Building GPU TEEs using CPU Secure Enclaves with GEVisor

Xiaolong Wu
Purdue University
wu1565@purdue.edu

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

Chung Hwan Kim
University of Texas at Dallas
chungkim@utdallas.edu

ABSTRACT

Trusted execution environments (TEEs) have been proposed to protect GPU computation for machine learning applications operating on sensitive data. However, existing GPU TEE solutions either require CPU and/or GPU hardware modification to realize TEEs for GPUs, which prevents current systems from adopting them, or rely on untrusted system software such as GPU device drivers. In this paper, we propose using CPU secure enclaves, e.g., Intel SGX, to build GPU TEEs without modifications to existing hardware. To tackle the fundamental limitations of these enclaves, such as no support for I/O operations, we design and develop GEVISOR, a formally verified security reference monitor software to enable a trusted I/O path between enclaves and GPU without trusting the GPU device driver. GEVISOR operates in the Virtual Machine Extension (VMX) root mode, monitors the host system software to prevent unauthorized access to the GPU code and data outside the enclave, and isolates the enclave GPU context from other contexts during GPU computation. We implement and evaluate GEVISOR on a commodity machine with an Intel SGX CPU and an NVIDIA Pascal GPU. Our experimental results show that our approach maintains an average overhead of 13.1% for deep learning and 18% for GPU benchmarks compared to native GPU computation while providing GPU TEEs for existing CPU and GPU hardware.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

KEYWORDS

Confidential Computing, GPU, TEE

SoCC '23, October 30–November 1, 2023, San Cruz, CA, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624659

1 INTRODUCTION

Graphics processing units (GPUs) have been a popular solution to accelerate computation in applications, such as data analytics, machine learning, and deep learning in the current cloud computing environment. As a result, the associated GPU security implications have drawn more attention due to the sensitivity of the data that GPUs operate on. An attacker can exploit vulnerabilities at the OS level to gain control of the GPU driver and then access sensitive data within a GPU through the Memory-mapped I/O (MMIO) and Direct Memory Access (DMA) interfaces [41]. In addition, a malicious user can break context isolation between GPU applications running on the same GPU by tampering with the GPU page table, leaking the sensitive data processed from within victim GPU applications [54].

In light of this problem, Trusted execution environments (TEEs) for GPUs have recently been proposed and adapted to isolate and secure GPU computation through CPU modification [27], GPU modification [51], and customized hardware TEE [56]. For instance, the latest NVIDIA's Hopper H100 architecture [6] and Microsoft's confidential computing cloud with NVIDIA A100 [5] provide TEEs within GPUs. However, all these solutions require hardware changes within the CPU and/or GPU, which prevents current systems from adopting them. Moreover, silicon scaling is slowing [47]. Any security flaw found within these hardware GPU TEE implementations is likely to stay there forever until a new GPU product is released.

On the contrary, it is possible to build a GPU TEE with only software changes. StrongBox [20] uses ARM TrustZone to build a GPU TEE by assigning the GPU to the secure world and preventing accesses from the non-secure world using the TrustZone Address Space Controller (TZASC) during a secure GPU computation. While StrongBox solves the GPU TEE problem for edge devices, it cannot be applied to the cloud environment because of the dominance of Intel CPUs. Another work [55] migrates the GPU device driver from the kernel space into the trusted computing base (TCB) in the user space and isolates the channel between the driver and device using a hypervisor. Unfortunately, such an approach increases the TCB size significantly as the size of a GPU driver tends to be large (e.g., 1.79 million lines of code (LoC) as of Linux 5.9 for AMD GPUs, and 209K LoC for an open-sourced NVIDIA GPU driver (nouveau)). In addition, moving a GPU device driver from the kernel space to the user space could be infeasible at all since most commodity GPU drivers are proprietary, e.g., NVIDIA GPU drivers.

Aiming at no hardware changes for practical deployment and reducing the TCB, in this paper, we propose using CPU secure enclaves, e.g., Intel SGX, to build a GPU TEE, considering the strong security guarantees provided by SGX enclaves. However, this approach immediately faces multiple critical challenges as follows. ① *Unlike ARM TrustZone, Intel SGX was never designed to support I/O operations*. There is no trusted I/O path between enclaves and GPU by default without trusting the OS. ② *A GPU device driver is needed for GPU computation but can easily bloat up the TCB*. Then the question is how to exclude the GPU device driver from the TCB. ③ *Balance between security guarantees and performance overhead*. Cryptographic primitives are usually the building block for secure I/O path between enclave and GPU [27, 51], however, they often incur high performance overhead. Meanwhile, this kind of design requires a special CUDA kernel transfer first for key exchange. How could a user securely transferring the CUDA kernel with a trusted-path without using that trusted path to transfer data? ④ *Without rigorous security verification, a security solution may introduce a new attack surface*. A new GPU TEE solution might still suffer from typical memory safety issues and/or incomplete protections, breaking the security guarantees provided by TEE.

In this paper, we address these challenges by designing a formally-verified tiny hypervisor, GPU Enclave hyperVisor (GEVISOR) running in the VMX root mode and cooperating with SGX enclaves to enable confidential GPU computation as follows.

Trusted I/O. GEVISOR leverages the GPU runtime within the enclave to cooperate with the hypervisor to protect GPU I/O memory region during data transfer. GEVISOR confines the GPU I/O access and ensures that only the corresponding enclave can access it during its life cycle using the extended page table (EPT).

Small TCB. GEVISOR removes the GPU device driver from the TCB by isolating the channel between enclaves and the GPU from other system software in the address space and monitoring the execution of the enclaves to securely authorize GPU device access without the driver. To further reduce the TCB, GEVISOR excludes the system bootstrap code and supports dynamic hypervisor measured launch leveraging the Dynamic Root of Trust for Measurement (DRTM) [40] with a Trusted Platform Module (TPM) and SGX combined linear remote attestation.

Low Overhead. To avoid the data encryption overhead, GEVISOR provides a unified GPU I/O protection (i.e., MMIO and DMA) by monitoring the DMA memory region for data transferring between an enclave and a GPU. This design allows GEVISOR to trade the strong but expensive I/O channel encryption with a weaker but performant EPT-based protection. We further design a novel asynchronous hypercall mechanism dedicated for I/O protections to minimize context switches to and from GEVISOR.

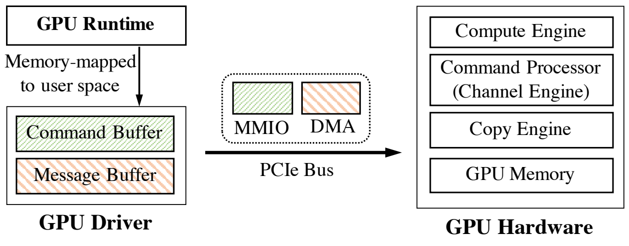


Figure 1: GPU software and hardware components.

Formal Verification. We formally verify the security guarantees of both the design specification and the implementation of GEVISOR using Dafny [33] and CBMC [18] respectively. We propose I/O dualization semantic verification by applying *noninterference* [22] on I/O protection verification.

The contributions of this paper are as follows.

- We propose building a GPU TEE using CPU secure enclaves with a tiny hypervisor (GEVISOR) that cooperates with GPU enclaves to realize TEE for commodity GPUs without any hardware changes (§5).
- We design a novel asynchronous hypercall mechanism to reduce the context switch overhead (§5.2), a linear remote attestation protocol combining TPM and SGX remote attestations (§5.4), and a unified MMIO and DMA protection solution to enable a trusted I/O path between enclaves and GPU (§6).
- We formally verify the security properties of GEVISOR design specification using dualization and step semantic verification techniques with Dafny (§7), and the ones of GEVISOR implementation using CBMC.
- We evaluate GEVISOR on two benchmark suites covering various GPU workloads and deep learning computation. Our quantitative results show that GEVISOR maintains an average overhead of 13.1% for Darknet and 18% for Rodinia while enabling confidential computing on GPUs.

We make our prototype implementation of GEVISOR publicly available.*

2 BACKGROUND

Intel SGX. Intel Software Guard Extensions (SGX) [39] ensures the confidentiality and integrity of user code and data. SGX is a subset of x86 instructions allowing a process to allocate a protected memory region, *i.e.*, an *enclave*, within its

*https://github.com/purseclab/GEVisor

address space. During booting, BIOS lays out a separate memory region within the DRAM called the Enclave Page Cache (EPC) for SGX enclaves. The EPC is access-restricted by the processor and can therefore securely execute its operations irrespective of the adversarial potential of other system components such as the operating system (OS), Virtual Machine Manager (VMM), etc. SGX comes at a cost of a performance overhead from enclave transitions when entering, resuming, or exiting an enclave.

While Intel SGX provides enclaves hardware-assisted protection against privileged attacks, it is not designed to protect external peripheral devices and their I/O channels. That is, the security boundary of SGX is within the CPU (communication between an enclave and an external peripheral device is unprotected), because all external device communication is handled by device drivers within OS by creating and maintaining a memory-mapped I/O channel to allow user-space applications to communicate with the intended devices. In the adversarial model of SGX, the OS, and thus device drivers, are untrusted as they reside outside the enclave.

