Telekine: Secure Computing with Cloud GPUs

Tyler Hunt
The University of Texas at Austin

Zhipeng Jia
The University of Texas at Austin

Vance Miller
The University of Texas at Austin

Ariel Szekely
The University of Texas at Austin

Yige Hu
The University of Texas at Austin

Christopher J. Rossbach
The University of Texas at Austin
and VMware Research

Emmett Witchel
The University of Texas at Austin

Abstract

GPUs have become ubiquitous in the cloud due to the dramatic performance gains they enable in domains such as machine learning and computer vision. However, offloading GPU computation to the cloud requires placing enormous trust in providers and administrators. Recent proposals for GPU trusted execution environments (TEEs) are promising but fail to address very real side-channel concerns. To illustrate the severity of the problem, we demonstrate a novel attack that enables an attacker to correctly classify images from ImageNet [17] by observing only the timing of GPU kernel execution, rather than the images themselves.

Telekine enables applications to use GPU acceleration in the cloud securely, based on a novel GPU stream abstraction that ensures execution and interaction through untrusted components are independent of any secret data. Given a GPU with support for a TEE, Telekine employs a novel variant of API remoting to partition application-level software into components to ensure secret-dependent behaviors occur only on trusted components. Telekine can securely train modern image recognition models on MXNet [10] with 10%–22% performance penalty relative to an insecure baseline with a locally attached GPU. It runs graph algorithms using Galois [75] on one and two GPUs with 18%–41% overhead.

1 Introduction

GPUs have become popular computational accelerators in public clouds. Accuracy improvements enabled by GPU-accelerated computation are driving the success of machine learning and computer vision in application domains such as medicine [38, 88] transportation [67], finance [32], insurance [69], gaming [89], and communication [70].

Unfortunately, it is currently impossible to run GPU workloads in the cloud without trusting the provider, eliminating cloud GPUs as an option for security-conscious users. Users must trust the provider because the provider controls the layer of privileged software responsible for management and provisioning. Even dedicated cloud instances (e.g., Amazon's EC2 dedicated hosts [1]) run the provider's virtualization software, making GPUs vulnerable to malicious or curious

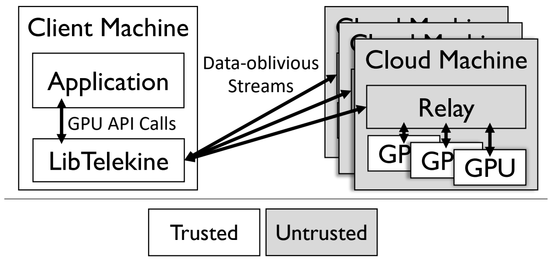


Figure 1. Telekine components and their organization.

cloud administrators. Virtualization software runs at a host machine's highest privilege level, exposing a wide attack surface that includes GPU memory, execution context, and firmware. Finally, unfettered visibility into host/device communication exposes both data and timing channels.

Trusted execution environments (TEEs) should, in principle, make the cloud an option for users who refuse to trust the provider. TEEs provide a hardware root of trust, allowing users to access cloud compute resources without trusting provider software–including the privileged software of the hypervisor and operating system. TEE hardware protects the privacy and integrity of user code and data from administrators and from attackers who control privileged software. TEEs exist currently on Intel CPUs via software guard extensions (SGX) [44], ARM CPUs via TrustZone [59], and RISC-V CPUs via Keystone [55]. Researchers have proposed GPU-based TEEs [100] and TEE extensions for GPUs [45], though none have been built or deployed. However, as we argue below, a design that simply composes components that run in hardware-supported CPU and GPU TEEs will fail to provide strong security due to side channels.

GPU-accelerated applications have three main software components: (1) an API and a user library (e.g., CUDA [66] or HIP [39]) that provides high-level programming functionality and executes on a CPU; (2) CPU-side control code at the user and the system level that manages communication with the GPU, and (3) GPU kernels (programs) that execute on the GPU device itself. It is the data and code that moves between the CPU and GPU that potentially creates side channels visible to CPU-side code.

An attacker can extract meaningful information from the execution time of code on the GPU, which privileged software can compute on the CPU by observing communication with the GPU. For example, we demonstrate a novel attack on image recognition machine learning models that allows malicious system software to correctly classify images from ImageNet [17] used as input to the model. By observing only the timing of a model trained to classify images (the image model), we build a new model (the timing model) that can classify images based on the execution timing of layers in the image model. Even if a security-conscious user encrypts their input data (and decrypts it on the GPU), a cloud provider's system administrator can use the timing information of GPU kernels (measured on the CPU) in the image model to classify its input images. We train the timing model to distinguish images of two classes with 78% accuracy. For more classes, accuracy decreases but stays above random guessing.

We propose Telekine, a system that enables the secure use of cloud GPUs without trusting the platform provider. GPU TEEs provide a secure execution environment but leave the user open to side channels when communication depends on secret data. Telekine makes communication with the GPU TEE data oblivious, that is, completely independent of secrets contained in the input data. Data obliviousness is a strong property that excludes the existence of side-channel attacks against CPU-side code and host/device communication whose observable behavior (e.g., timing, memory accesses, DMA sizes, etc.) depends on secret input data.

Telekine has three components (shown in Figure 1): libTelekine that runs on a trusted user machine (a *client*), GPUs physically attached to a cloud machine (a *server*) that supports GPU TEEs with specific security requirements (§3.1), and the relay which facilitates communication between libTelekine and the GPU. Telekine uses a GPU TEE because it needs a mechanism to protect GPU computation from the cloud provider; a GPU TEE is tailored to that task.

Telekine protects the application and GPU runtime by moving it from the cloud to the client. The advantage of this approach is that the user must already trust their client machine, and the application and user libraries are large and complex and therefore prone to side-channel attacks, making them difficult to secure if they execute in the cloud. The disadvantage is that GPU libraries assume a local GPU with a fast, high-bandwidth connection to the CPU. Telekine decouples the user library from low-level GPU control by interposing on the GPU API and efficiently forwarding these calls to the server (a technique known as API remoting) which has been used to virtualize GPUs [6, 8, 22, 23, 30, 33, 50, 57, 58, 85, 101, 107], but to our knowledge has never been used for security. A client using Telekine does not need to have a GPU installed.

Telekine treats the CPU-side control code on the cloud server ("Relay" in Figure 1) as completely untrusted, almost as if it were part of the network. The client machine establishes a cryptographically secure channel directly with the code executing on the cloud GPU. The network and the CPU-based code on the server can delay the computation, but cannot compromise its privacy or integrity.

Telekine secures the communication between the client machine and the cloud GPU by transforming the user's GPU API calls into *data-oblivious streams*. Data-oblivious streams are similar to constant time defenses [3] in that they aim to remove timing channels by ensuring that observable events are deterministic regardless of secrets. Telekine constructs data-oblivious streams by reducing all API calls to a sequence of code execution (`launchKernel`) and data movement (`memcpy`) commands. It then schedules these commands at a fixed rate, possibly creating new commands, or splitting `memcpy` commands into fixed-size pieces. Fixed-sized, fixed-rate communication is data oblivious; it ensures that any observable patterns are independent of the input data and therefore devoid of side-channel information. Fixed-rate communication is not a novel way to eliminate side channels, but Telekine's design shows how to apply it efficiently to modern GPU-based computing.

Given that Telekine requires a GPU TEE, it is logical to wonder why it does not use a CPU TEE. After all, putting the application and programming libraries into a CPU TEE would reduce the latency and increases the bandwidth for communication between libTelekine and the GPU. Unfortunately, Intel and ARM TEEs do not prevent side channels as part of their threat model [48, 78]. Keystone [55] and Komodo [25] intend to address side channels for RISC-V and ARM respectively, but work is ongoing. Also, making existing applications data oblivious is difficult for programmers, requires access to source code (not needed by Telekine), and often slows down a program greatly (e.g., Opaque [109] slows down data analytics by 1.6–46×). Should future CPU TEEs evolve to address side channels, Telekine can use them. Much of Telekine focuses on securing the communication between trusted components, which can be an improved CPU TEE and a GPU TEE or they can be the client machine and server GPU TEE, as they are in our prototype.

Telekine is the first system to offer efficient, secure execution of GPU-accelerated applications on cloud machines under a strong and realistic threat model. We use Telekine to secure several GPU-accelerated applications via two frameworks: the MXNet [10] machine learning framework and the Galois graph processing system [75]. On a realistic testbed Telekine provides strong secrecy and integrity guarantees, including side-channel protection. MXNet [10] training for three different, modern image recognition models incur a 10–22% performance penalty relative to a baseline with a locally attached GPU. MXNet inference for the same models over a connection from Austin, TX to the Vultur's Dallas, TX datacenter [102] incurs a penalties of 0-8% for batch sizes of 64 images. Telekine runs graph algorithms using Galois [75] on one and two GPUs with 18%–41% overhead.

This paper makes the following contributions.

- We demonstrate a CPU-side timing attack on deep neural networks that allows a compromised OS to correctly classify images in encrypted input (§4).
- We provide a design and prototype for Telekine, a system that eliminates CPU-based side-channel attacks against a GPU TEE with a novel variant of API remoting to execute secret-dependent code on the GPU TEE and a trusted client (§5).
- We thoroughly evaluate the performance, robustness, and security of Telekine protecting a variety of important workloads on one and two GPUs: machine learning and graph processing (§7).

