In Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)

Graviton: Trusted Execution Environments on GPUs

Stavros Volos
Microsoft Research

Kapil Vaswani
Microsoft Research

Rodrigo Bruno
INESC-ID / IST, University of Lisbon

Abstract

We propose Graviton, an architecture for supporting trusted execution environments on GPUs. Graviton enables applications to offload security- and performance-sensitive kernels and data to a GPU, and execute kernels in isolation from other code running on the GPU and all software on the host, including the device driver, the operating system, and the hypervisor. Graviton can be integrated into existing GPUs with relatively low hardware complexity; all changes are restricted to peripheral components, such as the GPU's command processor, with no changes to existing CPUs, GPU cores, or the GPU's MMU and memory controller. We also propose extensions to the CUDA runtime for securely copying data and executing kernels on the GPU. We have implemented Graviton on off-the-shelf NVIDIA GPUs, using emulation for new hardware features. Our evaluation shows that overheads are low (17-33%) with encryption and decryption of traffic to and from the GPU being the main source of overheads.

1 Introduction

Recent trends such as the explosion in volume of data being collected and processed, declining yields from Moore's Law [16], growing use of cloud computing, and applications, such as deep learning have fueled the widespread use of accelerators such as GPUs, FPGAs [37], and TPUs [20]. In a few years, it is expected that a majority of compute cycles in public clouds will be contributed by accelerators.

At the same time, the increasing frequency and sophistication of data breaches has led to a realization that we need stronger security mechanisms to protect sensitive code and data. To address this concern, hardware manufacturers have started integrating trusted hardware in CPUs in the form of trusted execution environments (TEE). A TEE, such as Intel SGX [28] and ARM Trustzone [1], protects sensitive code and data from system administrators and from attackers who may exploit kernel vulnerabilities and control the entire software stack, including the operating system and the hypervisor. However, existing TEEs are restricted to CPUs and cannot be used in applications that offload computation to accelerators. This limitation gives rise to an undesirable trade-off between security and performance.

There are several reasons why adding TEE support to accelerators is challenging. With most accelerators, a device driver is responsible for managing device resources (e.g., device memory) and has complete control over the device. Furthermore, high-throughput accelerators (e.g., GPUs) achieve high performance by integrating a large number of cores, and using high bandwidth memory to satisfy their massive bandwidth requirements [4, 11]. Any major change in the cores, memory management unit, or the memory controller can result in unacceptably large overheads. For instance, providing memory confidentiality and integrity via an encryption engine and Merkle tree will significantly impact available memory capacity and bandwidth, already a precious commodity on accelerators. Similarly, enforcing memory isolation through SGX-like checks during address translation would severely under-utilize accelerators due to their sensitivity to address translation latency [35].

In this paper, we investigate the problem of supporting TEEs on GPUs. We characterize the attack surface of applications that offload computation to GPUs, and find that delegating resource management to a device driver creates a large attack surface [26, 36] leading to attacks as page aliasing that are hard to defend without hardware support. Interestingly, we also find that architectural differences between GPUs and CPUs *reduce* the attack surface in some dimensions. For instance, all recent server-class GPUs use 3D-IC designs with stacked memory connected to GPU cores via silicon interposers [4, 11]. Unlike off-package memory connected to the CPU using copper-based traces on the PCB, which are easy to snoop and tamper, it is extremely hard for an attacker to open a GPU package and snoop on the silicon interconnect between GPU and stacked memory, even with physical access to the GPU. Thus, it is a reasonable assumption to include on-package memory within the trust boundary.

Based on these insights, we propose Graviton, an architecture for supporting TEEs on GPUs. In Graviton, a TEE takes the form of a *secure context*, a collection of GPU resources (e.g., device memory, command queues, registers) that are cryptographically *bound* to a public/private key pair and isolated from untrusted software on the host (including the driver) and all other GPU

contexts. Graviton guarantees that once a secure context has been created, its resources can only be accessed by a user application/runtime in possession of the corresponding private key. As long as the key is protected from the adversary (e.g., the key is hosted in a CPU TEE), the adversary cannot access the context's address space. Graviton supports two additional primitives: *measurement* for generating remotely verifiable summaries of a context's state and the platform, and *secure memory allocation and deallocation* for letting a device driver dynamically allocate and free memory without compromising security.

Graviton achieves strong security by redefining the interface between the GPU driver and the hardware. Specifically, we prevent the driver from directly accessing security sensitive resources, such as page directories, page tables, and other memory containing sensitive code and data. *Instead, we require that the driver route all resource allocation requests through the GPU's command processor.* The command processor tracks ownership of resources, and ensures that no resource owned by a secure context can be accessed by the adversary. The command processor also ensures that the resources are correctly initialized on allocation to a secure context, and cleaned up on destruction, preventing attacks that exploit improper initialization [23, 36, 52].

Our design has several key attributes including low hardware complexity, low performance overheads and crypto-agility. Graviton requires no changes to the GPU cores, MMU, or the memory controller. All changes are limited to peripheral components, such as the GPU command processor and the PCIe control engine; this is largely due to the assumption that on-package memory can be trusted. Graviton places no restrictions on the instruction set available within the TEE. We also show that a GPU runtime can use Graviton to build secure versions of higher-level APIs, such as memory copy, kernel launch, and streams, which can be used to build applications with end-to-end confidentiality and integrity.

We have evaluated our design on NVIDIA Titan GPUs, gdev [21], an open-source CUDA runtime, and nouveau [29], an open source GPU driver. In the absence of hardware that implements the proposed extensions, we implement and emulate the extensions using interrupts delivered to the host. Our evaluation using a set of representative machine learning benchmarks suggests that the overheads of running compute-bound GPU applications using secure contexts are low (17-33%) for the level of security we provide. The overheads are dominated by the cost of authenticated encryption/decryption of kernel launch commands and user data.

In summary, we make the following contributions.

- We propose Graviton, an architecture for supporting TEEs on accelerators, such as GPUs. Graviton provides strong security properties even against an adversary that might control the entire software stack on the host, including the accelerator driver.
- We define a threat model that places trust in the GPU hardware, including on-package memory.
- We propose a minimal set of extensions to the GPU hardware for implementing Graviton and show how these extensions can be used to design applications with end-to-end security guarantees. The design requires no changes to the GPU cores, MMU, or memory controller, resulting in low hardware complexity and low performance overheads.

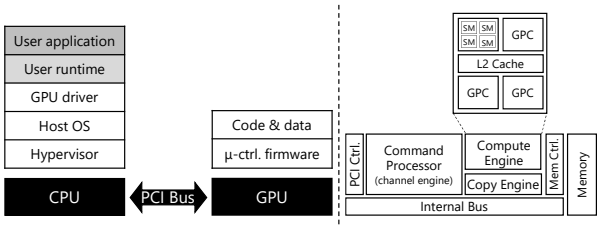


Figure 1: System stack (left) and hardware stack (right).

2 Background

2.1 GPU

We review the NVIDIA GPU architecture and the CUDA programming model to illustrate how a compute task is offloaded and executed on the GPU. We focus on the security critical parts of the architecture.

Software stack. A user-space application uses an API provided by the user-space GPU runtime (e.g., CUDA runtime), to program the GPU execution units with a piece of code known as the kernel, and transfer data between host and device memory. The GPU runtime converts each API call to a set of GPU commands for configuring the device and controlling kernel launches and data transfers. A GPU driver is responsible for submitting these commands to the GPU via the PCI bus and for managing device memory.

Hardware. The GPU (Figure 1) interfaces with the host CPU via the PCI control engine, which is connected with the rest of the GPU components via an internal bus. The key components are a command processor, compute and copy (DMA) engines, and the memory system, including the memory controller and memory chips.

The PCI control engine consists of (a) a PCI controller that receives incoming and outgoing PCI transactions, and (b) a master control engine, which exposes a set of memory-mapped-IO (MMIO) registers that are accessed by the host CPU to enable and disable the GPU engines.