Intel Virtualization Technology (VT-x). To support virtualization, Intel VT-x adds two CPU execution modes: the Virtual Machine eXtensions (VMX) root and the VMX non-root. When a VM tries to execute a sensitive instruction in the VMX non-root mode, the CPU detects it and context switches to a hypervisor in the VMX root mode (a.k.a., *VM Exit*). Intel VT-x also introduces two instructions, e.g., VMLAUNCH and VMRESUME, to switch from the VMX root mode to the VMX non-root mode (a.k.a., *VM Entry*).

Graphics Processing Units (GPU). We illustrate typical GPU software and hardware components in Fig. 1. The software stack consists of a high-level GPU Runtime and a low-level GPU driver, whereas the hardware stack includes the PCIe Bus and GPU Hardware. The GPU runtime could be the CUDA runtime and driver APIs for user-space applications (*e.g.*, NVIDIA CUDA SDK [13]). It is responsible for the creation and loading of GPU-compatible code called GPU kernels (or CUDA kernels) into the GPU device, and initiating and transmitting GPU data to and from the GPU memory through a set of APIs. The GPU device driver, such as the NVIDIA driver [13] or open-source Nouveau [2], is responsible for the creation, deletion, and upkeep of a communication channel with the GPU. The GPU hardware includes a command processor, compute and copy engines, and internal GPU memory. The command processor (a.k.a., the channel engine) is responsible for receiving commands from the GPU device driver. The compute and copy engines are responsible for processing commands issued by the command processor.

The GPU device driver contains these main functions:

Context and Channel Creation. GPU maintains contexts as part of its communication with applications and uses channels to isolate a context's address space from other contexts. A channel is the only way to submit commands to the GPU. Therefore, each GPU context allocates at least one GPU channel. The GPU driver creates one context for each GPU application process, and frees it upon the termination of the process. Each context has its own address space and is used to ensure separation of memory from other contexts. To create a communication channel with the device, the driver allocates a channel descriptor and a two-level page table within the GPU memory by accessing the Base Address Registers (BARs).

GPU Communication. A GPU driver is responsible for creating two buffers to communicate with GPU—a message buffer for code/data transferring between GPU memory and the application, and a command buffer for submitting commands to the GPU device. The *message buffer* is memory-mapped to both the application and the GPU driver's address space, which is directly accessible by the GPU's copy engine without the assistance of the host CPU through Direct Memory Access (DMA). The *command buffer* is also memory-mapped, and used to relay commands to the GPU device. The GPU runtime pushes various commands to the command buffer updating the state of the GPU. For example, the GPU initiation is achieved through specially-crafted commands directly written into the command buffer. Different from the message buffer, the command buffer is in the memory-mapped I/O (MMIO) region, which is also mapped to user-space through Memory Management Unit (MMU) so that user-space can access the GPU through virtual addresses.

CUDA Runtime. The GPU Runtime has the relevant CUDA driver APIs as follows: *cuCtxCreate* corresponds to the creation of a context and sets up of the message and command buffers in the application's address space. *cuModuleLoad* loads a specific kernel to the GPU memory through the message buffer. *cuMemcpyHtoD* copies memory from the host application to the GPU device using the message buffer. *cuMemcpyDtoH* calls result in sending a command through the command buffer to the GPU device, which returns the requested device memory to the host application through the message buffer. *cuCtxDestroy* corresponds to the deletion of a GPU context and the freeing of all the corresponding message buffers allocated to the host application.

3 SECURITY MODEL

Threat Model. We assume the attacker can control system software, including the OS, GPU device driver, guest VM, and even the commodity cloud hypervisor. The attacker can use these entities to launch an attack when a user offloads its computation from CPU to GPU. In particular, the attacker can attempt to directly monitor contents in the message and command buffers of GPU, craft new GPU commands, and send them to the GPU after compromising the system software.

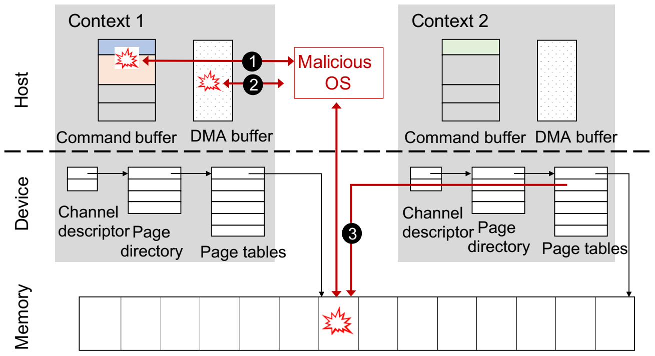


Figure 2: Attack surface.

Furthermore, we assume the attacker can also create a new GPU context and map it to the address space of a victim's GPU context to steal sensitive data. Fig. 2 shows the three attack surfaces that we consider.

Attack Surface 1: DMA Buffer. A GPU device driver allocates DMA buffers (❶) for applications running in the user space. A DMA buffer serves as a main communication channel between an application and the GPU. By having this channel under control, the attacker can manipulate data to and from the GPU.

Attack Surface 2: MMIO Mapping. An adversary can manipulate MMIO mapping (❷), which is another communication channel between CPU and GPU. Unlike DMA, MMIO involves CPU by design. Therefore, an attacker can control the mapping using a compromised OS and manipulate the data in the communication channel.

Attack Surface 3: GPU Context. An attacker controlling the GPU device driver may control the allocation of all GPU device memory and tamper with the page tables of the victim GPU context. Such an attacker could maliciously map physical GPU pages (❸) to another GPU context (or GPU application) under her control [32]. As such, the attacker would be able to steal sensitive information through her own GPU context.

Trust Model. We assume both CPU and GPU devices are trusted, including microcode with CPU. We trust the GPU device memory because modern GPUs usually integrate the device memory using 2.5D/3D silicon interposers inside the same package, which is difficult to observe and corrupt the data stored in it by the adversary [51]. We trust the measured boot built upon Intel Trusted Execution Technology TXT [1] and Trusted Platform Module (TPM). We also trust the system firmware such as Intel ME, UEFI, SMM and GPU firmware, which implies that a hypervisor, once put into the VMX root mode, has the full control of the system until VMXRESUME or VMXOFF.

Although we do not trust cloud providers, we assume a cooperative cloud vendor aiming to provide confidential computing for GPU as a service considering both security and business. Thus denial-of-service (DoS) attacks against Intel SGX or GPU are out of the scope. We do not consider physical attacks. Side-channel attacks on Intel SGX and GPU are orthogonal to the problem that we are trying to solve in this paper. Existing defense against the side-channel attacks are complementary to our solution. For the GPU context isolation protection, we expect the target GPU to support Unified Memory that CUDA supports in version 6.0 and above [49], that are readily available.

4 MOTIVATION AND DESIGN GOALS

Four directions exist to enable a trusted GPU execution environment. A comparison between the recent relevant works in these directions and ours is presented in Table 1. ① The most straightforward solution is enabling CPU TEEs to talk with GPU directly by modifying CPU hardware. For instance, HIX [27] modifies SGX CPU to remove its I/O limitation and builds a GPU enclave with the GPU driver inside. ② Instead, another solution is modifying GPU architecture to isolate GPU contexts by providing an internal TEE, such as Graviton [51]. However, this approach cannot be applied to most deployed GPU devices. ③ The third solution introduces a hypervisor as the TCB and includes a GPU device driver within the hypervisor or user space instead of relying on the one from OS, e.g., TrustedPath [55]. However, commodity hypervisors like Xen [9] or GPU drivers like NVIDIA could easily bloat the TCB. ④ The last solution relies on remote attestation to reduce the TCB, such as HETEE [56] and StrongBox [20] moving GPU driver, runtime, and the AI framework out of the TCB.

Compared with the previous work [20, 27, 51, 56], we argue that the security guarantees of trusted GPU execution should triumph over all other considerations, e.g., covering all the attack surfaces mentioned earlier while maintaining a small TCB and even providing formal guarantees. Meanwhile, as a practical solution, it cannot introduce any hardware changes so that it can be adopted and deployed to existing systems more likely. Lastly, the overhead of securing GPU computation should be low to be used in practice. Based on these considerations, we list a number of design goals that any desired solution of GPU TEE should strive to achieve.

- **G1: Complete Mediation.** We need to mediate all the attack surfaces exposed by a GPU communication channel to realize a GPU TEE, including GPU I/O accesses (e.g., DMA and MMIO), GPU contexts of different processes, and the temporary sensitive code and code exposure within the host memory.

Table 1: Comparison between existing works and GEVISOR.