2 Threat model

In all current cloud GPU platforms, the cloud provider's privileged software, and hence administrators, can gain easy access to GPU state, creating a significant attack surface including explicit channels such as GPU memory, firmware, and execution context. Work in this area agrees on the vulnerability of GPU state to privileged software [45, 100].

Telekine assumes a powerful adversary who controls all software on the platform, including privileged software such as device drivers, the host operating system and hypervisor. This captures typical cloud platforms, where the platform provider has full control over all software, and attackers can run malicious code on the same physical device as a target cloud application [81]. A malicious provider, a malicious administrator, or an OS-level attacker can use their control of privileged software to steal the secrets of tenants. We assume that the adversary cannot, however, compromise hardware–the physical GPU package.

Telekine assumes a GPU TEE, with capabilities similar to current research proposals like Graviton [100]. The details can vary, but a GPU TEE establishes secure memory on the GPU device and provides a protocol to initiate a computation that can be remotely attested to start from the correct state (code and initial data) and execute privately and without interference from the CPU side. We provide additional detail on Telekine's TEE requirements in Section 3.1.

GPU TEEs do not, by themselves, secure communication with the CPU and our attack (§4) shows how much information there is in the precise timing of CPU/GPU communication. Telekine protects communication with the GPU, guaranteeing that the adversary cannot learn about input data directly or through side channels, including timing channels.

While secure control of a GPU has been proposed [45, 100], there has been little work securing side channels. These side channels undercut the security of the TEE. In addition to the timing attack we developed (§4), AES key extraction using shared GPU hardware [31, 46, 47] has been demonstrated. And recent side-channel attacks [64] have shown practical methods to fingerprint websites using performance counters observed during GPU rendering in the browser.

2.1 Guarantees

Telekine provides the following secrecy properties which prevent any explicit or implicit data flow from input data to an external observer.

S1 (content): Messages are encrypted to ensure their content cannot be directly read by an observer.

S2 (timing): The transmit schedule for messages is fixed. Any transmission delays are independent of input data.

S3 (size): The size of each message is fixed. Telekine pads and/or splits messages to achieve fixed-sized messages.

Telekine also provides the following integrity properties to ensure that any result the user receives is either a result that could have been generated by a GPU hosted by a completely benign cloud provider, or an error.

I1 (content): The content of all communication is protected by an end-to-end integrity check; a message authentication code (MAC) allows Telekine to detect modifications, returning an error if any are detected.

I2 (order): Each message carries a sequence number which allows Telekine to detect out of order messages. The sequence numbers also prevent replay attacks.

I3 (API-preserving): Commands issued by the application should affect GPU state in the same way they would on a local GPU, regardless of any transformations that Telekine applies.

GPU commands have semantics that Telekine must maintain for correctness. For example, GPU runtimes expose a *stream* [71] abstraction to application code. API calls issued by the application on the same stream are executed serially in the order they were issued. A kernel launched from a particular stream will block the completion of subsequent API calls on that stream until that kernel terminates. Applications can have many streams which map to different command queues exposed by hardware. API calls made on separate streams can be executed in parallel. Telekine must respect the data dependence semantics of streams.

2.2 Limitations.

Physical side channels and denial of service attacks are out of scope. In situations where an adversary monitoring physical side channels like temperature [62], power [54], or acoustical emanations [11] is a concern, Telekine would need to be augmented with other techniques to maintain security. In our threat model, a cloud provider wishing to deny service can always do so, e.g., by interrupting the network or refusing to run user processes.

Telekine provides clients a mechanism to disguise their end-to-end runtime but does not impose policy. Applications can choose the most efficient policy for their security needs. We believe end-to-end runtime is a poor predictor of input data (and our experiments in Section 4 bear this out), further justifying the clients setting policy.

3 GPU background

Applications use GPUs through high-level, vendor-provided APIs such as CUDA [66] and HIP [39]; they include a user-level runtime and OS-level driver that communicate through a combination of `ioctl` system calls and memory-mapped command queues. The driver is responsible for creating mappings from virtual memory to physical MMIO regions. After these privileged operations are complete, any software that has a mapping (user or OS) may communicate directly with the device using registers or command queues exposed through the MMIO regions.

While memory management, synchronization, and other features (e.g., IPC, power management, etc.) require interaction with the driver state (e.g., creating and managing memory mappings), a workload that pre-allocates all of its required GPU memory and uses only data transfer and kernel launch primitives can function completely by writing commands into the GPU's command queue. It is possible to construct and submit these commands without referring to any state maintained by either the runtime or the driver. As we show in Section 5, this property enables Telekine's relay to be effectively stateless.

3.1 GPU TEE

Telekine requires a TEE on the GPU and Graviton [100] is a detailed proposal from the literature that provides the basic functionality that any GPU TEE (or indeed any TEE) should provide: secrecy for GPU code and input data, integrity for the GPU computation, and remote attestation for the computation's initial state. Graviton achieves most of its functionality by changing the GPU firmware, so it does not require extensive changes to the GPU hardware itself (neither does Telekine). This is achievable because the modern GPU firmware runs on a fully programmable control processor [68]. We explain GPU TEE functionality by saying what the GPU does, but the implementation could be firmware, hardware, or both.

The integrity, secrecy, and ordering of the commands sent to the GPU are ensured by a secure channel. Before computation begins, the client machine and the GPU agree on a shared symmetric key via a key exchange protocol (e.g., Diffie-Hellman). The client uses this key to send commands using a protocol like transport layer security (TLS) which provides a secure channel ([100] §5.2).

The integrity of the computation is assured by the GPU, which checks the initial execution conditions and attests these conditions to the remote user, who can verify that the expected code has been loaded into the expected address range with the expected permissions, and that the hardware generating the attestation is genuine. There are many variations on remote attestation, but it is a common feature for modern enclaves like SGX [14] and Keystone [55]. Telekine expects the GPU to have been initialized with all of the GPU kernels the application intends to launch and any initial data when attestation has completed.

Telekine and Graviton split GPU memory into untrusted and trusted regions so the untrusted host OS/Hypervisor can DMA into untrusted GPU memory, enabling efficient data transfers; the GPU can then copy data between untrusted GPU memory and trusted memory. This mechanism provides GPU memory protection even though the IOMMU is under control of the untrusted kernel. Telekine and Graviton disable unified memory, which allows privileged CPU code to demand page GPU memory and exposes side-channel memory access information.

The GPU TEE should turn off or refuse to report the state of any performance counters. Recent GPU side-channel attacks [28, 64] have successfully used timing data from GPU performance counters.

Due to Telekine's focus on side channels, it has requirements beyond the previously proposed GPU TEEs. These requirements are more straightforward to provide than the core TEE functionality.

Eliminate GPU side channels. Some TEE designs allow different tenants/principals to execute concurrently (e.g., SGX, Keystone), sharing the underlying hardware. Concurrent execution is attractive from a utilization perspective but it provides a rich side-channel attack surface which has plagued the security of CPU TEE designs. Telekine assumes side channels from concurrent principals (e.g., memory access timing and bandwidth) do not exist on the GPU TEE. A conservative design which prevents hardware side channels is to disallow concurrent execution. Graviton TEEs scrub their state (e.g., registers, memory, caches) after resources are freed so there is no danger of tenants observing transient state from previous computation.

Conceal kernel completions. GPUs signal the CPU via an interrupt when a kernel has completed its execution. Interrupt timing leaks information about the kernel's runtime. Rather than rely on interrupts, Telekine uses data-oblivious streams (§5.1) that include tagged buffers that allow the GPU to communicate computational results back to the client. The platform only sees DMA from the GPU to untrusted CPU memory at a fixed rate.

Support no-op kernel launches. Dependences between GPU kernels often cause the launch of one kernel to wait for another's completion, which provides indirect timing information. The GPU TEE must support a no-op kernel launch command so that Telekine can generate cover traffic ensure the adversary sees kernel launches at a fixed rate.

Timely command consumption. The GPU TEE should consume its command queue independently of how long kernels take to execute on the GPU. If the GPU waits until each kernel completes before dequeueing the next launch command, it can fall behind the input queue fill rate, allowing the input queue to fill. The adversary can detect this situation by observing how often the encrypted queue content changes, creating a proxy for kernel execution time. The GPU should consume command queue entries at a fixed rate, discard the no-ops, and store the real commands internally until they can be executed. Telekine can hold back real kernel launches and send no-op launches in their places to ensure these internal GPU queues do not fill up.

3.2 Communicating with GPUs

GPUs can be connected to the CPU memory interconnect (integrated) or to the PCIe bus (discrete). We focus on PCIe-attached GPUs because they are preferred in performance-focused settings like the cloud due to their higher memory bandwidth and better performance.

The PCIe interfaces provide two forms of communication: memory-mapped I/O (MMIO) and direct memory access (DMA). MMIO re-purposes regions of the physical memory address space for device communication. Contiguous physical ranges, or *BARs* (base-address-regions) are reserved by the hardware, and the hardware redirects loads/stores targeting those regions to the device. Modern GPUs use MMIO BARs to expose registers for configuring the device, and frequently accessed device memory (e.g., command queues).

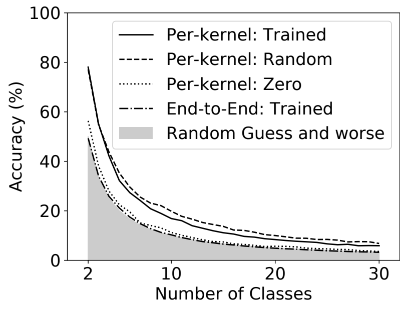
Any software that can obtain a mapping to MMIO can potentially communicate with the GPU to control it (through a register or command queue interface) or read/write its memory (through an MMIO memory BAR or by configuring DMA transfers to/from it). Telekine assumes GPU TEE support similar to Graviton [100] to prevent MMIO access to GPU status and configuration registers during secure execution.

