* Technical Report. Google Project Zero and Google Cloud Security.
[30] Confidential Computing Consortium. 2024. White Papers and Reports. https://confidentialcomputing.io/resources/white-papers-reports/.
[31] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016).
[32] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. In *RAID'20.*
[33] Hubert Eichner, Daniel Ramage, Kallista Bonawitz, Dzmitry Huba, Tiziano Santoro, Brett McLarnon, Timon Van Overveldt, Nova Fallen, Peter Kairouz, Albert Cheu, et al. 2024. Confidential Federated Computations. *arXiv preprint arXiv:2404.10764* (2024).
[34] Gabriel P. Fernandez, Andrey Brito, and Christof Fetzer. 2024. Triad: Trusted Timestamps in Untrusted Environments. arXiv:2311.06156 [cs.CR]
[35] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. 2022. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *ACM CCS'22.*
[36] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated system call policy generation for container attack surface reduction. In *RAID'20.*
[37] Felicitas Hetzelt and Robert Buhren. 2017. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices* 52, 7 (2017).
[38] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. 2021. Via: Analyzing device interfaces of protected virtual machines. In *ACSAC'21.*
[39] Benjamin Holmes, Jason Waterman, and Dan Williams. 2022. KASLR in the age of MicroVMs. In *EuroSys'22.* 149–165.
[40] Benjamin Holmes, Jason Waterman, and Dan Williams. 2024. SEVeriFast: Minimizing the root of trust for fast startup of SEV microVMs. (2024).
[41] Zhenghao Hu, Sangho Lee, and Marcus Peinado. 2023. Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis. In *ACM CCS'23.*
[42] Takayuki Imada. 2018. Mirageos unikernel with network acceleration for iot cloud environments. In *Conference on Cloud and Big Data Computing.*
[43] Intel. 2024. Intel Protected File System Library. https://www.intel.com/content/www/us/en/developer/articles/technical/overview-of-intel-protected-file-system-library-using-software-guard-extensions.html.

[44] Intel. 2024. Intel® Trust Domain Extensions (Intel® TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[45] The Linux Kernel. 2023. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html

[46] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *SOSP'09*.

[47] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating remote attestation with transport layer security. *arXiv preprint arXiv:1801.05863* (2018).

[48] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2022. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Communications ACM* (2022).

[49] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B Bobba, David Lie, and Jesse Walker. 2019. Multik: A framework for orchestrating multiple specialized kernels. *arXiv preprint arXiv:1903.06889* (2019).

[50] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A linux in unikernel clothing. In *EuroSys'20*. 1–15.

[51] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *NDSS*.

[52] Stefan Lankes, Jonathan Klimt, Jens Breitbart, and Simon Pickartz. 2020. RustyHermit: a scalable, rust-based virtual execution environment. In *High Performance Computing: ISC High Performance 2020 International Workshops*. Springer.

[53] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *EuroSys'20*.

[54] Hugo Lefeuvre, David Chisnall, Marios Kogias, and Pierre Olivier. 2023. Towards (Really) Safe and Fast Confidential I/O. In *HotOS'23*.

[55] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *SOSP'17*. 234–251.

[56] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *USENIX ATC'23*.

[57] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. 2021. Crossline: Breaking "security-by-crash" based memory isolation in AMD SEV. In *ACM CCS'21*.

[58] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. 2019. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *USENIX Security'19*.

[59] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. 2021. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *USENIX Security'21*.

[60] Xupeng Li, Xuheng Li, Christoffer Dall, Ronghui Gu, Jason Nieh, Yousuf Sait, and Gareth Stockwell. 2022. Design and Verification of the Arm Confidential Compute Architecture. In *OSDI'22*.

[61] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *SOSP'17*.

[62] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, Luc Lesoil, Jean-Marc Jézéquel, and Djamel Eddine Khelladi. 2022. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Transactions on Software Engineering* (2022).

[63] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *USENIX Security'17*.

[64] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *HASP'13*.

[65] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. 2022. Attestation Mechanisms for Trusted Execution Environments Demystified. In *DAIS'22*.