	Hardware Change	Software TCB	Formal Verification	GPU I/O Protection	GPU Context Isolation	GPU Attestation Support	Performance Overhead
HIX [27]	CPU	39K-2M (GPU driver)	-	✓	-	✓	26%
Graviton [51]	GPU	- (firmware)	-	-	✓	✓	17-33%
HETEE [56]	FPGA	- (firmware)	-	✓	-	✓	2.17%
TrustedPath [55]	-	18K-2M (driver+hypervisor)	-	✓	-	-	2×
StrongBox [20]	-	$\approx 0.3M$ (optee+security monitor)	-	✓	✓	-	4.7-15.26%
GEVISOR	-	$\approx 3.8K$ (GEVISOR)	✓	✓	✓	✓	13.1%

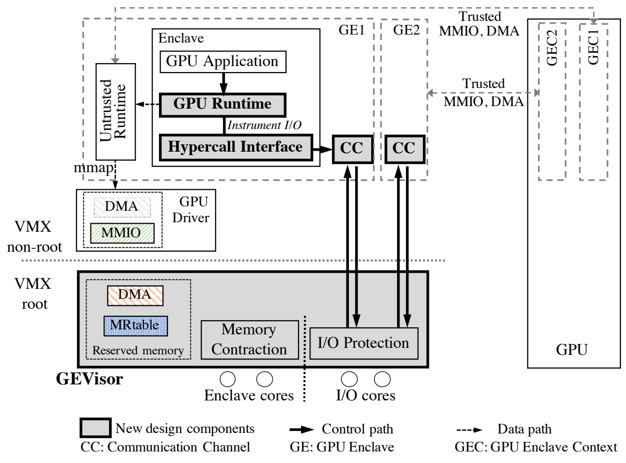


Figure 3: System architecture of GEVISOR.

- **G2: Tamperproofness.** The solution needs to be tamperproof against threats and attacks outside the TCB. For example, a compromised GPU driver cannot tamper with the GPU TEE or the GPU I/O channel.
- **G3: Verifiability.** The whole TCB should be small, thus allowing for formal verification of the code for both correctness and security, and minimizing the attack surface [50].
- **G4: Deployability.** The solution should be deployable for off-the-shelf commodity systems without requiring any hardware or major software change. Cloud providers could be incentivized to provide such a solution as a new feature to complement their existing confidential computing infrastructure.
- **G5: Low Overhead.** The solution should maintain a low performance overhead compared to a native GPU acceleration to be used in practice.

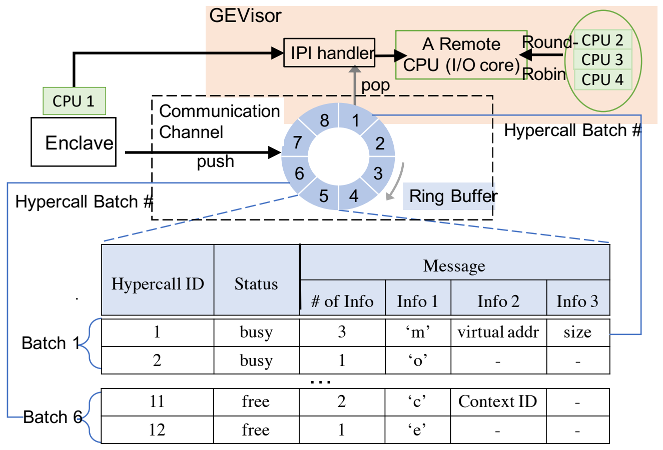
5 SYSTEM DESIGN

We propose a co-operative and lightweight hypervisor architecture in Fig. 3, which includes GPU Enclave (GE), a tiny hypervisor named GEVISOR, and the interaction between them, to achieve GPU TEE.

The GPU enclave consists of two components: UNTRUSTED RUNTIME and ENCLAVE GPU RUNTIME. UNTRUSTED RUNTIME is located within the non-enclave memory that is untrusted and can potentially behave maliciously. UNTRUSTED RUNTIME memory-maps the DMA and MMIO buffers to the user space and is responsible for GPU context creation by contacting the GPU device driver. ENCLAVE GPU RUNTIME has to copy data from the enclave to the GPU DMA and MMIO buffers within UNTRUSTED RUNTIME, to communicate with GPU. We design GEVISOR, a lightweight, late-launch, and formally security verified hypervisor, which coordinates with ENCLAVE GPU RUNTIME, to protect the two buffers from attacks. The interaction between GPU Enclave and GEVISOR consists of a communication channel (CC), a shared memory space between the enclave and GEVISOR, and a linear remote attestation protocol. ENCLAVE GPU RUNTIME passes the access control information from within the enclave through the communication channel to the GEVISOR for monitoring.

Our system achieves the aforementioned design goals in §4. GEVISOR mediates all GPU accesses for an enclave at a low performance overhead (**G5**) without sacrificing deployability to current systems. It works as a trusted reference monitor to mediate all GPU I/O operations from the VMX root mode (**G1**) with memory contraction and I/O protection function. GEVISOR isolates the GPU device's context of the enclave from other untrusted processes that share the same GPU to prevent them from accessing the GPU code and data in the shared GPU memory. GEVISOR is tamperproof from possible attacks from the non-root mode and maintains a small TCB formally verified for integrity, confidentiality, and isolation (**G2** and **G3**). This design allows us to avoid any changes to the GPU driver and GPU hardware, achieving **G4**.

We consider two threats when using enclave to build GPU TEE. When the enclave is executing, the attacker could attempt to access these buffers through another physical or hyperthread core. For this threat, we obtain SGX enclave ID, instrument the GPU context creation function and I/O memory access in GPU runtime, including the command and message buffer addresses for securing MMIO and DMA respectively (§6.1) and GPU memory pages address for context isolation (§6.2). This information is transferred to GEVISOR through the communication channel for monitoring. For the



Hypercall ID	Status	Message			
		# of Info	Info 1	Info 2	Info 3
1	busy	3	'm'	virtual addr	size
2	busy	1	'o'	-	-
		…			
11	free	2	'c'	Context ID	-
12	free	1	'e'	-	-

Figure 4: Enclave hypercall offloading.

enclave and GPU context creation, we pass the enclave and context IDs; while for the memory regions, we transfer the virtual address and size of these regions. When the enclave is halted, we instrument the entry point and the exit point of ECALL and OCALL (§8) to authorize enclave access to the MMIO regions and DMA buffer when the enclave resumes (§6.1).

5.1 GEVisor

GEVisor consists of memory contraction, IO protection, and MRTable. The techniques used in GEVisor— the late-launch method, I/O protection cores, and Extended Page Table-based protection aim to achieve the design goals **G2** and **G5**.

In order to reduce the TCB size, GEVisor uses a *late-launch method* to eliminate the machine bootstrap code. Specifically, we design a dynamic hypervisor launch method using Intel TXT [1] which allows launching and measuring GEVisor after the platform has been initialized. During the bootstrap phase, the user loads the Secure Initialization Authenticated Code Module (SINIT ACM) [8], TBoot [35], and GEVisor into memory. Our Chain of Trust is: CPU → SINIT ACM → TBoot (MLE) → GEVisor. To launch GEVisor, 1) TBoot invokes SENTER instruction with the physical address of SINIT ACM as the parameter; 2) the CPU microcode measures the SINIT ACM; 3) the SINIT ACM first checks the hardware (CPU, chipset) and the BIOS, then measures the TBoot; and 4) the TBoot provides a Measured Launch Environment (MLE) for GEVisor, measures and boots GEVisor. The hypervisor measurement result is used in linear remote attestation in §5.4.

Once launched, GEVisor reserves a few CPU cores (i.e., the I/O cores in Fig. 3) which do not run SGX enclaves but *for GPU I/O protection exclusively* without context switching to the VMX non-root mode. This design, combined with the asynchronous hypercall in §5.2, essentially allows to run enclaves and GPU I/O access control in parallel and reduce the number of context switches. Although the design of GEVisor *does not mandate it*, we dedicated these cores for the hypervisor as a performance optimization in our prototype implementation.

During the runtime, GEVisor relies on an Extended Page Table (EPT) to protect pages with sensitive code and data. Unlike the typical EPT usage where trapping happens for all the pages that belong to a VM, our memory contraction approach first *selects only a small subset of memory regions for EPT trapping*, including the MMIO/DMA memory regions, SGX enclave pages, and GPU communication channel pages. This design allows the host system to access most physical memory on the machine without EPT violation triggered. We further create a base EPT for *one-to-one mapping from Guest Physical Address (GPA) to Host Physical Address (HPA)*, reducing the overhead of traditional two-level Guest Virtual Address (GVA) to HPA address translation. Note that VM Function (VMFUNC) instruction was introduced from the Haswell generation and provides a hardware support for switching between multiple EPTs. Instead, GEVisor considers a more general EPT usage without assuming the availability of VMFUNC for compatibility, leaving VMFUNC as a complementary option for more recent CPUs.

5.2 Asynchronous Hypercall Offloading

Traditionally, EPT-based trapping mechanism for I/O monitoring has high overhead for large pages (e.g., DMA pages) due to high number of context switches and its page granularity based implementation, which is even worse in enclave environment. For one page access, EEXIT is needed to exit the enclave first and then execute VMEXIT to exit the VM mode. Similarly, to return from VMX root mode, VMENTER and EENTER will be needed to resume the enclave execution. In total, four context switches would happen, polluting local CPU cache lines.