The hypervisor and/or host operating system controls the PCIe bus, which routes packets to multiple devices connected to the PCIe root complex in a tree topology. Packets in transit to/from the GPU may be visible to other devices. Privileged host software may change the routing topology dynamically and can install pseudo-devices that allow it to sniff traffic. Securing communication with the GPU must defend against these passive and active PCIe attacks.

4 Example side-channel attack

Telekine addresses software attacks launched by an adverary resident on a cloud host, such as those launched by a malicious system administrator or a network-based attacker who has compromised the platform's privileged software. These attacks use privileged software to compromise the privacy or integrity of user code and data. Telekine is particularly focused on protecting against timing channels because effective, general-purpose attacks using timing channels have recently been demonstrated at the architecture level [53, 60, 84, 96], the OS level [97, 105], and the GPU programming level [46, 47]. Modern CPU TEEs exclude side channels from their threat model [31, 48, 78], leaving current hardware-supported security primitives vulnerable to side-channel attack. Telekine offers a unique and efficient security solution for cloud resident, GPU-based computation.

We demonstrate a proof-of-concept attack on machine learning inference in which the adversary uses the execution timing of individual GPU kernels to learn information about encrypted input data. Our attack allows privileged software on the cloud host to correctly classify images using only the timing of GPU kernel execution obtained on the CPU. The attacker can train their timing model on their own input, they do not need the victim's training data. The image data remains encrypted while on the CPU and the attack does not require any access to GPU architectural or microarchitectural state (including GPU timers).



(a) Batches of size 1

Purity	Accuracy
0.25	29.3%
0.40	33.0%
0.60	41.0%
0.80	50.0%
0.90	56.1%
1.00	65.4%

(b) Batches of size 32

Figure 2. Accuracy of multiclass classification for side-channel attacks. (a) shows the accuracy for a baches of size 1 with an increasing number of classes. (b) shows the accuracy for batches of size 32, 4 classes, and varies how much of the batch contains the target image (purity)

Attack basics. Convolutional neural networks (CNNs) are a popular neural network architecture for analyzing images [37, 40, 91]. Each network consists of multiple layers, including convolutions, which are good at detecting features of the input image that the remainder of the network can use to classify the image. When CNNs are executed on a GPU, the computation for each layer roughly corresponds to the execution of a single GPU kernel. While the actual mapping between layers and kernels is often more complex, the intuition behind our attack is that the timing of the execution of certain CNN layers (and hence their GPU kernels) indicates the presence or absence of certain features within the input image. This makes the per-layer execution time itself a rich feature.

Telekine defeats the attack by removing the adversary's ability to infer the timing of individual kernels. The adversary retains only the ability to measure the end-to-end runtime of the inference task. However, our data shows that end-to-end runtime provides very little predictive value, making the attack not much more accurate than randomly guessing (Figure 2a). Telekine gives users the mechanism to disguise their end-to-end execution time, should they decide to do so (§2.2).

Attack details. We demonstrate this attack on ResNet50 [37], a CNN widely used for image recognition, using the timing of GPU kernel completion events as detected by the operating system on the CPU (though we monitor a function in the GPU's user-level runtime for ease of implementation). We evaluate the accuracy of our attack using 5-fold cross validation.

We start with a pre-trained model for the standard ImageNet [17] dataset which contains 1,000 different image classes. Figure 2a shows the accuracy of distinguishing image classes based on the timing of the pre-trained model's layers (Per-kernel: Trained), versus the same attack using only end-to-end timing information (End-to-end: Trained). The accuracy of the per-kernel classifier is startlingly good for small numbers of classes: 78% for two classes, 55% for three and 42% for four. As the number of classes of input images increases, the accuracy of our classification declines, but it remains much better than random guessing, outperforming guessing by over 1.9× even among 30 input image classes.

We believe the root cause of the attack is timing dependent GPU operations, probably multiplication by zero. We compare a pre-trained model (Per-kernel: Trained with no zero-valued weights), a randomly initialized model (Per-kernel: Random with 0.2% zero-valued weights), and a model whose weights are all zero (Per-kernel: Zero with 100% zero-valued weights). The zero model has bad accuracy that is close to random guessing. A randomly initialized model is best, followed by the pre-trained model.

These results were generated using MXNet [10] ported to HIP on the ROCm version 1.8 stack for AMD GPUs which is used in the prototype; we saw similar results on the 2.9 version. Preliminary tests showed that this specific attack is much less powerful on NVIDIA GPUs.

Batched classification. Because inference is often done in batches, we examine the accuracy of a batched attack. We construct batches by splitting each ImageNet class into disjoint training and test sets. Images are then randomly sampled from each of these sets to form the batches.

We present the accuracy of our attack when distinguishing four ImageNet classes in batches of size 32 (Figure 2b.) Each batch is made up of the given fraction of images from a primary class (Purity), and randomly selected images from the remaining three classes. Our objective is to correctly identify the primary class.

Batches help, with the accuracy of our attack improving with larger batch sizes. Larger batches execute more operations, effectively amplifying the timing signal our attack relies on. Moreover, larger batches smooth out execution timings for outlier images which would otherwise be less recognizable to our attack model. When distinguishing four classes (Figure 2b), the batched attack is better than random guessing even when only 25% of the input images come from the target class. The accuracy increases with higher batch purity, outperforming single images by up to 64%.

5 Design

Telekine secures GPU-based computation from active attackers, including side-channel threats. Side channels include the execution timing of individual GPU kernels as well as data movement to and from the GPU. Telekine achieves its security by transforming an application's computation so that all communication—including data movement—among trusted components is data oblivious. Telekine only trusts the client machine and the in-cloud GPU TEE and must, therefore,

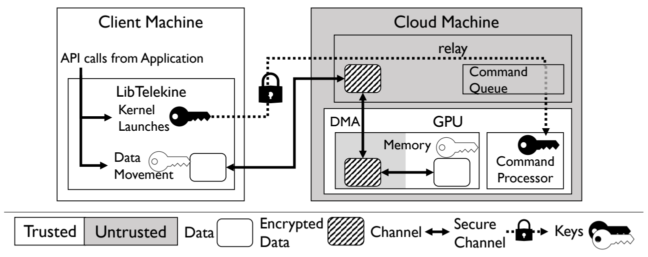


Figure 3. Detailed Telekine overview.

efficiently coordinate the computation between these entities, even though communication occurs over a wide area network, rather than over higher-bandwidth, lower-latency fabric like a data center network or a PCIe bus.

Telekine consists of three components (depicted in Figure 1 with detail in Figure 3).

- LibTelekine: a library that intercepts GPU API calls from the application and transparently transforms the calls into a data-oblivious command stream.
- Relay: an untrusted process that runs in the cloud and directs the client's command stream to the GPU.
- GPU: a GPU (or multiple GPUs) with TEE support that meets Telekine's requirements (see §3.1 for details).

LibTelekine is linked into the application running on the client. During its execution, the application issues a stream of GPU commands through the normal GPU API. Similar to normal API remoting [8, 21, 101], libTelekine redirects API calls made by the client to a server process with a GPU runtime–the relay on the cloud machine. Telekine treats the relay almost as if it were part of the network, relying on it to communicate with the GPU but protecting that communication with end-to-end techniques. The relay is not part of Telekine's trusted computing base.

All communication between libTelekine and the GPU is protected with authenticated encryption (AES-GCM [24] in our prototype) and sequence numbers. This creates a secure channel satisfying the secrecy property *S1 (content)* and the integrity properties *I1 (content)* and *I2 (order)* (described in §2.1), ensuring that the GPU commands issued by libTelekine can only be read by the GPU, and any tampering or reordering is detectable. However, by observing when messages are exchanged with the GPU (regardless of whether they are encrypted), the adversary can get timing information about the computation on the GPU.

Telekine's goal is to remove all timing information from the encrypted stream of GPU commands. It removes timing information by sending commands (GPU runtime API calls like `launchKernel` and `memcpy`) at a fixed rate, independent of input data. Fixed rating is a simple idea, but Telekine must overcome two major challenges to fix-rate GPU communication.

1. Different GPU command types are distinguishable because they have different sizes and they result in different communication patterns with the GPU. (e.g.,

launchKernel commands interact with MMIO ring buffers and memcpy commands are handled using DMA). Telekine must ensure that the attacker's ability to distinguish between these commands conveys no information about the input data.

2. Conventional GPU command streams (§2.1) exhibit a variety of data-dependent behavior whose timing is externally visible (e.g., a kernel launch after a data transfer will wait for the data transfer to finish). Telekine must maintain the ordering semantics induced by such data dependencies.

Telekine introduces a new primitive to overcome these challenges: *data-oblivious streams*. Data-oblivious streams transparently replace conventional GPU streams (and applications may have more than one), maintaining their semantics while making their communication with the GPU data oblivious. First, they separate commands by type, and schedule each type independently. Second, they split, pad, and batch commands of each type so that the encrypted payload is always the same size for messages of that type, satisfying *S3 (size)*. Third, they inject management commands as needed to maintain data-dependencies across message types, satisfying *I3 (API-preserving)*. Finally, data-oblivious streams send the transformed commands according to a fixed schedule, satisfying *S2 (timing)*.