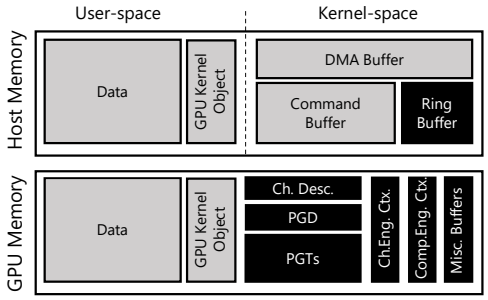


Figure 2: Host memory and GPU memory spaces.

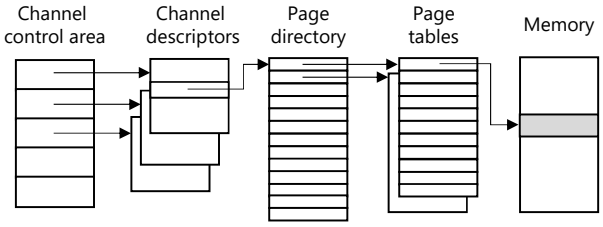


Figure 3: Channel-level address space management.

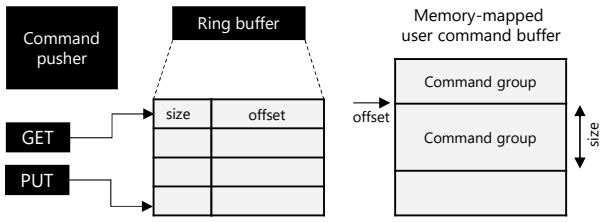


Figure 4: GPU command submission.

The command processor (aka channel engine) receives commands submitted by the device driver over as set of command queues known as *channels* and forwards them to the corresponding engines once they are idle. Channels are configured through a set of memory locations known as the *channel control area* which is mapped over the MMIO and serviced by the command processor.

The compute engine consists of a set of graph processing clusters (GPCs) and a shared L2 cache. Each GPC consists of a number of streaming multiprocessors (SMs), which are used to run GPU kernels. Each SM consists of multiple cores and a private memory hierarchy, including a read-only cache, L1 cache, and application-managed memory. GPU kernels specify the number of threads to be created, organized into thread blocks and grids. Thread blocks are divided into *warps*, where each warp is a unit of scheduling on each SM. Threads belonging to the same thread block share the caches and the application-managed memory.

Modern GPUs support virtual memory via a memory controller with page table walkers for address transaction, and a hierarchy of TLBs. For example, in the NVIDIA Volta, the L1 cache is virtually addressed and the L2 is physically addressed. The GPU has a shared two-level TLB used while accessing the L2 cache [19].

Context and channel management. Execution on GPUs is context-based. A CUDA context represent the collection of resources and state (memory, data, etc.) that are required to execute a CUDA kernel. Resources are allocated to contexts to run a compute task and are freed when a context is destroyed. Each context has its own address space. GPUs use channels to isolate a context's address space from other contexts. A channel is the only way to submit commands to the GPU. Therefore, every GPU context allocates at least one GPU channel.

To create a channel, the device driver allocates a *channel descriptor* and multi-level page tables in device memory (Figure 2 and 3). For example, a simple two-level page table consists of the page directory (PGD) and a number of leaf page tables (PGT). The driver writes the channel descriptor address to the channel control area, and the page directory address in the channel descriptor. The page directory consists of entries that point to leaf page tables, and leaf page tables contain virtual-to-physical mappings. Page tables typically support small (4KB) and big pages (128KB). The device driver updates all these data structures over the PCI bus via BARs.

Once the channel has been created, the device driver allocates device memory for a few channel-specific data structures, including (a) the internal context of the channel and compute engines, (b) a fence buffer used for synchronization between the host CPU and GPU, and (c) an interrupt buffer for notifying the host with interrupts generated by the GPU engines.

Command submission. The command processor is responsible for fetching commands submitted by the software stack and relaying them to the appropriate GPU engines. Figure 4 shows the data structures used for command submission. The driver allocates two buffers in kernel space, a command and a ring buffer. The command buffer is memory-mapped to the user space. The runtime pushes groups of commands to the command buffer, updates the channel's ring buffer with the size and offset of each group, and then updates over MMIO a register called the PUT register with a pointer to the command group. When the PUT register is updated, the command processor fetches a command group from the buffers, and updates the GET register to notify the runtime that the commands have been fetched.

Programming model. Next, we present an overview of the main stages of dispatching kernels to the GPU.

Initialization. An application wishing to use the GPU first creates a CUDA context. During context creation,

the runtime allocates a DMA buffer for data transfers between host memory and device memory (Figure 2). Subsequently, the application loads one or more CUDA modules into the context. For each kernel defined in the module, the runtime creates a corresponding *kernel object* on the GPU by allocating device memory for (a) the kernel's code, (b) constant memory used by the kernel, and (c) local memory used by each thread associated with the kernel. The runtime then copies code and constant memory to device memory via DMA.

Memory allocation. The application allocates device memory for storing inputs and outputs of a kernel using a memory allocation API (`cudaMalloc` and `cudaFree`). Memory allocations are serviced by the driver, which updates the page directory and page tables.

Host-GPU transfers. When the application issues a host-to-device copy, the runtime pushes a command group to the context's channel, passing the virtual addresses of source and destination to the copy engine. Once the engine is configured, it translates virtual addresses to physical ones and initiates DMA transfers.

Kernel dispatch. When the application executes a kernel, the runtime passes a command group to the command processor that includes the kernel's context, the base address of the code segment, the entry program counter, the grid configuration, and the kernel's environment, which includes the stack and parameters values. The command processor uses these parameters to initialize compute engines, which in turn initialize and schedule the computation on GPU cores.

Sharing. A GPU can be used to execute multiple kernels from multiple host processes using techniques such as pre-emptive multi-tasking [33, 42], spatial multi-tasking [2, 34], simultaneous execution [47], multi-process service [30], or virtualization [25]. In such scenarios, it is the responsibility of the host (driver) to isolate kernels using the channel abstraction and virtual memory. While spatial multi-tasking advocates for SM partitioning, it still shares memory resources and relies on virtual memory for isolation. Even in devices that support SR-IOV and partition resources in hardware (e.g., AMD MxGPU), system software is still responsible for assigning virtual devices to virtual machines.

2.2 Intel SGX

Trusted execution environments, or *enclaves* (e.g., Intel SGX) protect code and data from all other software in a system. With OS support, an untrusted hosting application can create an enclave in its virtual address space. Once an enclave has been initialized, code and data within the enclave is isolated from the rest of the system, including privileged software.

Intel SGX enforces isolation by storing enclave code and data in a data structure called the Enclave Page Cache (EPC), which resides in a pre-configured portion of DRAM called the Processor Reserved Memory (PRM). The processor ensures that any software outside the enclave cannot access the PRM. However, code hosted inside an enclave can access both non-PRM memory and PRM memory that belongs to the enclave. SGX includes a memory encryption engine that encrypts and authenticates enclave data evicted to memory, and ensures integrity and freshness.

In addition to isolation, enclaves also support *remote attestation*. Remote attestation allows a remote challenger to establish trust in an enclave. In Intel SGX, code hosted in an enclave can request a *quote*, which contains a number of enclave attributes including a measurement of the enclave's initial state. The quote is signed by a processor-specific attestation key. A remote challenger can use Intel's attestation verification service to verify that a given quote has been signed by a valid attestation key. The challenger can also verify that the enclave has been initialized in an expected state. Once an enclave has been verified, the challenger can set up a secure channel with the enclave (using a secure key exchange protocol) and provision secrets such as encrypted code or data encryption keys to the enclave.

3 Threat Model

We consider a strong adversary who controls the entire system software, including the device drivers, the guest operating system, and the hypervisor, and has physical access to all server hardware, including the GPU. Clearly, such an adversary can read and tamper with code or data of any victim process. The adversary can also access or tamper with user data in DMA buffers or with commands submitted by the victim application to the GPU. This gives the adversary control over attributes, such as the address of kernels being executed and parameters passed to the kernel. The adversary may also access device memory directly over MMIO, or map a user's GPU context memory space to a channel controlled by the adversary. In multi-tasking GPUs, malicious kernels can be dispatched to the GPU, thereby accessing memory belonging to a victim's context. These attacks are possible even in a virtualized environment (e.g., even if a device supports SR-IOV) because the mapping between VMs and virtual devices is controlled by the hypervisor.