We propose a novel asynchronous hypercall mechanism for I/O protection that offload the I/O monitoring task to the remote I/O cores. Observing that I/O protection requests from enclaves only carry small payload, e.g., I/O memory range and enclave status, without return values, we design an enclave hypercall interface with a unified hypercall entry format (Fig. 4). Each entry contains six fields: hypercall ID, the status, number of arguments, and arguments (payload), filling up one cache line (e.g., 64 bytes in x86_64). Our design supports batching hypercalls for group execution and asynchronous execution to minimize the frequency of context switches. Different from typical hypercall designs, asynchronous hypercall execution is offloaded to I/O cores reserved by GEVisor

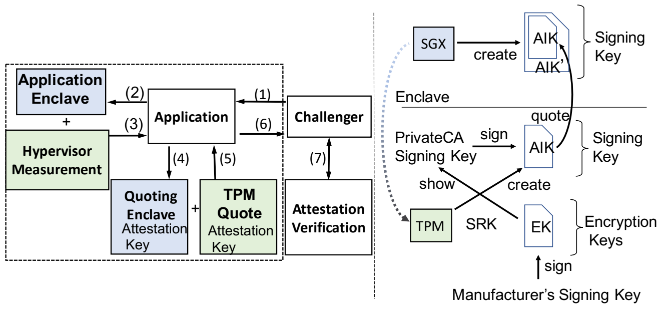


Figure 5: Linear remote attestation.

rather than holding the local enclave core. For instance, an enclave core could keep pushing hypercall requests into the ring buffer items with $free$ status without waiting for the returning of any of them (Fig. 4). Upon the request, GEVISOR will set the status to *busy* and forward it to an I/O core through the communication channel and the shared hypercall pages. We provide an Inter-processor Interrupt (IPI) handler for each I/O core to process the hypercall offloading following a round-robin fashion. Whenever the processing is completed, the status is changed from *busy* to $free$. In sum, we make the asynchronous hypercall design choices to achieve **G5**.

5.3 Communication Channel Protection

As shown in Fig. 3, the actual communication between the enclave hypercall and GEVISOR is via communication channels (CC), which is essentially a memory region allocated by the enclave host process and protected by GEVISOR. During the creation of an enclave, the host process `mmaps` a memory region at a fixed address (e.g., 0x100000), which has been agreed upon by both the host process and GEVISOR. Once allocated, GEVISOR protects CC using EPT to trap accesses to the CC pages. Traditionally, we could use `CR3` to distinguish accesses from different processes. However, we also need to prevent accesses from the host process of the enclave, which cannot be distinguished from the enclave itself using `CR3`. Instead, we track the state of enclaves to assign access permissions as needed. Specifically, we revoke the access permission of CC when an enclave is created by trapping the EINIT instruction, and when it exits or terminates by trapping the EEXIT and EREMOVE instructions. Similarly, the access permission of CC will only be assigned when an enclave starts to execute again, e.g., EENTER and ERESUME.

5.4 Linear Remote Attestation Protocol

Since our TCB includes both enclave and GEVISOR, we propose a linear remote attestation schema as shown in Fig. 5, which concatenates SGX enclave measurement and hypervisor measurement with TPM, as well as enclave quoting and TPM quoting, achieving the design goals **G3** and **G4**. We create the attestation key for enclave from TPM, and issue the certificates for enclave based on the AIK issued for hardware TPM. In particular, we create two attestation keys from TPM: the first AIK for hypervisor attestation; the second one is used to generate AIK′ for enclave. Meanwhile, the first AIK is used to certify the enclave attestation key instead of acquiring the certificate from a certificate authority. The resulting certificate chain ties the enclave's AIK′ to the AIK of the hardware TPM, and thus to the hypervisor. The advantage of this solution is that once an AIK has been issued for the hardware TPM, AIK′ for enclave can also be quickly certified. Through the chain, a link is established to the hardware-TPM platform, which make enclave and hypervisor symbiosis.

6 GPU PROTECTION

While GEVISOR provides the mechanisms to enforce page-level access control using EPT and hypercall, we elaborate how we achieve a trusted GPU execution with the help of GEVISOR protecting GPU I/Os, isolating GPU contexts.

6.1 Unified GPU I/O Protection

GEVISOR has to ensure that the DMA and command buffers are inaccessible to an attacker who attempts to use the CPU to access these memory regions. In particular, the attacker can attempt to access these buffers either while the enclave is executing (through another physical or hyperthread core) or while the enclave is in a halted state. In the following, we explain how GEVISOR prevents both types of attacks.

As illustrated in Fig. 1, GPU I/O includes both MMIO and DMA, both of which are outside SGX enclaves and vulnerable to the tampering from privileged software. To protect these memory regions, we propose an unified MMIO and DMA memory access control solution. In particular, GEVISOR maintains memory region mapping tables (MRtable) containing the virtual and physical address pairs of both MMIO and DMA memory regions per enclave within a reserved memory region, and traps accesses to these regions for access control, as shown in Fig. 6.

When an enclave starts, the GPU runtime within the enclave will pass the virtual addresses (VAs) of MMIO and DMA of the process to GEVISOR, as well as the enclave ID, via the communication channel (CC). GEVISOR will take a software page walk to get the corresponding physical addresses (PAs) instead of trusting the one within OS, and fill up an MRtable accordingly. During the execution of enclaves, GEVISOR can enforce access control of GPU I/O pages using either EPT or asynchronous hypercall, as shown in Fig. 6.

EPT Trapping. We remove the read and write permissions of each page within the MRtable ahead of time. Whenever the GPU driver tries to read or write the memory region within the MRtable, an EPT exception will be triggered. The EPT

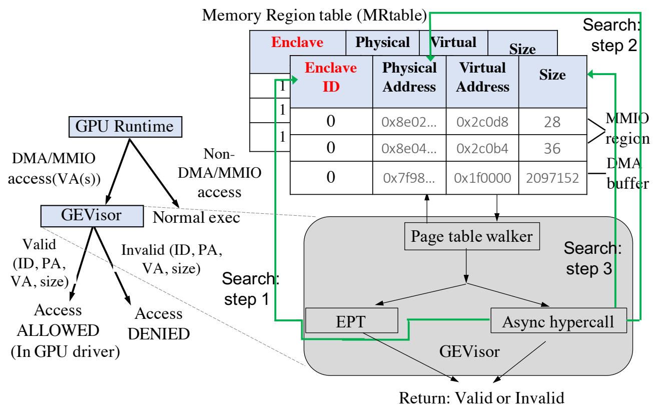


Figure 6: Memory access control.

exception handler validates the memory access by checking (1) the current process has an enclave ID registered in the MRtable; (2) the VA in the EPT entry matches the one in the MRtable; (3) the PA in the EPT entry matches the one in the MRtable.

Asynchronous Hypercall. During runtime, instead of revoking the EPT table entry's read and write permission for each DMA memory region page, Enclaves can also use asynchronous hypercall to proactively pass the MMIO and DMA access requests from within the enclave. We build an *access-list* containing all the GPU I/O addresses during normal executions of enclaves within GEVISOR. We then compare the list against the MRtable one enclave ID, PA, and the size of the memory access.

Upon an attack detection, GEVISOR handles illegal accesses by injecting a General Protection Fault (GPF) into the host OS, which will terminate the offending process. We provide two different access control mechanisms considering their pros and cons. For instance, while the expense of EPT trapping might be considerable, it has more number of context switches [7]. The asynchronous hypercall design should be lighter than the EPT one, but might introduce more overhead when batching and offloading to a limited number of I/O cores.

Concurrent CPU Access. The attacker can reside on a different non-enclave concurrent CPU core and attempt to access the physical memory region where the DMA and command buffers are located, while the enclave is currently executing. GEVISOR prevents this attack by enforcing that only the enclave-executing core(s) can actually access these memory buffers.

Halted CPU Access. As an EPT access is granted to the entire (physical or hyperthread) core, we have to revoke the access permission of the MMIO and DMA regions as soon as the enclave stops executing on that specific core. Failure to do so would mean that the attacker can access these memory regions through this core as soon as the enclave exits and break the security guarantees of GEVISOR. In particular, during an enclave execution, there may be three reasons that cause an enclave stops executing: (a) OCALLs, (b) Asynchronous Enclave Exit (AEX), or (c) enclave teardown. When the enclave performs an OCALL (e.g., to issue a system call), GEVISOR detects the message passed through the protected communication channel (CC) and removes the EPT table entry's read and write permission of the memory regions within the MRtable (§8). When an AEX event (e.g., a hardware interrupt in a CPU core) is detected by GEVISOR, it sets up EPT protection for sensitive memory regions to prevent illegal access. Similarly, GEVISOR detects enclave termination (e.g., when CUCTXDESTROY is invoked or the program causes a segmentation fault) and makes sure that the GPU runtime cleans up the DMA and command buffers before allowing the OS to access these buffers.

Defending against Malicious Peripherals. In a system with a malicious device, the compromised OS can map the MMIO memory of the malicious device to overlap with MMIO or DMA memory region of the GPU. We set up IOMMU to prevent this DMA mapping overlap. For malicious devices on MMIO accesses, GEVISOR check PCI configuration space and the Base Address Registers (BARs) by trapping the execution of all IN/OUT instructions via configuring the I/O port access bitmap within the VMCS.

Context ID	Enclave ID	Physical Address	Virtual Address	Size
0	1	0x1ee4d20	0x600000	2097152

Figure 7: Ownership data structure.