The relay, privileged software on the cloud machine, and the network stack can delay commands since they are under complete control of the (possibly adversarial) cloud provider. However, they cannot delay commands in a way that leaks input data because all observable behavior of the trusted computing base (including its timing) is independent of input data.

5.1 Data-oblivious stream construction

Constructing data-oblivious streams only requires reasoning about memcpy and launchKernel commands. The TEE takes care of initialization (§3.1). The only other runtime commands deal with stream synchronization, and Telekine transforms those commands into memcpy and launchKernel commands as well (discussed fully in §5.4). memcpy commands are visible to the untrusted host's privileged software because GPU drivers use DMA for efficient data transfers. In Telekine, the data itself is protected and copied to/from a fixed staging area in untrusted GPU memory so the destination/source of the memcpy does not leak information.

Conventional GPU streams can create timing channels from memcpy and launchKernel commands because a memcpy command waits for all previous launchKernel commands on the same stream. To eliminate this channel, Telekine uses two GPU streams to construct a single data-oblivious stream. Telekine uses one GPU stream to launch the application's kernels; this stream is called the ExecStream. Telekine uses the other stream—called the XferStream—to move data to and from the GPU. Telekine ensures that commands on the XferStream never leak information about the kernel execution time by waiting for commands on the ExecStream.

The ExecStream. Application kernels are all launched on the ExecStream. LibTelekine maintains a queue of the launchKernel commands requested by the application and releases the commands in order according to the fixed-rate schedule. The GPU consumes these commands independently of any ongoing kernel execution and buffers them internally since their execution must be serialized according to GPU stream semantics. Telekine honors data dependences between memcpy and launchKernel commands by inserting data management kernels that block the progress of the ExecStream by spinning until the data is in place.

The XferStream. Data transfers requested by the application are launched on the XferStream. Unlike launchKernel commands, memcpy commands are directional (i.e., client-to-GPU and GPU-to-client), and directions are detectable. For example, because the adversary can observe interaction with the network, it can differentiate between messages that came over the network in transit to the GPU, and messages copied from the GPU that are being sent over the network. LibTelekine maintains separate queues for each direction and schedules them independently to avoid leaking information. Data for client-to-GPU transfers starts on the client, flows through the relay and into untrusted memory on the GPU. LibTelekine then enqueues a kernel, which moves the data from the untrusted staging memory into trusted GPU memory. Similarly, in the GPU-to-client direction, Telekine first enqueues a launchKernel on the XferStream to move the data into untrusted GPU memory, then issues a memcpy to copy it to the relay where it can be transferred over the network back to the client.

Fixed-size commands. Telekine ensures that all memcpy commands are the same size by splitting and padding the memcpy commands issued by the application to a standard size. When there are no pending memcpy commands, Telekine maintains the same rate of data flow by scheduling dummy, standard-sized memcpys to/from a staging buffer. Similarly, all launchKernel commands are padded to the same size (320 bytes in our prototype). When no launchKernel command is available, Telekine schedules no-op launchKernel commands.

Schedules. Any schedule Telekine uses for GPU communication is secure so long as it does not depend on the data being protected. Our prototype uses simple schedules which send a fixed number of fixed-sized commands after each fixed-time interval. For instance, Telekine might launch 16 kernels on the ExecStream every 3 milliseconds, and send then receive 4MB of data every 6 milliseconds on the XferStream.

Schedules can leak the category. While scheduling work at a fixed rate is a well-known technique to avoid side-channel leakage, the exact schedule is relevant to performance. We

Algorithm 1 Telekine's replacement functions for `memcpy` and `launchKernel`. Splitting and padding steps are omitted for brevity.

```
 1: function LaunchKernel($kern$,$args$...)
 2:     Enqueue($kernelQueue$,$\{kern,args\}$)
 3: end function
 4:
 5: function MemcpyH2D($src$,$dst$)
 6:     $buf$ ← ChooseTaggedBuffer()
 7:     LaunchKernel(copy_in, $buf$, $dst$)
 8:     Enqueue($dataQueueH2D$, $\{src, buf\}$)
 9: end function
10:
11: function MemcpyD2H($src$, $dst$)
12:     $buf$ ← ChooseTaggedBuffer()
13:     LaunchKernel(copy_out, $src$, $buf$)
14:     Enqueue($dataQueueD2H$, $\{buf, dst\}$)
15: end function
```

report our schedules in Table 1, and they are the same for all tasks of a given category, e.g., training different machine learning models with MXNet. However, they can differ across categories, e.g., Galois has a different ExecStream schedule from MXNet (§7). Under our threat model, the adversary would be able to differentiate these workloads from their network traffic. A user can always choose a more generic, but lower performing schedule if this is a concern.

5.2 Telekine operation

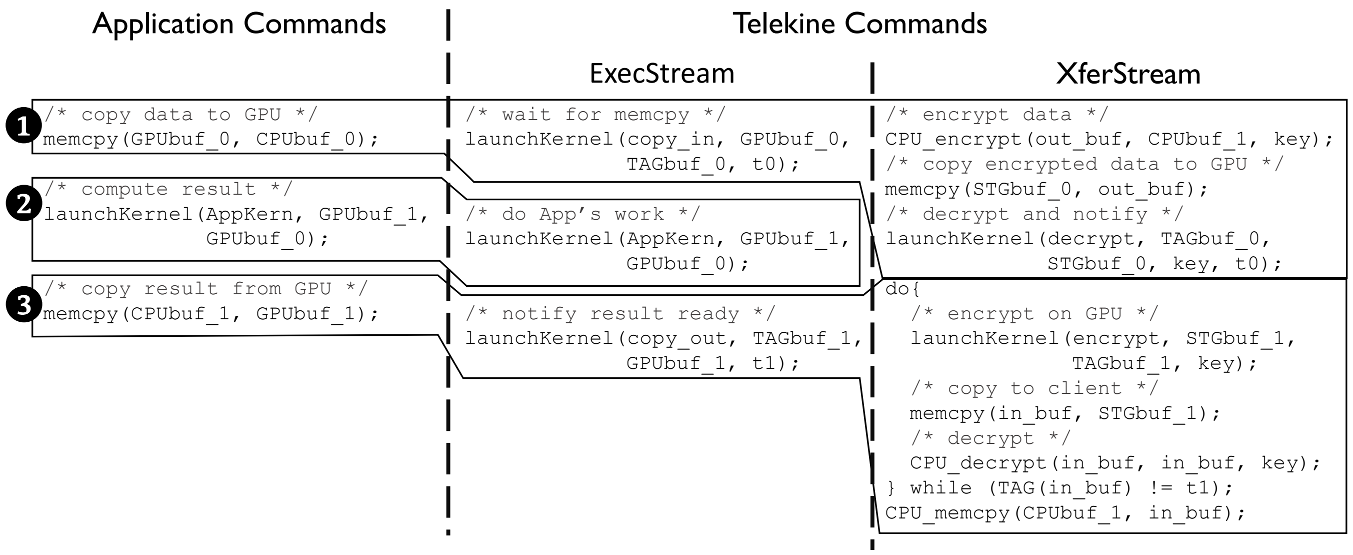
Algorithm 1 and Algorithm 2 provide a high-level description of Telekine's data-oblivious streams. In Algorithm 1, Telekine intercepts the application's calls to `launchKernel` and `memcpy` and transforms them into interactions with queues: kernelQueue, dataQueueH2D, and dataQueueD2H (splitting, padding, and encryption steps are omitted for brevity). The Telekine threads shown in Algorithm 2 dequeue the commands and release them to the GPU according to the schedule. Telekine waits at lines 7, 18, and 29 for the next available time slot ensuring that interactions with the queues do not influence the timing of messages.

Most `memcpy` commands have strict ordering requirements with respect to kernels that operate on their data. The `memcpy` then `launchKernel` idiom ensures that the launched kernel has fresh data to process. While Telekine decouples `memcpy` commands by scheduling them on their own stream for security, it needs to preserve the original ordering semantics expected by the application. Telekine maintains these semantics by injecting its own data management kernels into the ExecStream (shown on lines 7 and 13 of Algorithm 1) to enforce the ordering expected by the application. These data management kernels operate on *tagged buffers* which Telekine uses to synchronize data access.