An adversary with physical access to the server can mount snooping attacks on the host memory bus and the PCIe bus. However, we do trust the GPU and CPU packages and firmware, and assume that the adversary cannot extract secrets or corrupt state within the packages. This implies that we trust CPUs to protect code and data hosted inside TEEs. Side-channel attacks (e.g., based on speculative execution, access patterns and timing) and

denial-of-service attacks are also outside the scope of this paper. Side channels are a serious concern with trusted hardware [10, 12, 22, 38, 45, 50] and building efficient counter measures remains an open problem. In Graviton, we use TEEs to host the user application and the GPU runtime.

Unlike host memory, which is untrusted, we trust on-package GPU memory as GPU cores are attached to memory using silicon interposers, which make it extremely difficult for an attacker to mount snooping or tampering attacks. There is an emerging class of attacks on stacked integrated circuits (ICs), such as attacks where the package assembler inserts a trojan die between the GPU and memory dies [49]. Developing mitigations for these attacks is ongoing work [3] and outside the scope of this paper.

Even under this threat model, we wish to guarantee confidentiality and integrity for applications that use GPUs. Specifically, we wish to guarantee that the adversary cannot observe or tamper with code, data, and commands transferred to/from the GPU by a trusted application that runs in a CPU TEE or an on-premise machine. Finally, we wish to guarantee that the GPU computation proceeds without interference from the adversary.

4 Overview

Consider a CUDA application (Figure 5) that performs matrix multiplication, which is a key building block in machine learning algorithms. The application creates a new CUDA context (implicitly on the first CUDA API call), allocates memory for input and output matrices in host and device memory, populates the matrices, and then invokes the matrix multiplication kernel on the GPU (not shown), passing pointers to device memory and other kernel's parameters. After the kernel has completed, the application copies the results into host memory and releases memory allocated on the GPU.

As described earlier, an attacker with privileged access to the server can easily recover the contents of the matrices and the result even if this application is hosted in a CPU enclave. We can harden this application against such attacks simply by linking it against Graviton's version of the CUDA runtime. Graviton's version of the runtime creates a secure context (instead of a default context) on a Graviton-enabled GPU. In this process, the runtime authenticates the GPU and establishes a secure session with the GPU's command processor, with session keys stored in CPU enclave memory.

The runtime also provides a custom implementation of `cudaMalloc`, which invokes the device driver to allocate GPU memory and additionally verifies that allocated memory is not accessible from any other context or from the host. The secure implementation of `cudaMemcpy` ensures that all transfers between host and the GPU, including code and data, are encrypted and authenticated using keys inaccessible to the attacker. The implementation of `cudaLaunch` sends encrypted launch commands to the GPU's command processor over a secure session. Finally, the implementation of `cudaFree` authorizes the GPU's command processor to unmap previously allocated pages from page tables, and scrubs their content, enabling the driver to reuse the pages without leaking sensitive data.

```
int main() {
  ...
  float* h_A = malloc(M*N*sizeof(float));
  float* h_B = malloc(N*K*sizeof(float));
  float* h_C = malloc(M*K*sizeof(float));
  float* d_A, d_B, d_C;
  ...
  cudaMalloc((void**)&d_A, M*N*sizeof(float)
      );
  ...
  populate_matrices(h_A, h_B);
  cudaMemcpy(d_A, h_A, M*N*sizeof(float),
      cudaMemcpyHostToDevice);
  cudaMemcpy(d_B, h_B, N*K*sizeof(float),
      cudaMemcpyHostToDevice);
  ...
  matrixMul<<<grid,threads>>>(d_C,d_A,d_B,M,
      N,K);
  ...
  cudaMemcpy(d_C, h_C, M*K*sizeof(float),
      cudaMemcpyDeviceToHost);
  cudaFree(d_A);
  ...
}
```

Figure 5: Sample CUDA application.

5 Graviton Architecture

In this section, we describe extensions to existing GPU architectures for supporting secure contexts.

5.1 Remote Attestation

A Graviton-enabled GPU supports remote attestation for establishing trust between a secure context and a remote challenger. Hardware support for attestation is similar to TPMs; we require (a) a secret, known as the root endorsement key (EK), to be burned into the device's e-fuses during manufacturing and (b) a cryptographic engine for asymmetric key generation and signing. The EK is the root of trust for attestation and never leaves the GPU package. During boot, the GPU generates a fresh attestation key (AK) pair and stores the private key securely within the command processor. The GPU also signs the public part of the AK with the EK and makes it available to the device driver, which in turn sends the

signed AK to a trusted CA. The CA validates the signature using a repository of public endorsement keys provisioned by the manufacturer and generates a signed AK certificate. The certificate is stored by the device driver and used during secure context creation to prove to a challenger that the GPU holds and protects the private attestation key.

5.2 Secure Context Management

In Graviton, a secure context consists of one or more *secure channels*. We extend the GPU's command processor with new commands for creation, management, and destruction of secure channels (Figure 6).

A secure channel is created using the command `CH_CREATE`, which requires as parameters a channel identifier and a public key UK_{pub}. On receiving the request, the command processor generates a fresh channel encryption key (CEK) for encrypting commands posted to this channel. The public key UK_{pub}, CEK, and a counter are stored in a region of device memory accessible only to the command processor. `CH_CREATE` may be used to create multiple channels associated with the same secure context by passing the same UK_{pub}, in which case all such channels will use the same CEK.

After generating the CEK, the command processor establishes a *session* by securely transferring the CEK to the trusted user-space runtime. The command processor encrypts the CEK with UK_{pub} and generates a *quote* containing the encrypted CEK and a hash of UK_{pub}. The quote contains the channel identifier and security critical platform-specific attributes, such as the firmware version, and flags indicating whether preemption and debugging are enabled. The quote is signed using AK. The device driver passes this quote and the AK certificate (obtained during initialization) to the user-space runtime. The runtime authenticates the response by (a) verifying the AK certificate, (b) verifying the quote using the public AK embedded in the certificate, and (c) checking that the public key in the quote matches UK_{pub}. The runtime can then decrypt the CEK and use it for encrypting all commands sent to the GPU.

Once a session has been established, the command processor authenticates and decrypts all commands it receives over the channel using the CEK. This guarantees that only the user in possession of the CEK can execute tasks that access the context's address space. We use authenticated encryption (AES in GCM mode) and the per-channel counter as IV to protect commands from dropping, replay, and re-ordering attacks. This ensures that all commands generated by the GPU runtime are delivered to the command processor without tampering.

5.3 Secure Context Isolation

In existing GPUs, the responsibility of managing resources (e.g., device memory) lies with the device driver. For example, when allocating memory for an application object, the driver determines the virtual address at which to allocate the object, then determines physical pages to map to the virtual pages, and finally updates virtual-physical mappings in the channel's page tables over MMIO. This mechanism creates a large attack vector. A compromised driver can easily violate channel-level isolation—e.g., by mapping a victim's page to the address space of a malicious channel.

One way of preventing such attacks and achieving isolation is to statically partition resources in hardware between channels. However, this will lead to under-utilization of resources. Moreover, it prohibits low-cost sharing of resources between channels, which is required to implement features, such as streams. Instead, Graviton guarantees isolation by imposing a strict *ownership* discipline over resources in hardware, while allowing the driver to dynamically partition resources.

More formally, consider a physical page P that is mapped to a secure channel associated with a secure context C and a channel encryption key CEK, and contains sensitive data. We consider any object (code and data) allocated by the application in a secure context and all address space management structures (i.e., channel descriptor, page directory and page tables) of all channels as sensitive. We propose hardware changes to a GPU that enforce the following invariants, which together imply isolation.

Invariant 5.1 *P cannot be mapped to a channel associated with a context $C' \neq C$.*

Invariant 5.2 *P cannot be unmapped without authorization from the user in possession of CEK.*

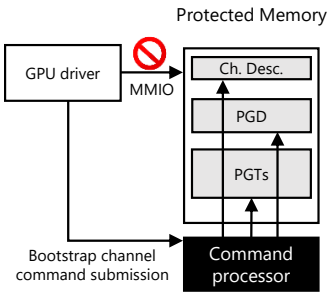
Invariant 5.3 *P is not accessible over MMIO to untrusted software on the host CPU.*

Invariant 5.4 *P is cleared before being mapped to another channel associated with a context $C' \neq C$.*

In the rest of this section, we describe hardware extensions for enforcing these invariants, and discuss their implications on the driver-GPU interface.