6.2 GPU Context Isolation

Under the unified virtual address space of GPU, the GPU page tables translate a GPU virtual address into both a GPU device physical address and a host physical address for initiating the DMA, which means different GPU contexts will reflect as different physical pages within the host memory. Enforcing access controls to these pages from GEVISOR will essentially allow us to achieve GPU context isolation. Similar to the MRtable, we design the ownership metadata (OM) table as illustrated in Fig. 7 to track the ownership of different GPU contexts. The GPU runtime will notify GEVISOR when a new GPU context is created, with the context ID, enclave ID, the virtual address, and the data size. GEVISOR takes a software page walk to translate the virtual address into a physical address, and fill up the OM table. Whenever the GPU driver tries to access the GPU memory region within OM, GEVISOR will trap the access using EPT and check the access permission based on the OM.

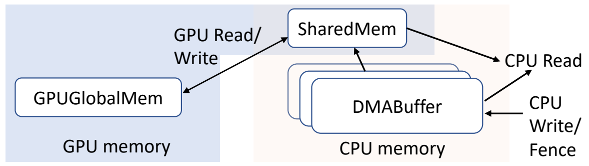


Figure 8: Memory system state that combines GPU and CPU.

6.3 Multi-User Support

We build separated communication channels and MRtables for multiple users, thus multiple enclaves can run simultaneously. This design avoids the race condition and allows enclaves parallel execution. GEVISOR support multiple GPU contexts for each enclaves, which is different from HIX [27] that each enclave corresponds to one GPU context. Multiple GPU contexts execute by context switches in GPU. If there is no any pending request for kernel execution in the current context, a context switch occurs to another context.

7 FORMAL VERIFICATION

With the design and the GPU I/O protection of GEVISOR in mind, we achieve **G3** by formally verifying the security guarantees provided by GEVISOR. For instance, we formally prove that GEVISOR maintains the consistency invariants of GPU I/O page states, and guarantees confidentiality, integrity, and isolation of GPU I/Os. We start with building a CPU-GPU memory model capturing the operational semantics of CPU-GPU communications. We specify only the features that a GEVISOR security needs. We then formalize the design specification of GEVISOR, followed by a formal proof of its security guarantees.

7.1 Heterogenous CPU-GPU Memory Model

As any shared-memory system, the behavior of loads and stores between heterogenous CPU-GPU shared memory is governed by a memory model. The heterogenous CPU and GPU shared-memory follows a relaxed memory model. To capture the operational semantics of CPU-GPU communications, we design a heterogenous programming language (HPL) for CPU-GPU model. HPL is heavily inspired by the kernel programming language (KPL) proposed in GPUVerify [12], which is an abstraction of GPU kernel code. Instead, HPL provides an abstraction for GPU devices, as shown in Fig. 9.

The state of the system in Fig. 8 can evolve by the CPU or GPU performing the memory action. We write $oldstate \xrightarrow{r} newstate$ to denote a state transition that coincides with the sending or receiving of action r.

We further define the operational semantics for CPU-GPU shared memory communication. **CPU Write** removes the entry at the head of the DMA buffer and updates the shared

$$
\begin{aligned}
l \in Loc &::= \mathbb{N} \\
v \in Val &::= \mathbb{Z} \\
t \in Tid &::= \{0, \ldots, T-1\} \\
c \in GPUContext &::= \{0, \ldots, N-1\} \\
DMABuffer &::= (Loc \times Val)list \\
DB \in DMABuffers &::= Tid \rightarrow DMABuffer \\
State_{cpu} &::= DMABuffers \\
GM \in GPUGlobalMem &::= (\{R, W\} \times Loc \times Val)list \\
State_{gpu} &::= GPUcontext \rightarrow GPUGlobalMem \\
SM \in SharedMem &::= Loc \rightarrow Val \\
s \in SyState &::= State_{gpu} \times SharedMem \times State_{cpu}
\end{aligned}
$$

Figure 9: HPL syntax.

memory with the corresponding value. **GPU Write** means that a write entry can be removed from the head of GPU global memory, whereupon shared memory is updated. **CPU Step** and **GPU Step** describe how the overall system can evolve as a result of a step on the CPU side or GPU side. **Fence** blocks the next execution until the actions in one GPU I/O execution are complete, which corresponds to *threadfence_system* in CUDA. **Sync** describes the synchronize primitive corresponding to *cudadevicesynchronize* or *cudastreamsynchronize*. Both **Fence** and **Sync** model the relaxed semantic of memory model.

7.2 Design Specification of GEVISOR

With the help of our heterogenous memory model, we further model the design specification of GEVISOR. At the core of this design specification is an abstraction of the MRtable and the OM. We use a predicate describing the contents of MMIO, DMA entries, and the page table mappings at the time of execution, as well as the EPT exception handler and the async-hypercall handler (i.e., the IPI handler). The specification relates the concrete I/O protection with the abstract MRtable and OM states after taking a GPU access to the final states (s' and m') prior to returning:

predicate *handler(s: state, m: MRtable, s′: state, m′: MRtable)*;
predicate *handler(s: state, m: OM, s′: state, m′: OM)*

A valid MRtable satisfies the invariants of internal consistency: e.g., enclave ID is correct, and all VA-to-PA mappings in a page table belong to the same address space. Similarly, OM satisfies its own invariants. Furthermore each EPT exception and async-hypercall preserves the MRtable and the OM invariants. All these invariants will form the basis of our security proofs.

The most challenging part is modeling the page table walker. We do not directly model virtual memory translation given the state transition prerequisite of virtual memory modeling. Instead, since the CUDA driver API directly manipulate the contents of the memory map at the address specified by its parameter, we define address validity solely based on the effective address, rather than the overall machine state, which simplifies the verification since prover validity is not affected by the state changes.

7.3 Formal Proof

We formally prove that the GEVISOR specification described above protects the confidentiality and integrity of enclave's GPU accesses, as well as GPU context isolation. We establish the security properties of GEVISOR by proving that the enclave's GPU I/O accesses are non-interference [21, 22] under an adversary who controls the GPU driver, e.g., 1) the GPU I/O status associated with one enclave is non-interference with states observable outside the enclave, and 2) the state which can be influenced by software outside the enclave is noninterference with the GPU I/O status of the associated enclave.

Different from existing methods reasoning about the entire execution traces [50], we focus on the GPU I/O status at the transition points to simplify our proof, based on the fact that attackers can only observe the execution of enclaves and GPU passively, since active attacks would be detected by GEVISOR. The transition points in our system are the beginning and end of enclave's GPU I/O accesses. We consider states *(s, m)*, which comprise a concrete I/O execution machine state *s* and an abstract MRtable *m*, such that *s* is an instantiation of *m*. Our confidentiality property claims that publicly observable GPU I/O outputs depend solely on publicly observable GPU I/O inputs. Our integrity property claims that trusted GPU I/O outputs depend solely on trusted GPU I/O inputs.

We formalize the confidentiality and integrity with a third-party observer. For the proof of confidentiality, the observer is a malicious GPU driver, *drivGPU*. For the integrity proof, the observer is a trusted GPU I/O access from an enclave, *encGPU*. GPU MMIO and DMA memory pages in the MRtable are linked to each enclave with an enclave ID. Therefore, *encGPU* represents a GPU I/O address space that identifies an enclave's GPU access. We define $\doteq_{encGPU}$ as relating MRtable entries and the GPU I/O pages that look the same to the enclave GPU I/O space when they are outside its address space *encGPU*:

DEFINITION 1 (Equivalence of GPU I/O pages $\doteq_{encGPU}$). MRtable entries m_1, m_2 are related by $m_1 \doteq_{encGPU} m_2$ *iff*:

$$\begin{aligned}(m_1.State_{cpu}? \wedge m_2.State_{cpu}?)\\ (m_1.\text{MMIOPage}? \wedge m_2.\text{MMIOPage}?)\\ \vee(m_1.\text{DMABufferPage}? \wedge m_2.\text{DMABufferPage}?)\\ \vee(m_1.\text{enclaveID}? \wedge m_2.\text{enclaveID}?)\end{aligned}$$

THEOREM 1 (INTEGRITY). Let the execution of EPT exception handler or IPI handler beginning in state(s, m) and returning in state(s′, m′) be denoted as handler(s, m, s′, m′). Then,

$$\begin{aligned}\forall(s_1, m_1), (s_2, m_2), (s_1', m_1'), (s_2', m_2').(s_1, m_1) \doteq_{encGPU} (s_2, m_2)\\ \wedge handler(s1, m1, s1', m1') \wedge handler(s2, m2, s2', m2')\\ \implies (s_1', m_1') \doteq_{encGPU} (s_2', m_2')\end{aligned}$$

The definition of $\doteq_{drivGPU}$ and the confidentiality lemma are similar with above. For the proof of integrity, $\doteq_{encGPU}$ is used, and for the proof of confidentiality, $\doteq_{drivGPU}$ is used.