Algorithm 2 Periodic tasks performed by Telekine according to the schedule. Encryption and decryption steps are omitted for brevity.

```
 1: loop                                          ▷ ExecStream Thread
 2:     if Empty($kernelQueue$) then
 3:         $op$ ← no_op
 4:     else
 5:         $op$ ← Dequeue($kernelQueue$)
 6:     end if
 7:     WaitForScheduledTime()
 8:     RemoteLaunchKernel($op$)
 9: end loop
10:
11: loop                    ▷ XferStream Client-to-GPU (H2D) Thread
12:     if Empty($DataQueueH2D$) then
13:         $src$ ← dummy_CPU
14:         $dst$ ← ChooseTaggedBuffer()
15:     else
16:         $\{src, dst\}$ ← Dequeue($dataQueueH2D$)
17:     end if
18:     WaitForScheduledTime()
19:     RemoteMemcpy($src$, $dst$)
20: end loop
21:
22: loop                    ▷ XferStream GPU-to-Client (D2H) Thread
23:     if Empty($DataQueueD2H$) then
24:         $src$ ← ChooseTaggedBuffer()
25:         $dst$ ← dummy_CPU
26:     else
27:         $\{src, dst\}$ ← Peek($dataQueueD2H$)
28:     end if
29:     WaitForScheduledTime()
30:     RemoteMemcpy($src$, $dst$)
31:     if $dst \neq$ dummy_CPU then
32:         if TagMatches($dst$) then
33:             Dequeue($dataQueueD2H$)
34:         end if
35:     end if
36: end loop
```

Tagged buffers. Tagged buffers are pre-allocated staging buffers on the GPU, each with an associated tag slot. Telekine assigns every `memcpy` operation a tagged buffer and a unique tag, represented by "ChooseTaggedBuffer" in Algorithm 1 and Algorithm 2. Data management kernels producing data (e.g., copying out the result of a kernel computation) write the tag into the tag slot of the chosen tagged buffer after the operation has completed and a memory barrier completes. Data management kernels that consume data (e.g., waiting for data a kernel expects to use as input) wait until the tag slot of the assigned buffer contains the expected value since they cannot be sure the buffer data is valid until the tag value matches its expectation.



```
/* copy data to GPU */
memcpy(GPUbuf_0, CPUbuf_0);

/* compute result */
launchKernel(AppKern, GPUbuf_1,
             GPUbuf_0);

/* copy result from GPU */
memcpy(CPUbuf_1, GPUbuf_1);
```

```
/* wait for memcpy */
launchKernel(copy_in, GPUbuf_0,
             TAGbuf_0, t0);

/* do App's work */
launchKernel(AppKern, GPUbuf_1,
             GPUbuf_0);

/* notify result ready */
launchKernel(copy_out, TAGbuf_1,
             GPUbuf_1, t1);
```

```
/* encrypt data */
CPU_encrypt(out_buf, CPUbuf_1, key);
/* copy encrypted data to GPU */
memcpy(STGbuf_0, out_buf);
/* decrypt and notify */
launchKernel(decrypt, TAGbuf_0,
             STGbuf_0, key, t0);

do{
  /* encrypt on GPU */
  launchKernel(encrypt, STGbuf_1,
               TAGbuf_1, key);
  /* copy to client */
  memcpy(in_buf, STGbuf_1);
  /* decrypt */
  CPU_decrypt(in_buf, in_buf, key);
} while (TAG(in_buf) != t1);
CPU_memcpy(CPUbuf_1, in_buf);
```

Figure 4. API calls made by the application and their mapping to underlying commands performed by Telekine.

Data management kernels. Telekine inserts its own data management kernels into the ExecStream which either produce or consume tagged buffers depending on the direction of the transfer. There are two kernels: `copy_in` and `copy_out`. Both kernels take an application-defined memory location, a tagged buffer, and a tag as arguments. For CPU-to-GPU `memcpy`s, libTelekine inserts a `copy_in` launch into the ExecStream. The `copy_in` will repeatedly check the tag slot of the buffer, completing the copy to the application's buffer only after verifying the tag slot matches the tag it was given as an argument. To service GPU-to-CPU `memcpy`s, Telekine inserts a `copy_out` into the ExecStream after the application kernel which generates the data. The `copy_out` writes the data to the assigned tagged buffer, followed by the tag to signal to Telekine that the data is ready. Since libTelekine runs on the client it has no way of knowing when the copy out has completed until the tagged buffer has been copied back, so it will retry the same GPU-to-CPU copy until the tag is correct corresponding to a complete copy. This is represented by the PEEK operation on line 27 of Algorithm 2, the operation is only dequeued after libTelekine verifies that the `copy_out` kernel did its work on line 32.

GPU-to-GPU data copies. Emerging hardware supports dedicated, high-bandwidth, cross-GPU communication links such as NVLink [26]. NVLink improves cross-GPU data copy efficiency but does not change the fundamental communication mechanisms used in a GPU stack. Telekine currently implements GPU-to-GPU copies as two copies: one from the first GPU back to the client and the second from the client to the second GPU. Direct GPU-to-GPU copies using NVLink would be far more efficient, but to be data oblivious they would have to occur at a fixed rate. We leave this task for future work.

Discussion. The XferStream is carefully constructed so that it never synchronizes with the ExecStream. The XferStream contains DMA operations which the OS can detect; if application kernels on the ExecStream occupy the GPU causing the encryption kernels on the XferStream—and transitively the DMAs—to wait, then the platform can learn some information about kernel execution times. There may still be leakage between the XferStream and the ExecStream because we cannot guarantee that kernels of the former will not interfere with the latter. However, we believe this leakage to be hard to exploit in practice, we have not seen it in any of our benchmarks, and we expect that future GPU features like strict priority [72] or preemption [92] will allow Telekine to seal the leak.

5.3 Data movement example.

Figure 4 shows an example of how Telekine transforms application commands into equivalent, data-oblivious commands on the ExecStream and XferStream. The application issues 3 commands: ❶ copy data to the GPU, ❷ launch a kernel to process that data, and ❸ copy the results of the computation out of the GPU back to the CPU.

❶: The application requests a `memcpy` from `CPUbuf_0` to `GPUbuf_0`. In response, Telekine chooses a tag, `t0`, and tagged buffer, `TAGbuf_0`, for this operation, then enqueues a kernel, `copy_in`, on the ExecStream. The `copy_in` kernel will spin on the GPU, using atomic operations to check the end of `TAGbuf_0` until it sees `t0`, then copy the contents of `TAGbuf_0` into `GPUbuf_0`. On the XferStream, Telekine encrypts the data, then copies the encrypted data to a staging buffer in untrusted GPU memory `STGbuf_0`. Finally, Telekine launches a kernel, `decrypt`, on the XferStream which reads the encrypted data out of untrusted memory and decrypts it into `TAGbuf_0`. After the data is written, the tag `t0` is appended

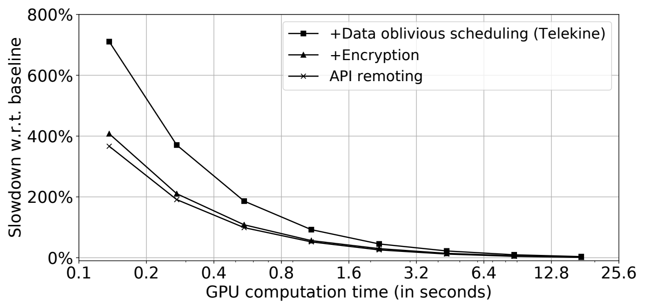


Figure 5. A microbenchmark which shows how Telekine overheads decrease as the running time of the GPU computation increases.

after a memory barrier, signaling to `copy_in` that the data is ready.

❷ : The application launches its kernel, `AppKern`, which processes the data in `GPUbuf_0` and writes its result into `GPUbuf_1`. Since `AppKern` is launched on the ExecStream after `copy_in` it will wait for `copy_in` to complete, ensuring that the data will be in `GPUbuf_0` before `AppKern` starts. The platform cannot detect that `AppKern` has started.

❸ : The application issues a request to copy the results of `AppKern` from `GPUbuf_1` to `CPU_buf1`. In response, Telekine again chooses a tag and tagged buffer, `t1` and `TAGbuf_1` respectively, and immediately enqueues a `copy_out` kernel on the ExecStream. After the application's kernel, `AppKern`, has completed, `copy_out` moves the result of its computation in `GPUbuf_1` into `TAGbuf_1` then atomically appends `t1`. While waiting for `copy_out` to finish, Telekine periodically encrypts `TAGbuf_1` into a staging buffer in untrusted memory, `STGbuf_1`, then issues a `memcpy` operation to copy the contents of `STGbuf_1` to a client-side buffer, `in_buf`. Telekine decrypts `in_buf` and checks the tag. If the tag matches `t1`, `copy_out` and `AppKern` must have completed and the data can be copied into `CPUbuf_1`. If not, this process will be repeated during the next scheduled GPU to client transfer.

5.4 Synchronizing data-oblivious streams

Applications sometimes wish to synchronize with their GPU streams (i.e., wait for all outstanding commands to complete), or synchronize one GPU stream with another (i.e., ensure another stream has completed some operation, n, before this stream starts operation, m). Telekine handles both of these cases by injecting kernels that increment a counter in GPU memory between kernels in the ExecStream. Because of stream semantics, the increment kernel only runs after all previous kernels in the stream, providing an accurate count of how many application kernels have executed. Telekine copies that counter back to the client periodically and can block the application thread until all submitted work has completed.

	ExecStream		XferStream		
Benchmark	Quantum	Size	Quantum	Size	Bandwidth
Microbench	15ms	32kerns	30ms	1MB	533 Mb/s
MXNet	15ms	512kerns	30ms	1MB	533 Mb/s
Galois1	15ms	32kerns	30ms	1MB	533 Mb/s
Galois2	15ms	32kerns	30ms	1MB	533 Mb/s

Table 1. Data-oblivious schedule parameters and the network bandwidth required. MicroBench from §7.1; MXNet from §7.2; Galois1 executes on one GPU, Galois2 on two from §7.3. ExecStream sizes are number of kernel launches, each of which is 320 bytes. XferStream streams contribute twice their size to bandwidth consumption because Telekine copies data in both directions at every quantum.

6 Implementation

The Telekine prototype is based on AMD's ROCm 1.8 [2], an open-source software stack for AMD GPUs. Telekine requires an open-source stack because we split its functionality between user and cloud machines. NVIDIA is generally thought to have higher hardware and software performance as well as better third-party software support. But NVIDIA only officially supports closed-source drivers and runtimes.

LibTelekine and the relay. All applications were ported to use HIP [39], the ROCm CUDA replacement. LibTelekine marshals the arguments of HIP API calls to be sent over a TLS protected TCP connection to the relay to support initialization. The libTelekine and relay prototype are based on code generated by AvA [107]; they total 8,843 and 5,650 lines of C/C++/HIP code respectively (measured by cloc [12]).