Memory regions. Our first extension is to partition device memory into three regions: *unprotected*, *protected* and *hidden*, each with different access permissions.

The *unprotected* region is a region in memory that is both visible from and accessible by the host via PCI BAR



Command	Attributes	Description
CH_CREATE	chid, pgd_address, public_key → quote	Sets page directory address within channel descriptor Returns quote for secure channels
CH_DESTROY	chid, MAC	Unsets channel descriptor and page directory Expects a keyed MAC of command for secure channels
CH_PDE	chid, pgd_index, PDE	Sets page table address in a page directory entry
CH_PTE	chid, pgd_index, pgt_index, size, PTE[], MAC → summary, MAC	Sets VA-PA mapping in page table entry Expects keyed MAC for user-authorized deallocation Returns a summary along with a MAC for secure allocations
CH_MEASURE	chid, address, size → measurement, MAC	Returns a measurement of a range of virtual addresses mapped in a secure channel along with a MAC

Figure 6: Commands for configuring a channel's address space and measuring the address space. PDE and PTE refer to page directory and page table entries respectively, and a MAC is a keyed message authentication code.

registers. The driver can use this region to allocate non-sensitive memory objects (e.g., synchronization and interrupt buffers) that are accessed over MMIO. This region can also be accessed from the GPU engines.

The *protected* region is visible to but not accessible from the host. The driver can allocate objects within the region (by creating page mappings), but cannot access the region directly over MMIO. Thus, this region can only be accessed from the GPU engines.

The *hidden* region is not visible or accessible to host CPU or to the GPU engines. Memory in this region are not accessible over PCI and are not mapped into any channel's virtual address space. This region is exclusively reserved for use by the command processor for maintaining metadata, such as ownership state of protected memory pages and per-channel encryption keys.

Regions can be implemented using simple range checks on MMIO accesses in the PCI controller and on commands that update address-space management structures in the command processor. The size of each region can be configured during initialization by untrusted host software. The size does not affect security; only availability as the system administrator could block creation of secure contexts by allocating a small protected region.

Address-space management. The next set of extensions are designed to enforce Invariant 5.1 and Invariant 5.2. We achieve this by decoupling the task of allocating and deallocating virtual and physical memory from the task of managing device-memory-resident address translation data structures (i.e., page directories and page tables) and delegating the latter to the GPU's command processor. In particular, we allow the driver to decide *where* in virtual and physical memory an object will reside, but require that the driver route requests to update page directories and page tables through the command processor using the API described in Figure 6.

The implementation of the API in the command processor enforces these invariants by tracking *ownership* of physical pages in the protected region in a data structure called the *Protected Memory Metadata* (PMM). We first describe PMM and then the commands.

Tracking ownership. The PMM is a data structure located in hidden memory, making it invisible to the host. It is indexed using the physical address of a memory page. Pages are tracked at the granularity of a small page (i.e., 4 KB). The PMM maintains the following attributes for each physical page.

- The attribute *owner_id* is the identifier of the channel that owns the page.
- The attribute $state \in \{\texttt{FREE}, \texttt{MAPPED}\}$ represents whether the page is free or already mapped to some channel. The initial value is `FREE`.
- The attribute `refcnt` tracks the number of channels a physical page has been mapped to.
- The attribute $lock \in \{\texttt{UNLOCKED}, \texttt{LOCKED}\}$ represents whether the page requires explicit authorization to be unmapped.
- The attribute *pgd_index* is an index into the page directory that points to the page table containing the mapping for the current page. Using this attribute, the command processor can reconstruct the virtual address of a physical page. In that sense, the PMM acts as an inverted page table for the protected region—i.e., stores $PA \rightarrow VA$ mappings.
- The attribute *pgt_entrycnt* is a 2-byte value that tracks the number of pages table entries allocated within a page table. Using this attribute, the command processor can know if a locked page table is empty and hence may be unmapped.

Assuming each PMM entry requires 64-bits, the total size of the PMM for a GPU with 6GB of physical memory is 12MB, which is $\sim 0.2\%$ of total memory.

Commands. The new commands for context and address space management use the PMM to enforce Invariant 5.1 and Invariant 5.2 as follows:

`CH_CREATE`**.** This command takes as a parameter the address of the page directory (*pgd_address*) for the newly created channel *chid*. It checks whether the channel descriptor and page directory are allocated on pages in the protected region, and that the pages are `FREE`. The former constraint ensures that after channel creation, the driver does not bypass the API and access the channel descriptor and page directory directly over MMIO.

If the checks succeed, the pages become `MAPPED` and the *owner_id* attribute of the pages is updated to the identifier of the channel being created. If a secure channel is being created (using a public key), the pages become `LOCKED`. The command processor then updates the address of the page directory in the channel descriptor, and clears the contents of pages storing the page directory to prevent an attacker from injecting stale translations. `CH_CREATE` fails if any of the pages containing the channel descriptor or the page directory is already `LOCKED` or `MAPPED` to an existing channel.

`CH_PDE`**.** This command unmaps an existing page table if one exists and maps a new page table at the index *pgd_index* in the page directory of the channel.

Before unmapping, the command checks if the physical pages of the page table are `UNLOCKED` or the *pgt_entrycnt* attribute is zero. In either case, the command decrements `refcnt`. If `refcnt` reaches zero, the pages become `FREE`. The command fails if the driver attempts to unmap a `LOCKED` page table or a page table with valid entries.

Before mapping a new page table, the command checks whether the page table is allocated on `FREE` pages in the protected region. If the checks succeed, the pages become `MAPPED`. Additionally, if the channel is secure, the pages become `LOCKED`. However, if these pages are already `MAPPED`, the command checks if the channel that owns the page (the current *owner_id*) and the channel that the page table is being mapped to belong to the same context by comparing the corresponding public key hashes. If the hashes match, the page's reference count is incremented. This allows physical page tables and hence physical pages to be shared between channels as long as they share the same context; this is required for supporting features such as CUDA streams [31]. If either of the checks succeed, the command creates a new entry in the page directory and clears the contents of the pages storing the page table. The command fails if the page table is mapped to a channel associated with a different context.

`CH_PTE`**.** This command removes existing mappings and creates new mappings (specified by PTE) for a contiguous range of virtual addresses of size *size* starting at *VA*, where *VA* is the virtual address mapped at index *pgt_index* by the page table at index *pgd_index* in the channel *chid*'s page directory. Before clearing existing page table entries, the command checks if the physical pages are `LOCKED`. To remove mappings for `LOCKED` physical pages, the command requires explicit authorization by the user runtime in the form of a MAC over the tuple $\{chid, VA, size\}$ using the CEK and a per-channel counter as the initialization vector (IV). The unforgeability of the MAC coupled with the use of a counter for IV ensures that a malicious driver cannot forge a command that unmaps physical pages allocated to secure channels, and then remapping them to other channels. If the checks and MAC verification succeed, the pages transition to `FREE`, and the page table entries are cleared.

Similarly, before creating new mappings, the command checks if the pages are `FREE`. Additionally, if the channel is secure, the command checks if the pages are located in the protected region (for sensitive code and data, discussed in Section 6). If the checks succeed, the page become `MAPPED` and if the page is being mapped to a secure channel, the pages become `LOCKED`.[1] If pages are already `MAPPED`, the command checks if the channel that owns the page (the current *owner_id*) and the channel that the page is being mapped to belong to the same context by comparing the corresponding public key hashes. On success, the command increments the *pgt_entrycnt* of the page table, updates the page table, and issues a TLB flush to remove any stale translations. While conventionally the latter is the responsibility of the device driver, in our design, the flush is implicit. The command fails if any of the pages are mapped to a channel associated with a different context.