We formalise the context isolation with an observer *conGPU* and define $\doteq_{conGPU}$ as relating OM entries and the GPU global memory pages that look the same to the GPU context when they are outside its context *conGPU*:

DEFINITION 2 (Equivalence of GPU context $\doteq_{conGPU}$). OM entries m_1, m_2 are related by $m_1 \doteq_{conGPU} m_2$ *iff*:

$$\begin{aligned}(m_1.\text{GM}? \wedge m_2.\text{GM}?) \vee (m_1.State_{gpu}? \wedge m_2.State_{gpu}?)\\ \vee(m_1.\text{contextID}? \wedge m_2.\text{contextID}?)\\ \vee(m_1.\text{enclaveID}? \wedge m_2.\text{enclaveID}?)\end{aligned}$$

We formalise GPU context isolation property as:

THEOREM 2 (ISOLATION). Let the execution of EPT exception handler or IPI handler beginning in state(s, m) and returning in state(s′, m′) be denoted as handler(s, m, s′, m′). Then,

$$\begin{aligned}\forall(s_1, m_1), (s_2, m_2), (s_1', m_1'), (s_2', m_2').(s_1, m_1) \doteq_{conGPU} (s_2, m_2)\\ \wedge handler(s1, m1, s1', m1') \wedge handler(s2, m2, s2', m2')\\ \implies (s_1', m_1') \doteq_{conGPU} (s_2', m_2')\end{aligned}$$

Proof. Similar to the 2-threaded version of the semantics bi-simulation in GPUVerify [12], to verify the multi-core concurrent execution, we reason about two GPU I/O executions beginning from initial states that are related by $\doteq_{encGPU}$, $\doteq_{drivGPU}$, and $\doteq_{conGPU}$ and we proof that the final states are also related by $\doteq_{encGPU}$, $\doteq_{drivGPU}$, and $\doteq_{conGPU}$. Like GPUVerify, we use a predicated execution technique similar with [12] to reduce concurrent execution to an analysis of a sequential program. Similar with the lock-step execution semantics in GPUVerify, our proof is structured into each smaller step proof for each EPT trapping and async-hypercall.

We use Dafny [33] with the help of Z3 SMT solver [38] to verify the specification of GEVISOR by specifying the predication and dualisation semantics. We first turn the invariants into predicated forms and then prove that MRtable and OM maintain the invariants. Furthermore, we encode the dualization semantics in Dafny and prove that the lemmas guarantee the confidentiality, integrity and GPU context isolation.

Besides the formal verification of GEVISOR design using Dafny, we implement GEVISOR following the verified design specification and further model check the GEVISOR implementation using CBMC [18]. However, the challenge is noninterference itself is not a logical property, which cannot be directly encoded into the temporal logic, because it refers to a relation between two program executions. As with previous method [24, 48], we logically formulate noninterference as *self-composition* [10], which composes the original program with a copy that all variables is renamed to avoid execution of a correlative pair. In particular, it reduces the noninterference information flow policies to a safety property: the noninterference of a program P reduces to a property about single program executions of the program P;P', where P' is a renamed copy of P. We take the above semantic-based approach by self-composing GPU I/O execution and specify noninterference through assertions. Then we verify this assertion code and GEVisor via model checker CBMC automatically. A tricky part is the software page table walker that contains loops that iterate over entire MRtable, which cannot be verified by bounded CBMC automatically. For these special constructs, we manually audit the code. As a result, we achieve **G3** for both the design and implementation of GEVisor.

8 IMPLEMENTATION

We implement GEVisor using the Bareflank [3] hypervisor, and GPU runtime using the Gdev [28].

Dynamic Hypervisor Measured Launch. There are two challenges to implement a dynamic hypervisor measured launch mechanism. First, current measured launch environment (e.g., TBoot) is an early-boot method relying on the grub bootloader with the multiboot protocol to boot itself, which conflict with our late-launched hypervisor design. Second, to launch OS/VMM modules as well as SINIT ACM module, the kernel launch procedure of TBoot also relies on the multiboot protocol to read images from disk. To solve the above challenges, we transform the TBoot binary into a new ELF-formatted binary, *tboot.elf*, to strip out the debugging symbols and the multiboot header. Then we parse the ELF format to read in each segment within *tboot.elf* and copy it to the correct location within the physical memory (e.g.,0x0804000), and get the entry point of TBoot. For SINIT ACM, we copy it to the memory location above the TXT heap to avoid the potential buffer overflow attack. Similarly, we implement GEVisor as an ELF, which can be parsed by TBoot to get the entry point and jump to it with a *jmp* instruction.

Enclave Monitoring. For enclave creation and termination, we trap EINT and EREMOVE leaf instructions of ENCLS instruction by setting the ENCLS-exiting-bitmap. For enclave entry and exit, we cannot trap them since ENCLU is an unprivileged instruction. Instead, we instrument the GPU runtime to capture these events by adding an exit signal function before OCALL and a resume signal function after ECALL. These two signal functions communicate with GEVisor via CC.

Prog.	OP	SP	VCC	Vars	CLS	Ts	T	M
GEVisor	192	183	19	9K	21K	6	16	32
GEVisor_M	194	184	21	9k	21k	7	18	32
GEVisor_L	192	183	19	9k	21k	6	17	32

Table 2: GEVisor verification results with CBMC. OP = number of assignments before slicing; SP = number of assignments after slicing; VCC = number of VCCs after simplification; Vars = number of variables in SAT formula; CLS = number of clauses in SAT formula; Ts = time (sec) taken by SAT solver; T = total verification time (sec); M = peak memory usage (MB).

Superpage Support. 4K-based page trapping incurs significant performance degradation. Since SGX currently does not support big page, we directly locate the EPC memory's physical address and size with the CPUID instruction to replace the page-table walking and use 2M-page granularity. For GPU I/O pages, we change both the page table walker and EPT from 4K to 2M page granularity directly.

Ring Buffer. We format the communication channel as a ring buffer data structure, while each buffer size is determined by the hypercall batch size. The enclave pushes hypercall requests to the ring buffer, while the popped hypercall are processed by IPI handler on a remote core for execution. A batch of hypercall requests occupies a slot, and they always have the same status (i.e., either all free or all busy). In order to issue a hypercall, the thread within the enclave has to look for an entry in one of its hypercall pages that contain a $free$ status. It then writes the hypercall ID and messages to the entry. Finally, the status field is changed to *busy*, indicating to GEVisor that the hypercall is ready for execution. The enclave core continues to look for another open slot (with $free$ status) in an increasing and cyclic order on the ring buffer. The increasing and cyclic order of the hypercall requests and the single-threaded request and execution on CPU cores ensure the push and pop actions do not have data race.

9 EVALUATION

In this section, we evaluate the security aspect of GEVisor regarding its TCB size and verifiability (§9.1), the performance impact of GEVisor on various GPU-accelerated applications (§9.2), and the sources of GEVisor overhead on applications for GPU acceleration protection and remote attestation (§9.3). Our experimental machine uses an Intel i7-8700K 4.7GHz CPU with Intel SGX (SDK v2.0) and 6 cores, a TPM (v1.2), and 32GB main memory. For hardware acceleration, we use a NVIDIA GeForce GTX TITAN Black GPU with 2,880 CUDA cores and 6,144MB GDDR5 384-bit memory. We use Ubuntu 18.04.4 LTS 64-bit with Linux kernel 4.15.0 as the OS.

App	Data size	DMA regions (4K/2M/3M)
Back Propagation (bp)	117MB	29952/59/39
Breadth-First Search (bfs)	46MB	11776/23/16
LU Decomposition (LUD)	16MB	4096/8/6
SRAD	24MB	6144/12/8
Gaussian Elimination (GS)	32MB	8192/16/11
Hotspot (HS)	8MB	2048/4/3
Needleman-Wunsch (NW)	128MB	32768/64/43
K-nearest Neighbors (NN)	334KB	84/1/1
Pathfinder (PF)	256MB	65536/128/86

Table 3: List of Rodinia benchmark applications.

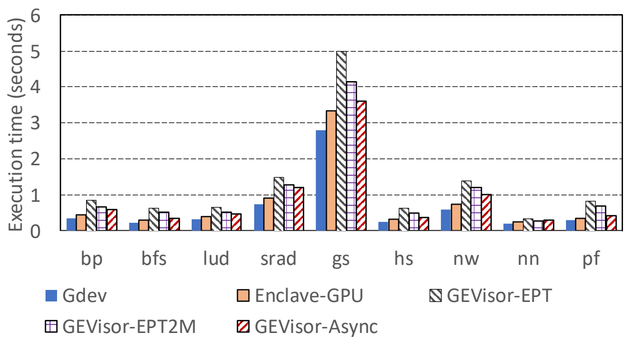


Figure 10: Execution time of the Rodinia benchmarks.

9.1 Code Size and Verification

Verification. Our implementation of GEVISOR only has 3.8K LoC allowing the hypervisor to be verifiable. To demonstrate the verifiability of GEVISOR, we ran the CBMC [18] model checker on the GEVISOR code with assertion added. CBMC was able to slice away unreachable code and unroll all the (bounded) loops successfully. Table 2 summarizes the results of the verification. We also manually injected two errors into GEVISOR to see if CBMC correctly identifies them. GEVISOR_M is our hypervisor with an unallocated pointer dereference error. GEVISOR_L contains a logical error that violates an assertion statement that we inserted. In all cases, CBMC was able to verify our hypervisor successfully.

