Safe and Practical GPU Computation in TrustZone

Heejin Park*
Apple
bakhi@apple.com

Felix Xiaozhu Lin
University of Virginia
felixlin@virginia.edu

Abstract

For mobile devices, it is compelling to run sensitive GPU computation within a TrustZone trusted execution environment (TEE). To minimize GPU software deployed in TEE, the replay approach is promising: record CPU/GPU interactions on a full GPU stack outside the TEE; replay the interactions inside the TEE without the GPU stack. A key dilemma is that the recording process must both (1) occur in a safe environment and (2) access the *exact* GPU models to be used for replay. To this end, we present a novel recording architecture called GR-T: a mobile device possessing the GPU hardware collaborates with a GPU-less cloud service which runs the GPU software; the two parties exercise the GPU hardware/software jointly for recording. To overcome the resultant network delays, GR-T contributes optimizations: register access deferral, speculation, and meta-only synchronization. These techniques reduce the recording delay by 20x, from hundreds of seconds to tens of seconds. Replay-based GPU computation incurs 25% lower delays compared to native execution outside TEE. The code is available at https://github.com/bakhi/GPUReplay.

CCS Concepts: • **Security and privacy** → **Systems security**; *Operating systems security*; *Mobile platform security*; *Trusted computing*.

Keywords: Secure GPU computation; Record and replay; Dry run; GPU stack; TrustZone; TEE

ACM Reference Format:
Heejin Park and Felix Xiaozhu Lin. 2023. Safe and Practical GPU Computation in TrustZone. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567483

*This work is done when the author was at Purdue University.

1 Introduction

Arm TrustZone is a trusted execution environment (TEE) where sensitive-code is isolated from the untrusted OS, ensuring the execution's confidentiality and integrity. While TrustZone is already capable of isolating GPU *hardware* [15, 44], the biggest obstacle is the GPU *software* stack (GPU stack[1] for short), which is large [46] and known for vulnerabilities [4, 5, 60]. Existing techniques transform the GPU stack [71] or workloads [7, 61, 69] to suit TEE; they however incur high engineering efforts and compatibility loss, as will be analyzed in Section 2.

Our recent work [57] (referred to as GR below) sheds light on how to deploy a lean GPU stack within TrustZone TEEs via GPU record/replay [14, 35, 41, 70]. Interposing the CPU/GPU boundary, GR executes a GPU workload $\mathcal{W}$, e.g. neural network inference, in two phases. (1) The record phase runs $\mathcal{W}$ on a full GPU stack and logs CPU/GPU interactions as a series of register accesses and memory dumps. (2) The replay phase runs $\mathcal{W}$ by replaying the pre-recorded CPU/GPU interactions on new input without needing a GPU stack. Of the two phases, the recording can be done in a safe environment outside of TEE; after the recording is done *once*, the replay can recur within the TEE on new input *repeatedly*. The replayer can be as simple as a few KSLoC, has little external dependency, and contains no vulnerabilities commonly seen in a GPU stack [2, 4, 5].

A key missing piece in applying GR to mobile devices is practicality. GR hinges on recording specific to the GPU hardware models (often called GPU SKUs) of the target mobile devices to reproduce GPU computation. Unfortunately, ML developers cannot run the recorder on target devices as mobile OSes are untrusted. They must generate recordings on a separate, trustworthy machine as shown in Figure 1(a). Doing so, however, *early-binds* GPU code to specific GPU SKUs, deviating from the common practice of *late binding*. With late binding, developers ship GPU code in hardware-neutral formats such as in OpenCL or Metal, which is later JIT-compiled on the target devices for specific GPU SKUs. Because of early binding, GR requires developers to foresee GPU SKUs on which their workloads may run, own such SKUs, and produce/ship per-SKU recordings. This burdens developers, considering as many as 80 GPU SKUs on today's smartphones (Figure 3).

Key idea We present a new approach called GR-T, allowing a mobile device (i.e. the client) to leverage the cloud for

[1]We stress that the GPU stack is software running on *CPU*.