When the command succeeds, it generates a *summary* structure, which encodes all $VA \rightarrow PA$ mappings created during the invocation of `CH_PTE`. The summary is a tuple $\{chid, VA, n, k, HASH(p\_1, ..., p\_n)\}$, where *VA* is the starting virtual address of the memory being allocated, n is the number of pages allocated in the protected region, k is the number of total pages allocated, and $p\_1, ...p\_n$ are addresses of protected physical pages. The command processor also generates a keyed MAC over this summary using the CEK. As described later, this summary is used by the runtime to verify that sensitive code and data are allocated in protected region.

`CH_DESTROY`**.** This command frees memory allocated to a channel by walking the page directory, finding physical pages owned by the channel, and resetting their en-

[1] Note that `CH_PTE` also permits pages in the unprotected region to be mapped to a secure channel; these pages can be accessed over MMIO and are used to store objects, such as fence buffers required by the driver for synchronization.

tries in the PMM. It then unmaps physical pages of the channel descriptor and the page directory, decrements `refcnt` for pages used for page tables, and pages become `FREE` if their `refcnt` reduces to 0.

For secure channels, the command requires explicit authorization in the form of a MAC over *chid* using the CEK and a per-channel counter as IV. However, the command processor also accepts unauthorized instances of this command; this enables the device driver to reclaim resources in scenarios where the user runtime is no longer able to issue an authorized command—e.g., due to a process crash.

In such a scenario, the command processor walks PMM to find physical pages mapped exclusively to the channel's address space, unmaps them, decrements their `refcnt` and clears their contents if their `refcnt` reduces to 0. The command processor also flushes all caches because memory accesses of the command processor do not go through the memory hierarchy of compute engines.

A malicious driver may misuse this mechanism to reclaim resources of a channel which is still in use, resulting in denial of service; there is not violation of confidentiality or integrity as pages containing sensitive information, including the channel descriptor, are scrubbed.

`CH_MEASURE` We extend the command processor with a command `CH_MEASURE` for generating a verifiable artifact that summarizes the contents of a secure channel. The artifact can be used to prove to a challenger (e.g., a GPU runtime) that a channel exists in a certain state on hardware that guarantees channel isolation. In our implementation, `CH_MEASURE` takes as parameters a range of virtual pages that should be included in the measurement. It generates a *measurement*, which contains a digest (HMAC) of the content of pages within the requested range, and a keyed MAC of the measurement using the CEK.

Bootstrapping. Introducing a command-based API for address-space management raises the following issue: *How does the driver send commands for managing the address space of secure channels without having access to the channel-specific CEK?* We overcome this by requiring the driver to use separate channels, which we refer to as *bootstrap channels*, for routing address-space management commands for all other channels. We allow the driver to create and configure one or more bootstrap channels over MMIO and allocate their address-space management structures in the unprotected region.

The command processor identifies a channel as a bootstrap channel by intercepting MMIO writes to the channel descriptor attribute in the channel control area. If the address being written to this attribute is in the unprotected region, the corresponding channel is marked as a bootstrap channel.

To ensure that the driver does not use a bootstrap channel to violate isolation of secure channels, the command processor prohibits a bootstrap channel from issuing commands to the copy and compute engines since such commands can be used to access sensitive state. The command processor also checks that all commands executed from a bootstrap channel are used to configure non-bootstrap channels. This prevents an adversary from allocating protected memory pages of a secure context as page directory and/or page tables for a bootstrap channel and then leveraging the `CH_PDE` and `CH_PTE` commands to tamper with the memory of the secure context.

Big page support. The virtual memory subsystem on modern GPUs employ multiple page sizes. For example, in NVIDIA GPUs, each page directory entry consists of two entries, one pointing to a small page (4KB) table, and another to a big page (128KB) table. Our design requires minor extensions to support large pages. In the PMM, we continue to track page metadata at the small page granularity, but we add a bit to each entry to indicate if the corresponding physical page was mapped to a small or big virtual page. In addition, we require an additional parameter in the `CH_PDE` and `CH_PTE` commands to specify whether the updates are to a small or big page table. Finally, these commands check that the same virtual page is not mapped to two different physical pages.

Error handling. When a command fails, the command processor writes the error in an SRAM register that is accessible by the device driver over MMIO. This allows the device driver to take necessary actions so as to guarantee consistent view of a channel's address space between the command processor and the device driver.

6 Software Stack

We now describe new CUDA primitives supported by the Graviton runtime that use secure contexts and enable design of applications with strong security properties.

Secure memory management. The Graviton runtime supports two primitives `cudaSecureMalloc` and `cudaSecureFree` for allocating and deallocating device memory in the protected region.

`cudaSecureMalloc` guarantees that allocated GPU memory is owned by the current context (Invariant 5.1) and lies within the protected region (Invariant 5.3). Much like the implementation of `cudaMalloc`, `cudaSecureMalloc` relies on the device driver to identify unused virtual memory and physical pages in device memory, and update page directory and page tables using the commands `CH_PDE` and `CH_PTE`. As described above, these commands implement checks to enforce Invariant 5.1. The runtime enforces Invariant 5.3 using the

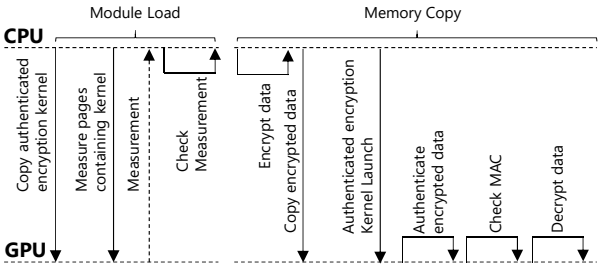


Figure 7: Secure memory copy protocol. The kernel is copied to the GPU during module creation.

summary structure generated by the CH_PTE command. In particular, the runtime uses the CEK and channel-specific counter to authenticate the summary structure(s) returned by the driver. The driver may return multiple summary structures in case the allocation spans multiple page tables. After authentication, the runtime can verify that memory objects are allocated in protected region using the attribute *n* in the summary.

cudaSecureFree first clears all allocated pages using a memset kernel. It then generates a MAC over the starting virtual address and size of the object using the CEK, and passes the MAC to the driver, which generates CH_PTE commands to remove entries from the page table. The MAC serves as an authorization to remove entries from the page table (Invariant 5.2). In the case where an object spans multiple page tables, the runtime generates one MAC per page table.

An implication of the redefined interface between the driver and the hardware is the inability of the driver to compact pages allocated to secure channels. Conventionally, the driver is responsible for compacting live objects and reducing fragmentation in the physical address space. However, Graviton prohibits the driver from accessing these objects. This can cause fragmentation in the protected region. We leave hardware support for compaction for future work.

Secure memory copy. The Graviton runtime supports the primitive cudaSecureMemcpy for securely copying code and data from the host TEE to device memory and vice versa. The protocol (Figure 7) works as follows.

1. After a secure context is created, the runtime allocates device memory using cudaSecureMalloc and copies a kernel that performs authenticated decryption (in clear text) into allocated memory, referred to as *AuthDec*. To ensure that the kernel was copied correctly without tampering, the runtime measures the region in device memory that contains the kernel (using CH_MEASURE), and checks the the returned digest matches the digest of the kernel computed inside the host TEE.
2. The implementation of cudaSecureMemcpy first encrypts data to be copied using a fresh symmetric key within the host TEE, and copies encrypted data to untrusted memory.
3. The runtime initiates a DMA to transfer encrypted data to target memory region. The command group that initiates the DMA is encrypted and integrity protected using the CEK.
4. The runtime uses the *AuthDec* kernel to decrypt data in device memory. It issues a command group to launch the kernel, passing the data's virtual address, the data encryption key, and the expected authentication tag as the kernel's parameters.
5. *AuthDec* authenticates encrypted data and generates an authentication tag which is checked against the expected authentication tag. If the check succeeds, the kernel decrypts the data in device memory, overwriting the encrypted data in the process.

A key attribute of secure memory copy is crypto-agility. Since the primitive is implemented fully in software, the runtime may support various encryption and authentication schemes without hardware changes.

Secure kernel launch. cudaSecureKernelLaunch uses secure memory copy to transfer the kernel's code and constant memory to the GPU, and then issues a command group to launch the kernel.