9.2 Performance of Applications

We compare the existing solutions based on their results from the literature (Table 1), since there existed no available system using SGX that we could run as a baseline to compare with GEVisor experimentally. Here, we use two baseline systems to evaluate the overhead of GEVISOR on various applications. Gdev uses an unmodified GPU runtime (Gdev [28]) without any GPU protection. Enclave-GPU uses an modified GPU runtime within enclave, but no GEVISOR protection. Using these baseline systems, we show the performance overhead of the applications protected by three different GEVISOR mechanisms in comparison: GEVISOR-EPT uses our EPT protection with the default page size (4KB); GEVISOR-EPT2M is with the EPT protection with 2MB superpages (§ 8); and GEVISOR-Async uses our asynchronous hypercall for GPU acceleration protection with 3MB DMA buffer. All of the systems run an application in a SGX enclave.

GPU Application Benchmarks. Rodinia [14] is a benchmarking suite for heterogeneous computing, which contains a list of GPU kernels for CUDA. These benchmarks include machine learning and graph analytic algorithms. Table 3 presents the list of applications selected from the Rodinia benchmark suite, and the data amounts transferred between the CPU and GPU. GEVISOR reserves 3MB DMA buffer using the technique similar with Contiguous Memory Allocation (CMA) [4]. Thus, we calculate the number of DMA regions corresponding to 3MB DMA buffer for each application within the table, which is used by GEVISOR-Asyn. In addition, we also list the DMA regions numbers for GEVISOR-EPT (4K) and GEVISOR-EPT2M (2M). The nine benchmarks are chosen by considering diversity as well as completeness of program behavior: `gs` representing compute bound, `bp, bfs, srad, nw`, and `pf` for I/O bound, and `hs, lud`, and `nn` for small kernels. Fig. 10 presents the execution time of the benchmarks with the baseline and GEVISOR systems. For most benchmarks, GEVISOR-Async performs most efficiently with an average of 18% overhead compared to Enclave-GPU. For the computation-bound benchmark `gs`, the EPT protection mechanisms (GEVISOR-EPT and GEVISOR-EPT2M) yield lower performance degradation compared with other I/O bound applications, `bp, bfs, srad, nw`, and `pf`.

We analyze that for I/O-bound workload, our asynchronous hypercall improves multi-core utilization significantly as the message batch size increases dramatically. On the other hand, computation-bound workload, hs, lud, and especially nn, with small GPU kernels does not benefit from asynchronous hypercall: the performance becomes even worse due to the small amount of hypercall requests does not amortize the IPI overhead. We expect this performance overhead can be further reduced by employing a hardware platform with more number of cores.

Deep Learning Benchmarks. We use the Darknet [43] deep neural network framework to evaluate our system with more complex and realistic workloads. We configure 128 MB for EPC region and employ nine popular pre-trained models with the ImageNet [44] dataset. Fig. 11 compares the execution time of the models for prediction with and without GPU protection. We mark the values of execution time of Gdev, Enclave-GPU, and GEVISOR-Async above the bars. represent with unmodified Gdev, modified Gdev with Enclave, and async-hypercall monitoring methods respectively. Using Enclave-GPU as the baseline, our system with asynchronous hypercall (GEVISOR-Async) has only a 13.1% overhead on

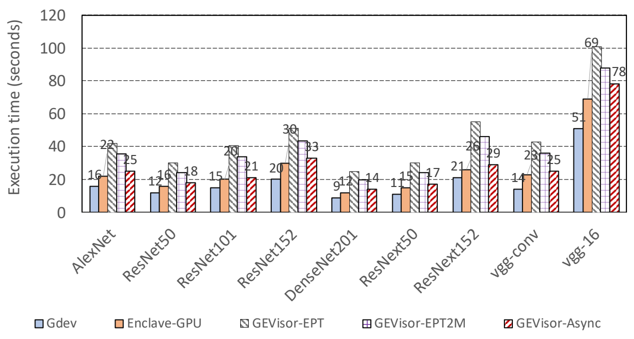


Figure 11: Execution time of the Darknet benchmarks.

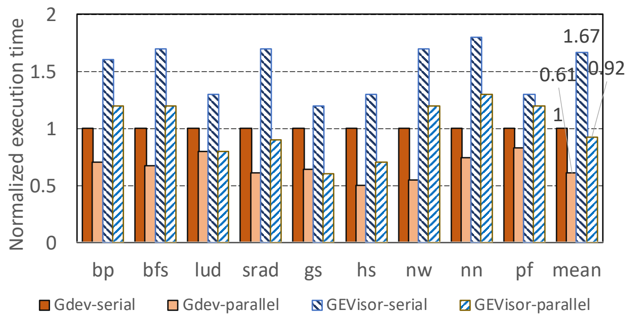


Figure 12: Performance of multiple parallel enclaves with the Rodinia benchmarks.

average, which is much better than an encryption-based solution (33% [51])

Parallel GPU Enclaves. We evaluate the performance of multiple enclaves sharing the same GPU simultaneously using the Rodinia benchmarks. The results with two parallel enclaves are shown in Fig. 12. We partition the input data evenly among two parallel enclaves. All execution times are normalized to that of Gdev running the benchmarks with unmodifed Gdev (Gdev-serial), and GEVISOR-serial and GEVISOR-parallel use asynchronous hypercall (GEVISOR-Async). GEVISOR with parallel enclaves (GEVISOR-parallel) demonstrates around 52% performance degradation compared to the parallel Gdev with unprotected GPU acceleration (Gdev-parallel), while it is still more performant than Gdev-serial. We find that this performance degradation is mainly due to the overhead from our GPU context isolation since GPU context switches happen frequently between the parallel enclaves during the execution.

9.3 Performance of GEVISOR Operations

We present benchmark results to analyze the sources of GEVISOR overhead and to evaluate the performance of our asynchronous hypercall and linear remote attestation.

	Init	MemcpyHtoD	Execute	MemcpyDtoH	Close
Enclave-GPU	171	5	301	30	2
GEVISOR-Async	180	10	301	60	2
Overhead	9	5	0	30	0

Table 4: CUDA operation microbenchmarks (in ms). MemcpyHtoD and MemcpyDtoH results are for 4MB data transfer equally.

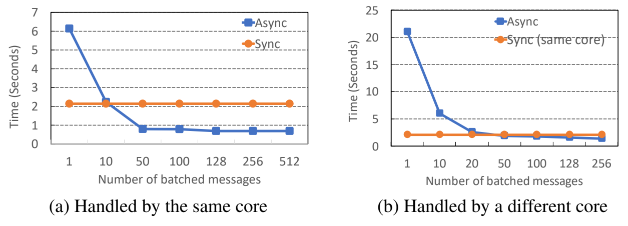


Figure 13: Asynchronous hypercall benchmarks.

Overhead on CUDA Operations. To understand the performance impact of GEVISOR on individual GPU operations, which is the cause of the overhead on the GPU application, we run a benchmark with a matrix multiplication application using CUDA. The application computes the multiplication of two 1042×1024 integer matrices (4MB for each matrix). Table 4 summarizes the overhead introduced by the key CUDA operations. The results show that GEVISOR causes overhead mainly on the GPU I/O from the GPU (MemcpyDtoH), but it has no performance impact on the GPU kernel computation (Execute) or the GPU runtime deinitialization (Close). The reason is that GEVISOR mainly affects the performance of I/O communication, especially on MemcpyDtoH because we protect the GPU context isolation by monitoring the GPU pages.

Asynchronous Hypercall. We evaluate the performance of asynchronous hypercall extensively in microbenchmarks by varying its batch size (Fig. 13). Fig. 13(a) compares the performance between synchronous (Sync) and asynchronous hypercalls (Async) when the hypercall is handled by the same processor core while Fig. 13(b) compares them when the hypercall is handled by another core with an inter-processor interrupt (IPI) that wakes up the core. We compare the two cases to verify if the direct costs are amortized when the number of batched calls varies. The results show that our asynchronous hypercall scales much better than synchronous hypercall as it does not increase the number of context switches (to and from the hypervisor) when the batch size increases in both cases. The performance of asynchronous hypercall starts to overtake that of synchronous hypercall when the batch sizes reach 10 and 23 for the same core and another core, respectively, and we find this batch size is very effective for applications with large DMA data transfering.

	Hardware	Extend	Seal	UnSeal	Quote
Enclave	SGX	2.43	2.20	4.06	10.99
Hypervisor	TPM	19.83	20.88	24.72	197.34

Table 5: Performance of linear remote attestation (in ms). We take an average of 10 runs with negligible variance.

Linear Remote Attestion. Table 5 summarizes the results of our microbenchmark on linear remote attestation. We show the latencies of each attestation operation within TPM and SGX separately.

10 DISCUSSION

Deployability. Given the emerging confidential VM technologies (e.g., AMD SEV-SNP [45] and Intel TDX [17]) that designed with an untrusted hypervisor in mind, the popularity of confidential VMs is taking over application-level enclaves in the cloud. GEVisor can be extended to Intel TDX and AMD SEV-SNP for VMs. We also deployed GEVISOR to a KVM-based VM environment. Based on our practice, we recommend using the GPU pass-through and EPT 1-to-1 mapping techniques to reduce the overhead of nested virtualization. For VMs sharing the same host-side EPT for all VCPUs, we need to add a VCPU ID within MRtable and flush the TLB of the requested VCPU core from outside the enclave, enforcing that only the enclave-executing VCPU can actually access GPU I/O during runtime.