GPU recording. As shown in Figure 1 (b), the cloud hosts the mobile GPU software stack without any GPU hardware. To record, the client TEE requests the cloud to run a GPU workload $\mathcal{W}$. The cloud "dry runs" $\mathcal{W}$ on its GPU stack while tunneling all the resultant CPU/GPU interactions to the physical GPU protected within the client TEE. The cloud logs all the interactions as a *recording* for $\mathcal{W}$, which the client downloads afterwards. In future executions of $\mathcal{W}$, the client TEE replays the recording on the protected GPU without invoking the cloud. With GR-T, the cloud recorder accesses exact, diverse GPU SKUs without the hassle of hosting the SKUs in the cloud.

Why can a remote cloud service be trusted for recording? (1) The record phase, by design, does not require the workload's exact input (§2.3). As such, the client TEE never sends out sensitive data such as ML input and model parameters. (2) The TEE expects the cloud GPU stack to be *integral*, for which the cloud can offer high assurance: it is managed with rigorous security measures [32]; it can be attested remotely via techniques including Intel SGX and AMD SEV; sealed in a VM, a GPU stack instance in the cloud *exclusively* serves one client through a narrow interface – encrypted communication. This is much more secure than running the GPU stack on a client's untrusted OS, which serves a myriad of third party apps and faces various threats including malware and misconfiguration. See Section 7.1 for a security analysis.

Challenges and Designs The main challenge to GR-T arises from spanning the GPU stack (running on CPU) and the GPU hardware over the wireless Internet. An ML workload induces frequent CPU/GPU interactions including accesses to GPU registers, shared memory accesses, and interrupts. When CPU and GPU are co-located on the same device, each interaction takes no more than microseconds. Over a wireless connection, however, naively forwarding each interaction takes milliseconds or seconds. Such a long delay would preclude GPU recording due to frequent software/hardware timeouts; it would render GR-T unusable due to formidable recording delays.

To overcome the long delays, our insights are twofold. (1) The sequence of GPU register accesses consists of many *recurring segments*, induced by repeated invocations to GPU driver routines, e.g. job submission and GPU cache flush. By learning these access patterns, the cloud service can speculate most register accesses and their outcomes even before the client responds. (2) Unlike cloud offloading [20] in which the cloud must produce correct computation results, for recording the cloud only has to dry run the GPU stack, extracting the CPU/GPU interactions of interest.

With the insights, GR-T automatically instruments the GPU driver code for the following mechanisms.

(1) *Register access deferral.* Although each register access was intended to execute on physical GPU synchronously, the cloud service queues and commits multiple accesses to the client GPU in a batch, coalescing their network round trips. Since the register accesses interleave with a driver's execution in program order, the cloud service represents the values of pending register reads as symbols and executes the driver symbolically. After the register accesses are completed, the cloud replaces the symbolic variables with concrete register values.

(2) *Register access speculation.* To further mask the network delay of a commit, the cloud service predicts outcomes of register reads. Without waiting for the client to finish a commit, the cloud continues its GPU driver execution with the predicted register values, and validates the prediction after the client returns the actual register values. In case of misprediction, both the cloud and the client roll back to their most recent valid states.

(3) *Meta-only synchronization.* Despite of being physically distributed, the cloud and the client must maintain a synchronized view of the CPU/GPU shared memory. GR-T reduces the synchronization *frequencies* by tapping in GPU's hardware events; GR-T reduces the synchronization *traffic* by only synchronizing GPU's metastate – GPU shaders, command lists, and job descriptions – while omitting the program data. Essentially, GR-T gives up computation correctness while faithfully preserving the semantics of CPU/GPU interactions.

Results We build GR-T atop Armv8 SoCs and Mali Bifrost, a popular family of mobile GPUs, and evaluate it on a series of ML workloads. Compared to naive forwarding, GR-T lowers