Recent GPUs have introduced preemption at instruction and/or thread-block boundaries. Extending our design to support preemption at the boundary of thread blocks is relatively straightforward because thread blocks are independent units of computation [42] and all ephemeral state, such as registers, application-managed memory, caches and TLBs can be flushed on preemption. Instruction-level preemption can also be supported by saving and restoring ephemeral state to and from a part of hidden memory reserved for each channel.

Secure streams. CUDA streams is a primitive used to overlap host and GPU computation, and I/O transfers. Each stream is assigned a separate channel, with each channel sharing the same address space, to enable concurrent and asynchronous submission of independent tasks. Our design supports secure streams (cudaSecureStreamCreate) by allowing channels within the same context to share page tables and pages. In particular, the runtime can remap a memory object to the stream's address space. Much like allocation requests, the driver uses CH_PTE command to update page tables. The runtime verifies that the $HASH(p\_1, ..., p\_n)$ generated by the CH_PTE command matches with the hash of the requested memory object.

7 Evaluation

7.1 Implementation

We implemented Graviton using an open-source GPU stack consisting of *ocelot*, an implementation of the CUDA runtime API [14], *gdev*, which implements the CUDA driver API [21], and *libdrm* and *nouveau*, which implement the user- and kernel-space GPU device driver [29]. Due to gdev's limitations (e.g., inability to use textures), we could not use some operations in cuBLAS (NVIDIA's linear algebra library) such as matrix-matrix (GEMM) and matrix-vector multiply (GEMV). Instead, we used implementations from Magma, an open-source implementation of cuBLAS with competitive performance [43]. Our implementation does not yet use SGX for hosting the user application and GPU runtime—porting the stack to SGX can be achieved using SGX-specific containers [5, 39, 44], and is outside the scope of this work.

Command-based API emulation. Since command processors in modern GPUs are not programmable, we *emulate* the proposed commands in software. Our emulator consists of (a) a runtime component which translates each new command and its parameters into a sequence of existing commands that triggers interrupts during execution, and (b) a kernel component, which handles interrupts, reconstructs the original command in the interrupt handler, and implements the command's semantics. The emulator uses the following commands: `REFCNT` sets the value of a 32-bit register in the channel control area which is readable from the host, `SERIALIZE` waits until previous commands are executed, `NOP` and `NOTIFY` triggers an interrupt when a subsequent `NOP` command completes.

Figure 8 shows the pseudo-code of the emulator along with an example for the `CH_CREATE` command. When a command is submitted, the runtime invokes the function *cmd_emu*, which translates each 32-bit value v in the command's original bit stream into the following sequence of commands: `REFCNT` with v as the parameter, `SERIALIZE`, `NOTIFY`, and `NOP`. This sequence is pushed into the ring buffer (using *push_ring_emu*), from where it is read by the command processor. When the command processor executes this sequence, it raises an interrupt, and an interrupt handler (*interrupt_handler*) is called on the host. The handler implements a state machine that reads the register in the channel control area and reconstructs the original command one value at a time. After reconstructing the entire command, the emulator implements its semantics (in this case using *chcreate_emu*) using reads and writes to device memory over MMIO (not shown in the figure).

We choose this emulation strategy because it allows us to run the software stack *as if* we had hardware support. Furthermore, it gives us a conservative approximation of performance because every command processor access to the PMM and memory mapping data structures in device memory translates into an access from the host to device memory over PCIe.

```
u32* ring_buf; /* ring buffer for context */
void push_ring(val) {
  *ring_buf=val;
  ring_buf++;
}

void push_ring_emu(u32 value) {
  push_ring(REF_CNT);
  push_ring(value);
  push_ring(SERIALIZE);
  push_ring(interrupt_buffer_addr >> 32);
  push_ring(interrupt_buffer_addr);
  push_ring(NOTIFY);
  push_ring(NOP);
}

void cmd_emu(u32 cmd, u32 param[], u32 size)
    {
  ...
  push_ring_emu(cmd);
  for (int i=0; i<size; i++)
    push_ring_emu(param[i]);
  ...
}

u32 chcreate(ctx_t *ctx, void *chan_base,
    void *pgd, u8 *pub_key) {
  ...
  cmd_emu(CMD_CHCREATE, param, 6);
  ...
}

void interrupt_handler(device_t *dev) {
  ...
  u32 val = mmio_rd32(dev);
  if (val != 0 || dev->cmd != 0) {
    if (dev->cmd == 0)
      dev->cmd = val;
    else {
      dev->param[dev->size++] = val;
      switch (dev->cmd) {
          ...
          case CMD_CHCREATE: {
            if (dev->size == 6) {
              chcreate_emu(dev);
              dev->size = 0;
              dev->cmd = 0;
            }
          } break;
  ...
}
```

Figure 8: Pseudo code for command emulation.

Command group authentication emulation. We also emulate the command processor logic for authenticated decryption of command groups. Our emulator intercepts encrypted command groups *before* they are copied to de-

vice memory and decrypts them. As we show later, this gives us a conservative approximation of performance since we decrypt command groups in software (aided by AES-NI) instead of a hardware encryption engine.

To estimate performance that can be achieved using a hardware encryption engine, we added a mode in our emulator which encrypts command groups on the host, but sends command groups in cleartext to the device, and adds a delay equal to the latency of decrypting the command group using a hardware engine. We compute the latency of decryption using the published latency of decrypting a block in hardware [40] and the size of the command group (in blocks).

Secure memory copy. Our implementation of secure memory copy combines AES and SHA-3 for authenticated encryption. We choose SHA-3 as its parallel tree-based hashing scheme is a good fit for GPUs. It also provides means for configuration (e.g., the number of rounds) allowing developers to explore different performance-security trade-offs [6, 7].

7.2 Performance Overheads

Testbed setup. For our evaluation, we used an Intel Xeon E5-1620-v3 server with 8 cores operating at 3.5 GHz, and two different NVIDIA GPUs: GTX 780 with 2304 CUDA cores operating at 863 MHz and GTX Titan Black with 2880 CUDA cores operating at 889 MHz. The general performance trends were similar with both GPUs. Therefore, we present results only for Titan Black. The host CPU runs Linux kernel 4.11 and uses CUDA toolkit v5.0 for GPU kernel compilation.

Command-based API. First, we quantify the overhead of using the command-based API using a matrix-matrix addition microbenchmark. Table 1 shows a breakdown of latency of command execution into five components: **Base**, which is the cumulative latency of all MMIO operations performed during command execution without any security extensions; **Inv.** which is the additional latency of invariant checks including PMM maintenance; **Init** which is the latency for initialization of page directory and page tables; **Crypto** which includes all required cryptographic operations; and interrupt handling, labeled as **Intr**. Note that the measured latency is obtained based on emulation, and therefore overestimates the latency that can be achieved with dedicated hardware support.

We find that the latency of `CH_CREATE` is dominated by the cost of initializing the page directory, and that of `CH_DESTROY` is dominated by the cost of walking the page directory to remove pages of locked page tables from PMM. For `CH_PDE`, we measure the latency for allocating a page directory entry for a small and big page tables. Allocating an entry for a small page table incurs a higher latency because the command needs to reset a larger number of entries. Finally, we measure the latency of `CH_PTE` for allocating an object spanning eight entries of a small page table or a big page table. Here, the latency is higher for big page tables because a larger number of invariant checks using the PMM, which tracks ownership at small page granularity.

Table 1: Command execution latency (μs)

Command	Base	Inv.	Init	Crypto	Intr
`CH_CREATE`	2	13	2246	11	58
`CH_DESTROY`	1	10592	N/A	23	68
`CH_PDE` (S)	1	104	3783	N/A	62
`CH_PDE` (B)	1	62	57	N/A	63
`CH_PTE` (S-8)	8	14	N/A	21	82
`CH_PTE` (B-8)	8	297	N/A	21	78

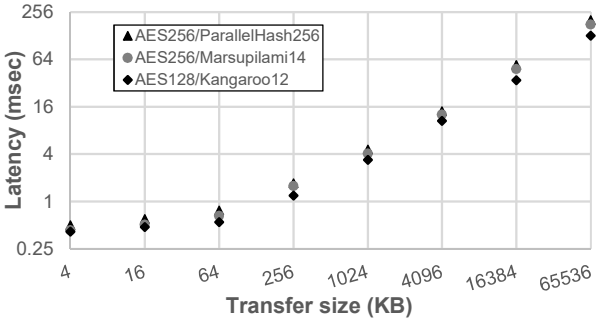


Figure 9: Secure memory copy performance for various sizes and configurations.