[66] Subhas C. Misra and Virendra C. Bhavsar. 2003. Relationships between Selected Software Measures and Latent Bug-Density: Guidelines for Improving Quality. In *International Conference on Computational Science and Its Applications (ICCSA)*.

[67] Mathias Morbitzer, Manuel Huber, and Julian Horsch. 2019. Extracting secrets from encrypted virtual machines. In *ACM Conference on Data and Application Security and Privacy*.

[68] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Smascha Wessel. 2018. Severed: Subverting AMD's virtual machine encryption. In *European Workshop on Systems Security*.

[69] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. 2021. Severity: Code injection attacks against encrypted virtual machines. In *IEEE Security and Privacy Workshops (SPW)*.

[70] Dominic P Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo JM Vincent. 2021. Confidential Computing—a brave new world. In *international symposium on secure and private execution environment design (SEED)*.

[71] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. 2020. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. In *USENIX ATC'20*.

[72] Rolf Oppliger. 2023. *SSL and TLS: Theory and Practice*. Artech House.

[73] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages* OOPSLA (2020).

[74] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143* (2019).

[75] Martin Radev and Mathias Morbitzer. 2020. Exploiting interfaces of secure encrypted virtual machines. In *Reversing and Offensive-oriented Trends Symposium*.

[76] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. 2013. Speeding up packet I/O in virtual machines. In *Architectures for Networking and Comm. Systems*.

[77] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* (2008).

[78] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. 2021. Toward Confidential Cloud Computing. *Communications ACM* (2021).

[79] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. 2021. Demystifying attestation in Intel Trust Domain Extensions via formal verification. *IEEE Access* (2021).

[80] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. In *USENIX Security'17*.

[81] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *ASPLOS'20*. 955–970.

[82] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *ASPLOS'19*. 121–135.

[83] Kirill Shutemov. 2023. x86/coco: Disable 32-bit emulation by default on TDX and SEV.

[84] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability. In *HotDep'12*.

[85] Google Cloud team. 2024. Confidential Computing. https://cloud.google.com/confidential-computing.

[86] IBM Cloud team. 2024. Confidential Computing. https://www.ibm.com/cloud/confidential-computing.

[87] Microsoft Azure team. 2024. Azure Confidential Computing Overview. https://learn.microsoft.com/en-us/azure/confidential-computing/overview.

[88] 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. 2014. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *EuroSys'14*.

[89] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX ATC'17*.

[90] Michael S. Tsirkin and Stefan Hajnoczi. 2023. Trust, confidentiality, and hardening: the virtio lessons. https://lpc.events/event/17/contributions/1516/

[91] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or not to FUSE: Performance of User-Space file systems. In *FAST'17*.

[92] Wenhao Wang, Linke Song, Benshan Mei, Shuang Liu, Shijun Zhao, Shoumeng Yan, XiaoFeng Wang, Dan Meng, and Rui Hou. 2024. NestedSGX: Bootstrapping Trust to Enclaves within Confidential VMs. *arXiv* (2024).

[93] Carsten Weinhold, Nils Asmussen, Diana Göhringer, and Michael Roitzsch. 2023. Towards Modular Trusted Execution Environments. In *Workshop on System Software for Trusted Execution*.

[94] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. The severest of them all: Inference attacks against secure virtual enclaves. In *AsiaCCS'19*.

[95] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. 2020. Sevurity: No security without integrity: Breaking integrity-free memory encryption with minimal assumptions. In *IEEE SP'20*.

[96] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. 2021. Undeserved trust: Exploiting permutation-agnostic remote attestation. In *IEEE Security and Privacy Workshops (SPW)*.

[97] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. 2017. Cold Boot Attacks are Still Hot: Security Analysis of Memory Scramblers in Modern Processors. In *HPCA'17*.

[98] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *RAID*.

[99] Shixuan Zhao, Mengyuan Li, Yinqian Zhangyz, and Zhiqiang Lin. 2022. vSGX: virtualizing SGX enclaves on AMD SEV. In *IEEE SP'22*. IEEE, 321–336.