Scalability. GEVISOR's current design only supports a single GPU, however, it should not impact the scalability of GEVisor. For multiple GPUs, each GPU will be assigned a unique GPUID in the MRtable and GEVisor will perform access control using GPUID with only a small overhead (e.g., a few milliseconds). For complex workloads, the bandwidth between CPU and GPU may become a bottleneck depending on the workload. Application-specific algorithms to optimize performance and scalability within an enclave (*e.g.*, deep learning [29]) are complementary to our system since GEVISOR does not require modifying the GPU application.

11 RELATED WORK

GPU TEE. Graviton [51] relies on architectural modification to the GPU to support TEE for GPU. Similarly, HIX [27] relies on hardware modification to the CPU including SGX hardware and PCIe routing. Meanwhile, HETEE [56] supports large-scale confidential computing using PCIe Express-Fabric to distribute computation over server nodes that are *physically* isolated. StrongBox [20] builds a GPU TEE for ARM Endpoints based on TrustZone with an integrated GPU, while GEVISOR is designed to support trusted GPU execution with SGX enclaves without hardware change.

Software Protection against Privileged Code. Overshadow [16], SP3 [52], InkTag [25], TrustVisor [36], Cloudvisor [53] and Virtual Ghost [19], etc. assume a trusted virtualization layer to protect applications from a privileged attacker. Existing works [31, 37, 46, 55] propose a hypervisor-based solution to provide trusted execution of generic external I/O devices for user-level programs. However, unlike GEVISOR these solutions are not designed with the limitations of SGX enclaves in mind.

Trusted Execution with Intel SGX. There have been several recent studies [11, 15, 42] to build trusted execution environments with Intel SGX to support generic applications without modification. Graphene-SGX [15] allows running an unmodified legacy application within an SGX enclave. Haven [11] shields execution of unmodified legacy applications on Windows operating system. SCONE [42] provides a container environment protected by Intel SGX. Different from these solutions, GEVISOR is designed to provide trusted GPU execution for any enclave implementations.

Formal Verification. Our verification technique is built based on GPUVerify [12, 26]. In comparison, GPUVerify mainly verifies the GPU kernels and focuses on two classes of bugs only (*i.e.*, data races and barrier divergence) while our work focuses on verifying the security properties (confidentiality, integrity, and context isolation) of I/O operations between the CPU and GPU for enclaves. There exist other formally verified systems for security: seL4 [30], ExpressOS [34], CertiKOS [23], Komodo [21] and uberXMHF [50]. Although GEVISOR, seL4, CertiKOS, and Komodo all verify security properties based on noninterference [22], GEVISOR is the first work to apply noninterference to GPU I/O protection to the best of our knowledge.

12 CONCLUSION

In this paper, we propose a formally verified reference monitor GEVISOR that cooperates with SGX enclave to build a GPU TEE without any hardware changes.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Yang Cao for their valuable comments and suggestions. This work was supported in part by the National Science Foundation (NSF) under Grant CNS-2145744, the Office of Naval Research (ONR) under Grant N00014-23-1-2157, the Texas A&M Engineering Experiment Station (TEES) on behalf of its SecureAmerica Institute, and Wistron. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of NSF or ONR.

REFERENCES

[1] 2010. Intel Inc. Intel trusted execution technology. www.intel.com/technology/security/

[2] 2012. Nouveau Open-Source Driver. http://nouveau.freedesktop.org/

[3] 2020. Bareflank Hypervisor SDK. http://bareflank.github.io/hypervisor/

[4] 2021. A deep dive into cma. https://lwn.net/Articles/486301/.

[5] 2022. Microsoft confidential cloud using Nvidia GPUs. https://www.microsoft.com/en-us/research/blog/powering-the-next-generation-of-trustworthy-ai-in-a-confidential-cloud-using-nvidia-gpus/

[6] 2022. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core

[7] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.

[8] Will Arthur, David Challener, and Kenneth Goldman. 2015. Platform security technologies that use TPM 2.0. In *A Practical Guide to TPM 2.0*. Springer, 331–348.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS operating systems review* 37, 5 (2003), 164–177.

[10] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252.

[11] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)* 33, 3 (2015), 1–26.

[12] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: a verifier for GPU kernels. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 113–132.

[13] Ian Buck. 2007. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*. 6–es.

[14] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[15] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[16] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *ACM SIGOPS Operating Systems Review* 42, 2 (2008), 2–13.

[17] 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. *arXiv preprint arXiv:2303.15540* (2023).

[18] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

[19] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual ghost: Protecting applications from hostile operating systems. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 81–96.

[20] Yunjie Deng, Chenxu Wang, Shunchang Yu, Shiqing Liu, Zhenyu Ning, Kevin Leach, Jin Li, Shoumeng Yan, Zhengyu He, Jiannong Cao, et al. 2022. StrongBox: A GPU TEE on Arm Endpoints. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 769–783.

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

[22] Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*. IEEE, 11–11.

[23] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. {CertiKOS}: An Extensible Architecture for Building Certified Concurrent {OS} Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 653–669.

[24] Jonathan Heusser and Pasquale Malacaria. 2010. Quantifying information leaks in software. In *Proceedings of the 26th Annual Computer Security Applications Conference*. 261–269.

[25] Owen S Hofmann, Sangman Kim, Alan M Dunn, Michael Z Lee, and Emmett Witchel. 2013. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 265–278.

[26] Dan Iorga, Alastair F. Donaldson, Tyler Sorensen, and John Wickerson. 2021. The Semantics of Shared Memory in Intel CPU/FPGA Systems. *Proceedings of the ACM Programming Languages* 5, undefined (2021).

[27] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2019)*. ACM, Providence, RI, 455–468. http://doi.acm.org/10.1145/3297858.3304021

[28] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 401–412. https://www.usenix.org/conference/atc12/technical-sessions/presentation/kato

[29] Kyungtae Kim, Chung Hwan Kim, Junghwan "John" Rhee, Xiao Yu, Haifeng Chen, Dave (Jing) Tian, and Byoungyoung Lee. 2020. Vessels: Efficient and Scalable Deep Learning Prediction on Trusted Processors. In *11th ACM Symposium on Cloud Computing (SoCC '20)*.

[30] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems (TOCS)* 32, 1 (2014), 1–70.

[31] Youngjin Kwon, Alan M Dunn, Michael Z Lee, Owen S Hofmann, Yuanzhong Xu, and Emmett Witchel. 2016. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 277–290.

[32] S. Lee, Y. Kim, J. Kim, and J. Kim. 2014. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *2014 IEEE Symposium on Security and Privacy*. 19–33. https://doi.org/10.1109/SP.2014.9

[33] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

[34] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. 2013. Verifying security invariants in ExpressOS. In *Proceedings of the eighteenth international conference*

on Architectural support for programming languages and operating systems. 293–304.

[35] Richard Maliszewski, Ning Sun, Shane Wang, Jimmy Wei, and Ren Qiaowei. 2015. Trusted boot (tboot).

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

[37] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2020. (Mostly) Exitless VM protection from untrusted hypervisor through disaggregated nested virtualization. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1695–1712.

[38] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[39] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. 2006. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal* 10, 3 (2006).

[40] Cong Nie. 2007. Dynamic root of trust in trusted computing. In *TKK T1105290 Seminar on Network Security*. Citeseer.

[41] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA leaks: a detailed hack for CUDA and a (partial) fix. *ACM Transactions on Embedded Computing Systems (TECS)* 15, 1 (2016), 1–25.

[42] PR Pietzuch, S Arnautov, B Trach, F Gregor, T Knauth, A Martin, C Priebe, J Lind, D Muthukumaran, D O'Keeffe, et al. 2016. SCONE: Secure Linux Containers with Intel SGX. USENIX.

[43] Joseph Redmon. 2013–2016. Darknet: Open Source Neural Networks in C. http://pjreddie.com/darknet/.

[44] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* 115, 3 (2015), 211–252.

[45] AMD Sev-Snp. 2020. Strengthening VM isolation with integrity protection and more. *White Paper, January* (2020), 8.

[46] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. 2009. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Washington, DC, USA) *(VEE '09)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/1508293.1508311

[47] T. Simonite. 2016. Intel puts the brakes on Moore's Law. https://www.technologyreview.com/s/601102/.

[48] Cong Sun, Liyong Tang, and Zhong Chen. 2009. Secure information flow by model checking pushdown system. In *2009 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*. IEEE, 586–591.

[49] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2015. Gpuvm: Gpu virtualization at the hypervisor. *IEEE Trans. Comput.* 65, 9 (2015), 2752–2766.

[50] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan McCune, James Newsome, and Anupam Datta. 2013. Design, implementation and verification of an extensible and modular hypervisor framework. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 430–444.

[51] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[52] Jisoo Yang and Kang G Shin. 2008. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 71–80.

[53] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the twenty-third acm symposium on operating systems principles*. 203–216.

[54] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems.. In *USENIX Security Symposium*, Vol. 20. 23.

[55] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. 2012. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy*. 616–630. https://doi.org/10.1109/SP.2012.42

[56] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. 2020. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1450–1465.