Secure memory copy. Figure 9 plots the latency of secure copy for three `AES/SHA3` variants and transfer sizes. The variants `ParallelHash256` and `Marsupilami14` provide 256-bit hashing strength while `Kangaroo12` provides 128-bit hashing strength. We find that latency remains flat for small transfer sizes and scales almost linearly for larger transfer sizes. Unless stated otherwise, we utilize the `AES256/Marsupilami14` configuration for the rest of the evaluation.

Table 2 shows a breakdown of the latency for `AES256/Marsupilami14` configuration. `Base` refers to the latency of normal (insecure) copy, and the other four components refer to the latency of executing AES and SHA-3 on the CPU and GPU. We find that as the transfer size increases, `SHA3-CPU` and `AES-GPU` account for a majority of the overheads (over 75% of the latency for 64MB transfers). For small data sizes, the `AES-GPU` phase, which is compute bound, under-utilizes the GPU cores and hence the execution time remains flat. In contrast, the `SHA3-GPU` kernel scales better due to lower algorithmic complexity. More generally, we attribute both

Table 2: Secure memory copy breakdown for AES256/Marsupilami14. Latency is reported in ms.

Size	4KB	64KB	256KB	4MB	64MB
Base	0.02	0.03	0.08	1.07	11.05
AES-CPU	0.01	0.05	0.10	1.46	25.45
SHA3-CPU	0.02	0.11	0.34	3.19	51.79
SHA3-GPU	0.12	0.13	0.12	0.31	0.58
AES-GPU	0.31	0.38	0.79	6.54	87.98
Total	0.46	0.70	1.43	12.57	176.87

Table 3: CUDA driver API latency (ms)

API	Normal	Secure
cuCtxCreate	77.65	252.63
cuCtxDestroy	17.00	29.43
cuModuleLoad	1.72	85.27
cuMemAlloc (S|B)	0.02|0.03	0.19|0.43
cuMemFree (S|B)	0.03|0.05	0.28|0.66

these costs to the lack of ISA support for SHA-3 on the CPU and for AES on the GPU.

CUDA driver API. Our implementation of secure extensions to the CUDA runtime API are based on extensions to the CUDA driver API. Table 3 shows the impact of adding security on latency of these APIs. As expected, all driver APIs incur higher latencies. The relatively high latency of secure version of context creation `cuCtxCreate` is dominated by the latency of creating an RSA key (75% of latency). The secure version of module load `cuModuleLoad` is more expensive because it (a) bootstraps secure copy, which measures the authenticated encryption kernels and (b) uses secure copy to transfer the application kernels to device memory. These APIs are typically used infrequently, and therefore these latencies do not have a large impact on overall execution time in most applications. On the other hand, `cuMemAlloc` and `cuMemFree` can be on the critical path for applications that use a large number of short-lived objects. The increased latency of these operations is predominantly due to emulation (interrupts and MMIO accesses). We expect an actual implementation of these operations on real Graviton hardware to incur much lower overheads with no involved interrupts and reduced memory access latency.

7.3 Applications

Finally, we quantify the impact of using secure CUDA APIs on end-to-end application performance using a set of GPU benchmarks. We use two benchmarks with different characteristics, namely Caffe, a framework for training and inference of artificial neural networks [18], and BlackScholes, an option pricing application [8].

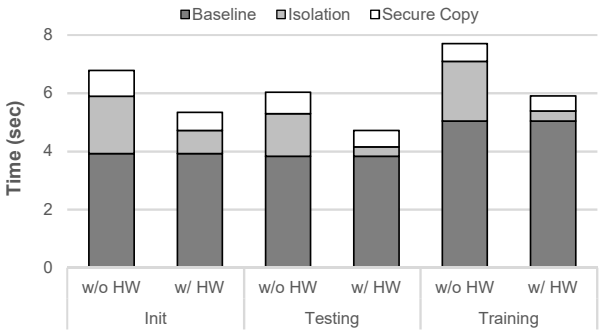


Figure 10: Cifar-10 performance. For training, time is reported for 25 batches averaged across all epochs. HW refers to a hypothetical hardware encryption engine used for command group authenticated decryption.

Cifar-10. We use Caffe to train a convolutional neural network on the Cifar-10 dataset, which consists of 60000 32x32 images spanning 10 classes. The network consists of 11 layers: 3 layers of convolution, pooling, rectified linear unit non-linearities (RELU) followed by local contrast normalization and a linear classifier. We run 40 epochs (each epoch is a sweep over 50000 images) with a batch size of 200 and test the model at the end of every epoch using 10000 images with a batch size of 400. Both the baseline system and Graviton achieve the same training accuracy.

Figure 10 shows Graviton's impact on execution time for three phases of execution—i.e. initialization, testing (which is similar to inference), and training. For training, execution time is reported for 25 batches averaged across all epochs. We also breakdown the overhead into two buckets, isolation (i.e., using the secure CUDA driver API and command group authentication) and secure memory copy. In the emulated environment, our security extensions cause a slowdown of 73%, 57% and 53% respectively in each of these phases.

The overheads during initialization are due to secure context and module creation (22% of the overhead), secure copy of the model and data used for the initial test (31%), and command authentication during an initial testing phase (47%).

A breakdown of testing and training overheads shows that that command group authentication accounts for 66% and 77% of the overhead, respectively. This is because this workload executes a large number of relatively short kernels (one for each batch and layer). We profiled the time spent on kernel launches, and find that a large fraction of the overhead is due to the cost of emulating authenticated decryption of commands. In par-

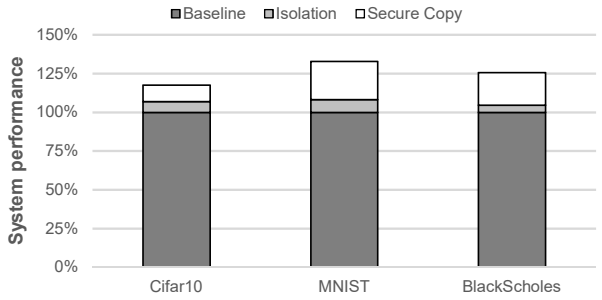


Figure 11: System performance for various benchmarks.

ticular, each secure kernel launch incurs a 9.2μs latency, with 0.8μs on encryption in the runtime, and 3.0μs on decryption in the emulator.

The figure also shows the estimated overhead assuming we extend the command processor with a hardware encryption engine. The overhead reduces from 35-41% to 5-7% for testing and training phases due to a reduction in time spent on authenticated decryption from 3μs to around 30ns. Adding a hardware encryption engine reduces the overall overhead to 17% (Figure 11).

MNIST. We use Caffe to train an autoencoder on the MNIST dataset, which consists of 60000 28x28 handwritten digits. The network consists of six encoding layers and six decoding layers. We run 10000 batches (with a batch size of 128) and test the model every 500 batches using 8192 images with a batch size of 256. Both baseline and Graviton achieve same accuracy.

As shown in Figure 11, Graviton introduces 33% performance overhead. The overhead is higher than in Cifar-10 as the complexity of encoding and decoding layers is lower than convolutional layers, and hence each iteration spends higher fraction time on secure memory copy.

BlackScholes. We run BlackScholes with 10 batches of four million options and 2500 iterations each. As shown in Figure 11, the overall overhead is 26%. Unlike Cifar-10, command authentication is not a factor in BlackScholes as it executes one long-running kernel per batch; thus, the overhead for enforcing isolation is attributed mainly to secure context and module creation.

8 Related Work

Trusted hardware. There is a history of work [9, 13, 15, 17, 24, 27, 32, 41, 48] on trusted hardware that isolates code and data from the rest of the system. Intel SGX [28] is the latest in this line of work, but stands out because it provides comprehensive protection and is already available in client CPUs and public cloud platforms. Graviton effectively extends the trust boundary of TEEs on the CPU to rich devices, such as GPUs.