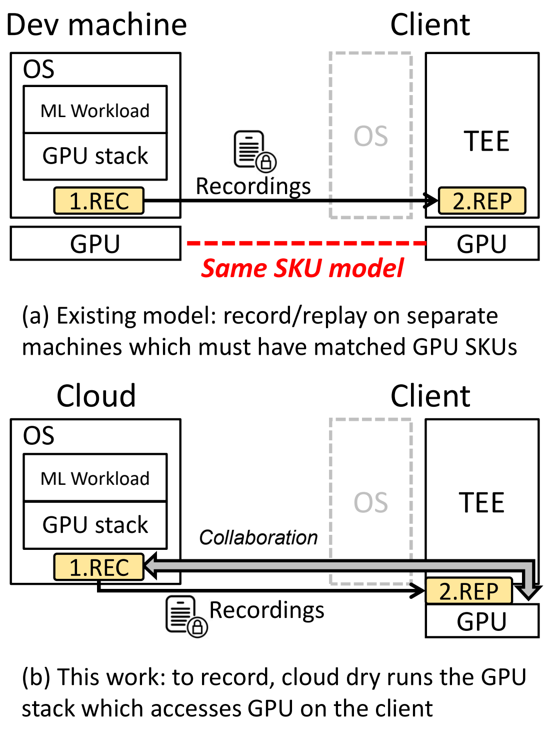

Dev machine
Client
OS
ML Workload
GPU stack
1.REC
Recordings
OS
TEE
2.REP
GPU
GPU
Same SKU model
(a) Existing model: record/replay on separate machines which must have matched GPU SKUs
Cloud
Client
OS
ML Workload
GPU stack
Collaboration
1.REC
OS
TEE
2.REP
GPU
Recordings
(b) This work: to record, cloud dry runs the GPU stack which accesses GPU on the client

Figure 1. A comparison between (a) the existing GR model and (b) GR-T (this work). Figure 4 shows GR-T in detail.

the recording delays by more than one order of magnitude, from several hundred seconds down to tens of seconds; it reduces the client energy consumption by up to 99%. Its replay incurs 25% lower delays as compared to insecure, native execution outside TEE.

Contributions We present a holistic solution for GPU computation within the TrustZone TEE. We address the key missing piece – a safe, practical recording process. We make the following contributions.

- A novel architecture called GR-T, where the cloud and the client TEE collaboratively exercise the GPU software/hardware for recording.
- A suite of key I/O optimizations that exploit GPU-specific insights in order to overcome the long network delays between the cloud and the client.
- A concrete implementation for practicality: lightweight instrumentation of the GPU driver; crafting device trees for cloud VMs to operate remote GPUs; a TEE module managing GPU for record and replay.

2 Motivations

2.1 Mobile GPUs

This paper focuses on mobile GPUs commonly seen on mobile and embedded SoCs. Today, a mobile GPU has its own MMU and shares the main memory with the CPU.

GPU stack and execution workflow A modern GPU stack consists of ML frameworks (e.g. Tensorflow), a userspace runtime for GPU APIs (e.g. OpenCL), and a GPU driver in the kernel. Figure 4 shows details.

When an app executes ML workloads, it invokes GPU APIs, e.g. OpenCL. Accordingly, the runtime prepares GPU jobs and input data: it emits GPU commands, shaders, and data to the shared memory which is mapped to the app's address space. The driver sets up the GPU's page tables that store the shared memory mapping between the GPU's virtual and the physical addresses; it configures GPU hardware and submits the GPU jobs. The GPU then loads the job shader code and data from the shared memory, executes the code, and writes back compute results and job status to the memory. After the job, the GPU raises an interrupt to the CPU. For throughput, the GPU stack often supports multiple outstanding jobs.

CPU/GPU interact through three channels:

- Registers, for configuring GPU and controlling jobs.
- Shared memory, to which CPU deposits commands, shaders, and data and retrieves compute results. Modern GPUs have dedicated page tables, allowing them to access shared memory using GPU virtual addresses.
- GPU interrupts, which signal GPU job status.

The GPU driver manages these interactions; thus it can interpose and log these interactions.

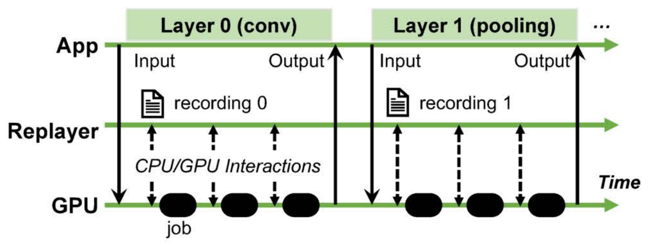


Figure 2. A timeline for replaying NN inference.

2.2 Prior approaches

Towards GPU compute inside the TrustZone TEE, prior approaches are inadequate.