GPU TEE. GPU TEE requirements are made explicit in Section 3.1, and most of those requirements are safety properties that do not impact performance. A notable exception is the cryptography required to secure the secrecy and integrity of kernel launch commands. We model the timing of these features by decrypting kernel launch commands in the relay.

7 Evaluation

We quantify the overheads of the security Telekine provides by comparing it to an insecure baseline: applications run on cloud provider machines that offload computation to GPUs directly through the GPU runtime.

We measure Telekine across two testbeds. The first is the *simulated testbed* which simulates wide-area network (WAN) latency and bandwidth, providing a controlled environment for measurement. The second is the *geodist testbed* in which the server and client are geodistributed and connected by the Internet. Both testbeds use the same "cloud machine" (the *server*), which has an Intel i9-9900K CPU with 8 cores @3.60GHz, 32GB of RAM and two Radeon RX VEGA 64 GPUs each with 8GB of RAM. All machines are running Ubuntu 16.04.6 LTS with Linux kernel version 4.13.0, and AMD's ROCm-1.8 runtime and HIP-1.5 compiler.

In the simulated testbed, the client has an Intel Xeon E3-1270 v6 processor with 4 cores @3.8GHz and 32GB of RAM.

	ResNet	InceptionV3	DenseNet
Model size	97.5 MB	90.9 MB	30.4 MB
Input size			
Input image	224x224x3	299x299x3	224x224x3
Batch size	64	64	48
Data size per batch	9.2 MB	16.4 MB	6.9 MB
Single-GPU training baseline			
T-put	20.27 MB/s	11.05 MB/s	13.57 MB/s
T-put (less sync)	22.69 MB/s	11.66 MB/s	17.46 MB/s

Table 2. Overview of machine learning training on MXNet. Input size is given in pixel dimensions, batch size in images per GPU. T-put is throughput.

Both this client and the server are equipped with a Gtek X540 10Gb NIC, which we connect directly. We simulate a client-to-cloud network connection in a controlled environment using `netem` [65], which allows us to add network delays and limit bandwidth. We always limit the bandwidth of the connection to 1Gbps and unless otherwise mentioned we add delays in both directions so that the total round trip time (RTT) is 10ms. These parameters are conservative for a network connection to an edge cloud server [16, 106].

In the geodist testbed, the client is a VM hosted by vultr [102] in their Dallas, TX datacenter (the server is in Austin, TX). The VM has 8 vCPUs and 32GB of RAM. We measured the RTT between the server and this client at 12ms, and the average bandwidth at 877Mbps.

Different applications use different schedules to get good performance, though Table 1 shows strong similarity among the data-oblivious schedules we use for evaluation.

7.1 Telekine performance tradeoff

Figure 5 shows the performance tradeoff for a microbenchmark with 16MB of input and output and a GPU kernel with a configurable running time on the simulated testbed. The different lines show the costs of specific sources of overhead. The "API remoting" line uses the XferStream and the ExecStream over the network. The "+Encryption" line adds encryption to API remoting. Finally, the "Data-oblivious scheduling" line adds the data-oblivious schedule described in Table 1 to encryption. When the GPU kernel executes for only 0.14 seconds, the overhead of Telekine is nearly 8×. Once the computation takes 4.4s the overhead is only 22%. Telekine is a remote execution system; it makes communication more expensive because of its oblivious scheduling as well as network delay and limited bandwidth. It is most efficient when computation dominates communication, which is the case for our benchmarks.

7.2 Machine learning algorithms

We port MXNet [10], a state-of-the-art machine learning library, to run on the HIP runtime. Our port is based on MXNet v1.1.0 (git commit `07a83a03`). We also use AMD's MIOpen library for efficient neural network operators. Some parts of

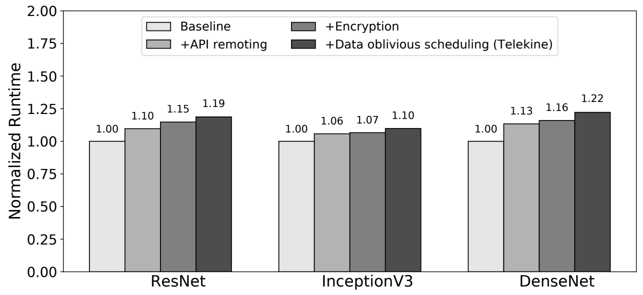


Figure 6. Performance of machine learning training algorithms using a single GPU with Telekine on the simulated testbed.

ResNet	InceptionV3	DenseNet
1.23×	1.08×	1.20×

Table 3. Performance of machine learning training algorithms on Telekine, measured on the geodist testbed.

MXNet adaptively choose from different GPU kernel implementations by measuring execution times on the available hardware and choosing the most performant option. To ensure the baseline and Telekine are running the same kernels for measurement purposes, we record the kernels chosen by the baseline, and hard-code those kernel choices for all runs.

Optimizing MXNet. We applied several optimizations to MXNet which help to mitigate the fact that Telekine is communicating with the GPU over a WAN:

- The models we evaluate represent the pixel channels of the input bitmaps using 4-byte floating point quantities, even though they range in integer values from 0 to 255. To save network bandwidth, we send bytes instead of floats, reducing bandwidth by 4×. Bytes are changed back floats on the GPU.
- We determined that MXNet was overly conservative in its GPU synchronization strategy and were able to reduce the number of synchronizations it performs by removing unnecessary calls to `hipStreamSyncronize` ("less sync" in Table 2). Telekine also optimizes synchronization calls by using tagged buffers (§5.1) to coordinate data transfers.

Machine learning training. We evaluate the training performance of deep neural networks on Telekine using three state-of-the-art convolutional neural network architectures: ResNet [37], InceptionV3 [91], and DenseNet [40]. All models are trained using the ImageNet dataset (a substantial data set consisting of 1.4 million training images). For ResNet, we use the 50-layer variant. For DenseNet, we use the 121-layer variant. We evaluated all networks using batches size of 64. Table 2 summarizes the input sizes that were used to evaluate the three network architectures.

Figure 6 shows the performance of training three neural nets on Telekine using the simulated tesdbed, normalized to the insecure baseline. The bars break down Telekine's overheads and match the descriptions from Section 7.1. Both Telekine and the baseline use a single GPU. Table 3 shows the same

Batch size	ResNet Base	ResNet Telekine	InceptionV3 Base	InceptionV3 Telekine	DenseNet Base	DenseNet Telekine
Simulated testbed						
1	20	273 (13.7x)	29	259 (8.93x)	26	248 (9.54x)
8	42	270 (6.43x)	65	264 (4.06x)	47	241 (5.13x)
64	233	389 (1.67x)	368	559 (1.52x)	246	405 (1.65x)
256	988	1195 (1.21x)	1520	1806 (1.19x)	946	1163 (1.23x)
Geodist testbed						
1	20	200 (10.0x)	31	205 (6.61x)	26	201 (7.73x)
8	69	241 (3.49x)	111	247 (2.23x)	84	209 (2.49x)
64	462	481 (1.04x)	637	685 (1.08x)	484	483 (1.00x)

Table 4. Latencies (in ms) of machine learning inference workloads with the baseline system (Base in the table) and Telekine.

experiment on the geodist testbed; the results are similar to the simulated testbed.

Machine learning inference. We evaluate neural network inference workloads for ResNet, InceptionV3, and DenseNet with Telekine. For inference, latency is the priority for users, but throughput is still a priority for providers. Batching inference can substantially improve throughput by fully utilizing hardware capabilities and amortizing the overheads from other system components [15]. We evaluate the latency of inference with different batch sizes, ranging from 1 to 256. Our baseline is an insecure server with one local GPU, communicating with the over the network. Table 4 shows the inference latency of three neural networks with different batch sizes. The overheads with on the simulated testbed for batches of size of 256 are 21%, 19%, and 23% for ResNet, InceptionV3, and DenseNet, respectively which are slightly improved compared to the overheads we report for training (§7.2), although the training batch size was 64. With a batch size of 64, the overheads on the simulated testbed inflate to 67%, 52%, and 65%. When we move to the geodist testbed, the performance of the baseline suffers more that Telekine; at batches of size 64, the standard deviation of our measurements exceed the differences between the mean Telekine and baseline runs. Clipper [15] uses an adaptive batch size to meet the latency requirement of the application, which Telekine could adopt.

7.3 Graph algorithms

Galois is a framework designed to accelerate parallel applications with irregular data access patterns, such as graph algorithms [75]. We port Galois's GPU computation to use the HIP runtime instead of CUDA and evaluate it on three graph algorithms: breadth-first search (BFS), PageRank, and single source shortest paths (SSSP). All measurements use the USA roads graph dataset [18]. Figure 5 shows the performance of these applications on Telekine with one and two GPUs. The baseline is an unmodified system with local GPU(s). Baseline performance for single GPU applications is: BFS 54.1s, SSSP 74.6s, Pagerank 60.9s; for two GPUs: BFS 36.4s, SSSP 42.8s. For the input distributed with Galois, two GPU Pagerank slows down, so we do not evaluate it.

Application		Normalized runtime
BFS	(1 GPU)	1.18x
SSSP	(1 GPU)	1.21x
Pagerank	(1 GPU)	1.29x
BFS	(2 GPUs)	1.38x
SSSP	(2 GPUs)	1.41x

Table 5. Performance of Galois applications with Telekine.