Trusted execution on GPUs. A number of researchers have identified the need for mechanisms that allow an application hosted in a TEE to securely communicate with I/O devices over a *trusted path*. Yu et al. [51] propose an approach for using the trusted path approach for GPUs. Their approach relies on a privileged host component to enforce isolation between virtual machines and display, whereas our attacker model precludes trust in any host component.

PixelVault proposed an architecture for securely offloading cryptographic operations to a GPU [46]. Subsequent work has demonstrated that such design suffers from security vulnerabilities due to lack of page initialization upon allocations and module creation, lack of kernel-level isolation, and information leakage of registers by either attaching a debugger to a running kernel (and the GPU runtime) or invoking a kernel on the same channel [23, 52]. In contrast, Graviton enables a general-purpose trusted execution environment on GPUs. Information leakage via kernel debugging is prevented as the user hosts its GPU runtime inside a CPU TEE, guaranteeing that debugging cannot be enabled during execution.

9 Conclusion

Unlike recent CPUs, GPUs provide no support for trusted execution environments (TEE), creating a tradeoff between security and performance. In this paper, we introduce Graviton, an architecture for supporting TEEs on GPUs. Our proof-of-concept on NVIDIA GPUs shows that hardware complexity and performance overheads of the proposed architecture are low.

An interesting avenue for future work is to extend Graviton to secure kernel execution and communication across multiple GPUs and to investigate support for advanced features, such as on-demand paging and dynamic thread creation. In addition, we would like to investigate whether it is possible to remove the dependency on CPU TEEs, such as Intel SGX, and if so to quantify the implications on system performance. Finally, we would like to validate whether the proposed architecture extends to other accelerators, such as FPGAs.

Acknowledgements

We would like to thank our shepherd, Andrew Warfield, and the anonymous reviewers for their insightful comments and feedback on our work. Istvan Haller provided input during the early stages of this work. We thank Dushyanth Narayanan and Jay Lorch for comments on earlier drafts of the work.

References

[1] Security on ARM Trustzone. https://www.arm.com/products/security-on-arm/trustzone.

[2] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *Interantional Symposium on High Performance Computer Architecture*, 2012.

[3] S. Alhelaly, J. Dworak, T. Manikas, P. Gui, K. Nepal, and A. L. Crouch. Detecting a trojan die in 3D stacked integrated circuits. In *2017 IEEE North Atlantic Test Workshop*, 2017.

[4] AMD. AMD Radeon RX 300 series. https://en.wikipedia.org/wiki/AMD_Radeon_Rx_300_series.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyer, R. Kapitza, P. R. Pietzuch, and C. Fetzer. SCONE: Secure Linux containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[6] G. Bertoni, J. Daemen1, M. Peeters, G. V. Assche1, and R. V. Keer. Keccak and the SHA-3 standardization. https://keccak.team/index.html.

[7] G. Bertoni, J. Daemen1, M. Peeters, G. V. Assche1, R. V. Keer, and B. Viguier. KangarooTwelve: Fast hashing based on Keccak-p . In *Cryptology ePrint Archive: Report 2016/770*.

[8] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81:637–654, 1973.

[9] R. Boivie. SecureBlue++: CPU support for secure execution. In *IBM Research Report RC25287*, 2012.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX cache attacks are practical. *CoRR*, abs/1702.07521, 2017.

[11] I. Buck. NVIDIAs next-gen Pascal GPU architecture to provide 10X speedup for deep learning apps. https://blogs.nvidia.com/blog/2015/03/17/pascal.

[12] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang. Detecting privileged side-channel attacks in shielded execution with DéJà Vu. In *ACM Asia Conference on Computer and Communications Security*, 2017.

[13] V. Costan, I. A. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[14] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[15] D. Evtyushkin, J. Elwell, M. Ozsoy, D. V. Ponomarev, N. B. Abu-Ghazaleh, and R. Riley. Iso-X: A flexible architecture for hardware-managed isolated execution. In *International Symposium on Microarchitecture*, 2014.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31:6–15, 2011.

[17] O. S. Hofmann, S. M. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[18] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *International Conference on Multimedia*, 2014.

[19] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *CoRR*, abs/1804.06826, 2018.

[20] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, 2017.

[21] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-class GPU resource management in the operating system. In *USENIX Conference on Annual Technical Conference*, 2012.

[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[23] S. Lee, Y. Kim, J. Kim, and J. Kim. Stealing webpages rendered on your browser by exploiting GPU vulnerabilities. In *IEEE Symposium on Security and Privacy*, 2014.

[24] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[25] B. Madden. NVIDIA, AMD, and Intel: How they do their GPU virtualization. http://www.brianmadden.com/opinion/NVIDIA-AMD-and-Intel-How-they-do-their-GPU-virtualization.

[26] C. Maurice, C. Neumann, and A. Heen, Olivier andFrancillon. Confidentiality issues on a GPU in a virtualized environment. In *Financial Cryptography*, 2014.

[27] J. M. McCune, N. Qu, Y. Li, A. Datta, V. D. Gligor, and A. Perrig. Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*, 2009.

[28] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[29] Nouveau. Accelerated open source driver for NVIDIA cards. `https://nouveau.freedesktop.org/wiki`.

[30] NVIDIA. CUDA multi process service overview. `https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf`.

[31] NVIDIA. Cuda streams. `http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf`.

[32] E. Owusu, J. Guajardo, J. M. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer and Communications Security*, 2013.

[33] J. J. K. Park, Y. Park, and S. Mahkle. Chimera: Collaborative preemption for multitasking on a shared GPU. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[34] J. J. K. Park, Y. Park, and S. Mahkle. Dynamic resource management for efcient utilization of multitasking GPUs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[35] B. Pichai, L. Hsu, and A. Bhattacharjee. Architectural support for address translation on GPUs: Designing memory management units for CPU/GPUs with unified address spaces. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[36] R. D. Pietro, F. Lombardi, and A. Villani. CUDA Leaks: A detailed hack for CUDA and a (partial) fix. *ACM Transactions on Embedded Computing Systems*, 15(1), 2016.

[37] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecuture*, 2014.

[38] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[39] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. Panoply: Low-TCB Linux applications with SGX enclaves. In *Network and Distributed System Security Symposium*, 2017.

[40] Synopsys. DesignWare pipelined AES-GCM/CTR core. `https://www.synopsys.com/dw/ipdir.php?ds=security-aes-gcm-ctr`.

[41] R. Ta-Min, L. Litty, and D. Lie. Splitting Interfaces: Making trust between applications and operating systems configurable. In *USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[42] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *International Symposium on Computer Architecture*, 2014.

[43] S. Tomov, J. Dongarra, I. Yamazaki, A. Haidar, M. Gates, S. Donfack, P. Luszczek, A. Yarkhan, J. Kurzak, H. Anzt, and T. Dong. MAGMA: Development of high-performance linear algebra for GPUs. In *GPU Technology Conference*, 2014.

[44] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-SGX: A practical library os for unmodified applications on SGX. In *USENIX Annual Technical Conference*, 2017.

[45] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[46] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. PixelVault: Using GPUs for securing cryptographic operations. In *ACM Conference on Computer and Communications Security*, 2014.

[47] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *International Conference on High-Performance Computer Architecture*, 2016.

[48] S. Weiser and M. Werner. SGXIO: Generic trusted I/O path for Intel SGX. In *ACM Conference on Data and Application Security and Privacy*, 2017.

[49] E. Worthman. Designing for security. `https://www.semiengineering.com/designing-for-security-2`.

[50] Y. Xu, W. Cui, and M. Peinado. Controlled-Channel Attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*, 2015.

[51] M. Yu, V. D. Gligor, and Z. Zhou. Trusted display on untrusted commodity platforms. In *ACM Conference on Computer and Communications Security*, 2015.

[52] Z. Zhu, S. Kim, Y. Rozhanski, Y. Hu, E. Witchel, and M. Silberstein. Understanding the security of discrete GPUs. In *Workshop on General Purpose GPUs*, 2017.