Porting GPU stack to TEE One approach is to pull the GPU stack to the TEE ("lift and shift") [36, 51]. The biggest problem is the clumsy GPU stack: the stack spans large codebases (e.g. tens of MB binary code), much of which are proprietary. The stack depends on POSIX APIs which are unavailable inside TrustZone TEE. For these reasons, it will be a daunting task to port proprietary runtime binaries and a POSIX emulation layer, let alone the GPU driver. *Partitioning* the GPU stack and porting part of it, as suggested by recent works [34, 71], also see significant drawbacks: they still require high engineering efforts and sometimes even custom hardware. The ported GPU code is likely to introduce vulnerabilities to the TEE [1, 3, 4], bloating the TEE and weakening security.

Outsourcing Another approach is for TEE to invoke an external GPU stack. One choice is to invoke the GPU stack in the normal-world OS of the same device. Because the OS is untrusted, the TEE must prevent it from learning ML data/parameters and tampering with the result. Recent techniques include homomorphic encryption [25, 69], ML workload transformation [29, 43], and result validation [17]. They support limited GPU operators and often incur significant efficiency losses.

2.3 GR for TrustZone

Unlike prior approaches, GR provides a new execution paradigm [57]. (1) In the record phase, app developers run their ML workload *once* on a trusted GPU stack; a recorder at the CPU/GPU boundary logs all the CPU/GPU interactions – register accesses, GPU memory dumps, and interrupt events. (2) In the replay phase, a target app invokes a simple replayer to reproduce the logged computation on new input data.

Example Figure 2 exemplifies how GR works for neural network (NN) inference. To record, developers run the NN inference once and produce a sequence of recordings, one for each NN layer; each recording encloses multiple GPU jobs for an NN layer. To replay, a target ML app executes the recordings in the layer order. The granularity of recordings is a developers' choice as the tradeoff between composability and efficiency. Alternatively, developers may create one monolithic recording for all the NN layers (not shown in the figure).

Why GR works? It addresses three design concerns [57] below.
(1) ***Completeness***. Happening at the CPU/GPU boundary, recording transparently captures all the information needed to reproduce the original GPU computation: GPU commands, shaders, page tables, and job input/output addresses. For instance, CPU's dynamic updates to the GPU address space are recorded in snapshots of GPU page tables; GPU memory layout is recorded as memory dumps; CPU/GPU synchronization, e.g. OpenCL barriers, is recorded as register polling for GPU interrupts.
(2) ***Determinism***. When recording, GR takes steps to forestall nondeterministic CPU/GPU interactions such as job submission timing and concurrency. The recorder/replayer serialize job executions and avoid concurrent GPU jobs; during record and replay, only one app can access the GPU. This makes CPU/GPU interactions deterministic, ensuring that the replayer can faithfully reproduce recorded computation.
(3) ***Independence of input***. Common ML workloads (e.g. CNN and RNN) often have static graphs of GPU jobs; no conditional branches exist *among jobs*, a key property exploited in prior work [72]. A workload invokes the same set of GPU jobs regardless of its input data. Therefore, a single record run can exercise all GPU jobs in a workload and capture them. Once recorded, the same GPU jobs can be reproduced repeatedly: the replayer injects a new input to the recorded input address and can later retrieve the corresponding output from the recorded output address.

2.4 The problem of recording

Requisites for recording environment To apply GR to TrustZone, a missing component is the recording environment where the GPU stack is exercised and recordings are produced. First, the recording environment must be accessible to the exact GPU SKU to be used for replay in the future. In our experience, even subtle SKU differences can break replay. Examples include variations in GPU hardware resources, e.g. shader core count, which determines how the JIT compiler generates and optimizes GPU shaders; variations in GPU page table formats; variations in shared memory layout, with which GPU communicates its execution status with CPU. Second, it must be trustworthy to guarantee the GPU compute integrity. A corrupted recording may induce unintended replay outcomes and even break the entire TEE's integrity. This precludes *local* recording on the client device as its normal-world OS is untrusted, often managed by novice users and exposed to malware and clickbait.

Alternative recording environments Developers may choose the following recording environments, *remote* from clients that still fall short of portability and practical use.