RTT (ms)	ResNet	InceptionV3	DenseNet
10	1.19x	1.10x	1.22x
20	1.29x	1.13x	1.37x
30	1.44x	1.16x	1.49x
40	1.53x	1.18x	1.66x
50	1.62x	1.30x	2.09x

Table 6. Normalized runtime of machine learning workloads with respect to network round trip time (RTT).

Telekine imposes moderate overheads on single-GPU Galois applications, adding latency to data transfer times. Galois implements each graph algorithm as a single GPU kernel that is iteratively called until the algorithm reaches termination. Multi-GPU applications exchange data between GPUs through the host after each iteration. Telekine imposes higher overheads for multi-GPU workloads because of increased data movement over the network.

7.4 WAN latency sensitivity

Telekine assumes that the client communicates with the server over a WAN. The greater distances crossed by WANs result in longer round trip times (RTTs). The batching of commands that Telekine does for security also makes it resilient to these increased RTTs, especially when the ratio of GPU computation to communication is high. To demonstrate this we increased the RTT between our machines using `netem` [65] and ran the machine learning training benchmarks for different RTTs (Table 6). Overheads increase with RTT. At 30ms which we measured to be the RTT between the client and an Amazon EC2 instance, the overhead for InceptionV3 is still only 16%.

8 Related Work

Enclave-based security. Several recently proposed systems aim to protect applications from an untrusted platform. Haven [7], SCONE [4], and Graphene-SGX [95] provide an environment to support unmodified legacy applications. Ryoan [43] protects user data from untrusted code and an untrusted platform. VC3 [83] and Opaque [109] provide SGX-protected data processing platforms. None of these systems allow for GPU computation and none of them focus on the communication issues that then arise.

Trusted execution environments on GPUs. HIX [45] extends an SGX-like design with duplicate versions of the enclave memory protection hardware to enable MMIO access from code running in an SGX enclave. This enables HIX to guarantee that a single enclave has exclusive access to the MMIO regions exported by a GPU, in principle, defeating a malicious

OS that wants to interpose or create its own mappings to them. While this design provides stronger GPU isolation than current enclaves, it remains vulnerable to side-channel attacks because communication is not data oblivious.

Graviton [100] supports GPU TEEs based on *secure contexts* that use the GPU command processor to protect memory from other concurrently executing contexts. Similar to Telekine, Graviton secures communication using cryptographic techniques. Telekine can adopt many of Graviton's clever mechanisms for its TEE functionality (§3.1), such as restricting access to GPU page tables without trusting the kernel driver. But Graviton does not protect against side channels, which is Telekine's primary mission.

The opportunity to provide stronger security for GPU-accelerated applications using TEEs and oblivious communication has been observed by others [41].

Securing accelerators. SUD emulates a kernel environment in user space to isolate malicious device drivers [9]. Previous work has explored techniques to support trusted I/O paths, leveraging hypervisor support [103, 110] or system management mode [52]. Our work focuses on the secure use of GPUs with untrusted system software and does not rely on support from the software at lower privilege layers. Border Control [74] addresses security challenges for accelerator-based systems but focuses on protecting the system from a malicious accelerator, rather than Telekine which protects CPU and GPU code from an untrusted platform.

GPU security and protection. Studies have analyzed GPU security properties and vulnerabilities [112]. Frigo et al. [28] demonstrate techniques that leverage integrated GPUs to accelerate side-channel attacks from browser codes using JavaScript and WebGL. PixelVault [98] exploits physical isolation between CPUs and GPUs to implement secure storage for keys, though it was shown to be insecure [112]. CUDA Leaks [77] shows techniques to exfiltrate data from the GPU to a malicious user. Attacks that take advantage of GPU memory reuse without re-initialization are a common theme [36, 56, 111]. Several systems have proposed mechanisms that bring the GPU under tighter control of system software, exploring OS support [34, 49, 63, 82], access to OS-managed resources [51, 86, 87], hypervisor support [20, 30, 33, 85, 90, 93, 101] and GPU architectural support for cross-domain protection [5, 13, 76, 79, 99].

Secure machine learning. Ohrimenko et al. describe an SGX-based system for multi-party machine learning on an untrusted platform [73]. Their data-oblivious algorithm for convolutional neural networks explicitly does not support state-of-the-art operations that are data dependent (e.g., max pooling). Telekine can support any data-dependent operations but requires a GPU TEE. Chiron [42] provides a framework for untrusted code to design and train machine learning models in SGX. Telekine does not support untrusted code, but does allow the use of GPUs which Chiron excludes. CQSTR [108] lets a *trusted* platform operator confine untrusted machine learning code so that it can be securely applied to user data. By contrast, Telekine protects user data from an untrusted platform operator. MLcapsule [35] protects service provider secrets (machine learning model) and client data by running machine learning algorithms in an SGX enclave but does not suggest extensions to allow secure GPU acceleration.

Slalom [94] secures training of DNNs using a combination of TEEs and local GPUs. Slalom's guarantees are achieved by partitioning DNN training into linear layers using matrix multiplication, which are offloaded to a GPU, the remaining operators, which execute on the CPU in a TEE such as SGX. Matrix multiplication is verified and turned private using algorithmic techniques [27], which enables secure GPU offload without requiring GPU TEE support.

Recent work [19, 61] demonstrates how to efficiently *apply* neural networks to encrypted data. As far as we know, today there are no practical techniques for *training* deep neural networks on encrypted data.

API remoting. API remoting [6, 22, 23, 50, 57, 58, 80, 104] is an I/O virtualization technique that interposes a high-level user-mode API. API calls are forwarded to a user-level computing framework [85] on a dedicated appliance VM [101], or on a remote server [23, 50]. To our knowledge, Telekine is the first system to use API remoting as a security technique.

OS-level time protection. Recent extensions to seL4 [29] suggest general OS-level techniques that prevent timing-based covert channels by eliminating sharing of hardware resources that can form the basis of covert channels. The techniques do not yet generalize to I/O-attached accelerators.

9 Conclusion

Telekine enables secure GPU acceleration in the cloud. Telekine protects in-cloud computation with a GPU TEE and application/library computation by placing it on a client machine. It secures their communication with a novel GPU stream abstraction that ensures the execution is independent of input data. Telekine allows GPU-accelerated workloads such as training machine learning models to leverage cloud GPUs while providing strong secrecy and integrity guarantees that protect the user from the platform's privileged software and its administrators.

10 Acknowledgements

We would like to thank Işil Dillig for her invaluable feedback on an early version of this paper. We would also like to thank Bryan Parno (our shepherd) and our anonymous reviewers; their constructive feedback made this paper better than it would have been otherwise. We gratefully acknowledge funding from NSF grants CNS-1900457, CNS-1618563, and CNS-1846169.

References

[1] Amazon. EC2 Dedicated Hosts. https://aws.amazon.com/ec2/dedicated-hosts/. (Accessed: February 12, 2020).

[2] AMD. ROCm, a New Era in Open GPU Computing. https://rocm.github.io/index.html. (Accessed: February 12, 2020).

[3] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards Verified, Constant-time Floating Point Operations. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1369–1382, 2018.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16. USENIX Association, 2016.

[5] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, MICRO'17. IEEE, 2017.

[6] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2019 IEEE International Conference on Cluster Computing Workshops and Posters*, CLUSTER WORKSHOPS, September 2010.

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14. USENIX Association, 2014.

[8] Bitfusion: The Elastic AI Infrastructure for Multi-Cloud. https://bitfusion.io. (Accessed: February 12, 2020).

[9] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10. USENIX Association, 2010.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Acoustic Cryptanalysis. *Journal of Cryptology*, 30, April 2017.

[12] cloc: Count Lines of Code. https://github.com/AlDanial/cloc. (Accessed: February 12, 2020).

[13] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. Supporting Address Translation for Accelerator-Centric Architectures. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA. IEEE, 2017.

[14] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[15] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17. USENIX Association, 2017.

[16] Richard Cziva and Dimitrios P Pezaros. On the Latency Benefits of Edge NFV. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS. IEEE, 2017.

[17] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *The Conference on Computer Vision and Pattern Recognition*, CVPR. IEEE, 2009.

[18] DIMACS. 9th DIMACS Implementation Challenge - Shortest Paths. http://users.diag.uniroma1.it/challenge9/download.shtml, 2005. (Accessed: February 12, 2020).

[19] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy. In *International Conference on Machine Learning*, 2016.

[20] Micah Dowty and Jeremy Sugerman. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[21] J. Duato, AJ. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. rCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters. In *2010 International Conference on High Performance Computing Systems*, HPCS, 2010.

[22] José Duato, Francisco D Igual, Rafael Mayo, Antonio J Peña, Enrique S Quintana-Ortí, and Federico Silla. An efficient implementation of GPU virtualization in high performance clusters. In *European Conference on Parallel Processing*, Euro-Par'09, pages 385–394, Berlin, Heidelberg, 2009. Springer, Springer-Verlag.

[23] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling CUDA acceleration within virtual machines using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[24] Morris Dworkin. NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf, 2007. (Accessed: February 12, 2020).

[25] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using Verification to Disentangle Secure-enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 287–305, New York, NY, USA, 2017. ACM.

[26] Denis Foley. Ultra-Performance Pascal GPU and NVLink Interconnect. In *HotChips*, 2016.

[27] Rusins Freivalds. Probabilistic Machines Can Use Less Running Time. In *IFIP Congress*, pages 839–842, 1977.

[28] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In *IEEE Symposium on Security and Privacy*, May 2018.

[29] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *European Conference in Computer Systems*, EuroSys, 2019.

[30] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU transparent virtualization component for high performance computing clouds. In *European Conference on Parallel Processing*, pages 379–391. Springer, Springer, 2010.

[31] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, 2017.

[32] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating Financial Applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 127–136, New York, NY, USA, 2013. ACM.

[33] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[34] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 3–3. USENIX Association, 2011.

[35] Lucjan Hanzlik, Yang Zhang, Kathrin Grosse, Ahmed Salem, Max Augustin, Michael Backes, and Mario Fritz. MLCapsule: Guarded Offline Deployment of Machine Learning as a Service. *CoRR*, abs/1808.00590, 2018.

[36] Ari B. Hayes, Lingda Li, Mohammad Hedayati, Jiahuan He, Eddy Z. Zhang, and Kai Shen. GPU Taint Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 209–220. USENIX Association, 2017.

[37] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[38] Nicole Hemsoth. Medical Imaging Drives GPU Accelerated Deep Learning Developments. https://www.nextplatform.com/2017/11/27/medical-imaging-drives-gpu-accelerated-deep-learning-developments/, November 2017. (Accessed: February 12, 2020).

[39] HIP: Convert CUDA to Portable C++ Code. https://github.com/ROCm-Developer-Tools/HIP. (Accessed: February 12, 2020).

[40] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017.

[41] Tyler Hunt, Zhipeng Jia, Vance Miller, Christopher J. Rossbach, and Emmett Witchel. Isolation and Beyond: Challenges for System Security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 96–104, New York, NY, USA, 2019. ACM.

[42] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *CoRR*, abs/1803.05961, 2018.

[43] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 533–549. USENIX Association, 2016.

[44] Intel(R) Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014. (Accessed: February 12, 2020).

[45] Insu Jang, Adrian Tang, Taehoo Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19, 2019.

[46] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A complete key recovery timing attack on a GPU. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[47] Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A Novel Side-Channel Timing Attack on GPUs. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, GLSVLSI '17, pages 167–172, New York, NY, USA, 2017. ACM.

[48] Simon Johnson. Intel SGX and Side-Channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels, March 2017. (Accessed: February 12, 2020).

[49] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, USENIXATC'11, pages 17–30. USENIX Association, 2011.

[50] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, page 341352. ACM, 2012.

[51] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216. USENIX Association, 2014.

[52] Yonggon Kim, Ohmin Kwon, Jinsoo Jang, Seongwook Jin, Hyeongboo Baek, Brent Byunghoon Kang, and Hyunsoo Yoon. On-demand bootstrapping mechanism for isolated cryptographic operations on commodity accelerators. 62, 7 2016.

[53] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *CoRR*, January 2018.

[54] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, pages 388–397, Berlin, Heidelberg, 1999. Springer-Verlag.

[55] Dayeol Lee, David Kohlbrenner, Kevin Cheang, Cameron Rasmussen, Kevin Laeufer, Ian Fang, Akash Khosla an Chia-Che Tsai, Sanjit Seshia, Dawn Song, and Krste Asanovic. Keystone Enclave: An Open-Source Secure Enclave for RISC-V. https://keystone-enclave.org/files/keystone-risc-v-summit.pdf, 2018. (Accessed: February 12, 2020).

[56] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 19–33, Washington, DC, USA, 2014. IEEE Computer Society.

[57] Teng Li, Vikram K Narayana, Esam El-Araby, and Tarek El-Ghazawi. GPU resource sharing and virtualization on high performance computing systems. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 733–742. IEEE, 2011.

[58] Tyng-Yeu Liang and Yu-Wei Chang. GridCuda: A Grid-Enabled CUDA Programming Toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146, March 2011.

[59] Arm Limited. Introducing Arm TrustZone. https://developer.arm.com/technologies/trustzone. (Accessed: February 12, 2020).

[60] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Dkaniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, January 2018.

[61] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious Neural Network Predictions via MiniONN transformations. Cryptology ePrint Archive, Report 2017/452, 2017. http://eprint.iacr.org/2017/452.

[62] Ramya Jayaram Masti, Devendra Rai, Aanjhan Ranganathan, Christian Müller, Lothar Thiele, and Srdjan Capkun. Thermal Covert Channels on Multi-core Platforms. In *USENIX Security Symposium*, 2015.

[63] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM.

[64] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[65] netem. https://wiki.linuxfoundation.org/networking/netem, 2019. (Accessed: February 12, 2020).

[66] NVIDIA. CUDA Zone. https://developer.nvidia.com/cuda-zone. (Accessed: February 12, 2020).

[67] NVIDIA. Driving Innovation: Building AI-Powered Self-Driving Cars. https://www.nvidia.com/en-us/self-driving-cars/. (Accessed: February 12, 2020).

[68] NVIDIA. RISC-V Story. https://riscv.org/wp-content/uploads/2016/07/Tue1100_Nvidia_RISCV_Story_V2.pdf. (Accessed: February 12, 2020).

[69] NVIDIA. GPUs and DSLs for Life Insurance Modeling. https://devblogs.nvidia.com/gpus-dsls-life-insurance-modeling/, March 2016. (Accessed: February 12, 2020).

[70] NVIDIA. Microsoft Sets New Speech Recognition Record. https://news.developer.nvidia.com/microsoft-sets-new-speech-recognition-record/, August 2017. (Accessed: February 12, 2020).

[71] NVIDIA. NVIDIA CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html, 2017. (Accessed: February 12, 2020).

[72] NVIDIA. CUDA Toolkit Documentation (Streams). https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#streams, 2018. (Accessed: February 12, 2020).

[73] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Sebastian Nowozin Aastha Mehta, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security Symposium*, 2016.

[74] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border Control: Sandboxing Accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 470–481, New York, NY, USA, 2015. ACM.

[75] Sreepathi Pai and Keshav Pingali. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM.

[76] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, 2014.

[77] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. *ACM Trans. Embed. Comput. Syst.*, 15(1):15:1–15:25, January 2016.

[78] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Computing Surveys*, 51(6), 2019.

[79] Jonathan Power, Mark D Hill, and David A Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *HPCA*, 2014.

[80] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Orti. CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[81] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, 2009.

[82] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Symposium on Operating Systems Principles*, SOSP'11, pages 233–248. ACM, 2011.

[83] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[84] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *CoRR*, abs/1905.05726, 2019.

[85] Lin Shi, Hao Chen, and Jianhua Sun. vCUDA: GPU accelerated high performance computing in virtual machines. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009.

[86] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 32, March 2013.

[87] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: Integrating a File System with GPUs. *ACM Transactions on Computer Systems*, 32(1), 2014.

[88] Erik Smistad, Thomas L. Falch, Mohammadmehdi Bozorgi, Anne C. Elster, and Frank Lindseth. Medical image segmentation on GPUs A comprehensive review. *Medical Image Analysis*, 20(1):1–18, 2015.

[89] Matthew J.A. Smith, Mikayel Samvelyan, and Tabish Rashid. Using AI to Solve Collaborative Challenges by Playing StarCraft. https://news.developer.nvidia.com/using-ai-to-solve-collaborative-challenges-by-playing-starcraft/. (Accessed: February 12, 2020).

[90] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *USENIX ATC*, USENIX ATC'14, pages 109–120. USENIX Association, 2014.

[91] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[92] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling Preemptive Multiprogramming on GPUs. In *ISCA*, 2014.

[93] Kun Tian, Yaozu Dong, and David Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *USENIX ATC*, pages 121–132, 2014.

[94] Florian Tramè and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations*, ICLR '19, 2019.

[95] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 645–658. USENIX Association, 2017.

[96] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[97] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think. In *USENIX Security*, August 2018.

[98] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1131–1142, New York, NY, USA, 2014. ACM.

[99] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[100] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[101] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. GPU virtualization for high performance general purpose computing on the ESX hypervisor. In *Proceedings of the High Performance Computing Symposium*, page 2. Society for Computer Simulation International, 2014.

[102] Vultr.com. https://www.vultr.com/products/cloud-compute/. (Accessed: November 2019).

[103] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 261–268, New York, NY, USA, 2017. ACM.

[104] Shucai Xiao, Pavan Balaji, James Dinan, Qian Zhu, Rajeev Thakur, Susan Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu-chun Feng. Transparent accelerator migration in a virtualized GPU environment. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid, pages 124–131, 2012.

[105] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[106] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog Computing: Platform and Applications. In *ACM/IEEE Workshop on Hot Topics in Web Systems and Technologies*, HotWeb, 2015.

[107] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated Virtualization of Accelerators. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[108] Yan Zhai, Lichao Yin, Jeffrey S Chase, Thomas Ristenpart, and Michael M Swift. CQSTR: Securing Cross-Tenant Applications with Cloud Containers. In *ACM Symposium on Cloud Computing*, 2016.

[109] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.

[110] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[111] Zhe Zhou, Wenrui Diao, Xiangyu Liu, Zhou Li, Kehuan Zhang, and Rui Liu. Vulnerable GPU Memory Management: Towards Recovering Raw Data from GPU. *PoPETs*, 2017(2):57–73, 2017.

[112] Zhiting Zhu, Sangman Kim, Yuri Rozhanski, Yige Hu, Emmett Witchel, and Mark Silberstein. Understanding The Security of Discrete GPUs. In *Proceedings of the General Purpose GPUs*, GPGPU-10, pages 1–11, New York, NY, USA, 2017. ACM.