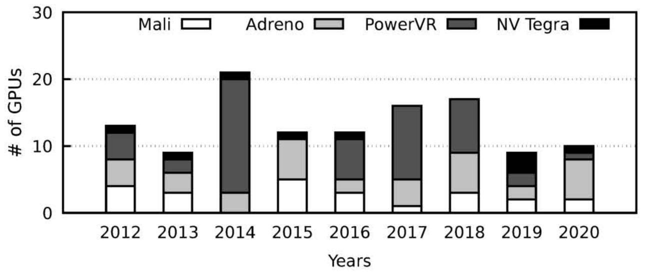


Figure 3. Numbers of new mobile GPU SKUs per year [24], showing the diversity of mobile GPUs.

(1) ***Developer's machines.*** App developers may maintain various GPU SKUs and produce per-SKU recordings for target clients. However, they must cope with diverse, ever-changing mobile hardware [63, 73]. Figure 3 highlights the diversity of today's mobile GPUs [38]: around 80 SKUs are seen on today's smartphones; no SKUs are dominating; new SKUs are rolled out frequently. It is impractical for app developers, e.g. those developing video analytics or activity recognition as secure extensions to their apps, to foresee all possible GPU SKUs on clients and possess them. More importantly, since recordings are SKU-specific, distributing them among clients violates today's common practice of distributing ML apps – shipping GPU programs in hardware-neutral formats for portability. All major ML frameworks we know follow this common practice, e.g. TFLite shipping OpenCL/GL/Metal shaders [42] and ncnn shipping Vulkan shaders [68].
(2) ***Mobile device farm in the cloud.*** While such a device farm relieves developers' burden, managing a large, diverse collection of mobile devices in the cloud is impractical. Not designed to be hosted, mobile devices do not conform to the size, power, and thermal requirements of data centers. The device farm is not elastic: a device can serve one client at a time; planning the capacity and device types is difficult. As new mobile devices emerge every few months, the total cost of ownership is high.

3 GR-T

We advocate for a new recording approach: when a workload is executed for the first time, dry run the GPU stack in the cloud while using GPUs on the clients.

3.1 The workflow

The GR-T workflow is as follows. (1) Developers write ML apps as usual, e.g. MNIST inference atop Tensorflow; they ship GPU programs in hardware neutral formats as usual. They are oblivious to the TEE, the GPU SKU, and the cloud service. (2) Before executing the workload for the first time, the client TEE requests the cloud service to dry run the workload. As the cloud runs the GPU stack, it forwards GPU hardware access to the client TEE and receives the GPU's response from the latter. In the meantime, the cloud records all the CPU/GPU interactions. (3) For actual executions of the ML workload, the client TEE replays the recorded CPU/GPU interactions on new input (exploiting input independence, §2.3); it no longer invokes the cloud. For security, the cloud *never* caches and reuses recordings across clients even if they have the same GPU SKU.

Our approach fundamentally differs from remote I/O or I/O-Device-as-a-Service [64]. Our goal is neither to execute GPU compute in the cloud [18, 21] nor run the GPU stack *precisely* in the cloud, e.g. software testing [67]. It is to extract the CPU's stimuli to GPU and the GPU's response. Our unique goal allows novel optimization to be described later.

Why use the cloud for recording? The cloud has the following benefits.

(1) *Safe.* The cloud hosts the GPU stack in an environment subject to rigorous security measures [32] and attestation. Each GPU stack instance is sealed in a dedicated VM, serving one authenticated client exclusively. This contrasts to the client OS where the GPU stack is shared by many apps and constantly faces threats such as malware and misconfiguration. See Section 7.1 for a detailed security analysis.

(2) *Rich resources.* The cloud can run a GPU stack that is too big to fit in the TEE; it can also host multiple GPU stack variants, catering to different APIs and frameworks used by ML workloads.

Can the cloud emulate GPUs? One may wonder if the cloud runs software-based GPU emulators [23] without the need for physical GPUs on clients. However, precise emulation of modern GPUs is hard: they are diverse; they often have undisclosed behaviors, interfaces, and hardware quirks.

Will the cloud have too many GPU drivers? While the cloud VMs need to host drivers for all GPU SKUs on clients, the total number of needed GPU drivers will be small. This is because a single GPU driver often supports many GPU SKUs of the same family [12, 13]; SKUs share much driver logic while differing only in register definitions, hardware revisions, and erratum. For instance, Mali Bifrost and Qualcomm Adreno 6xx drivers each support 6 and 7 GPUs [10, 47]. As Section 6 will show, by instructing the kernel device tree, we can incorporate multiple GPU drivers in one unified Linux kernel image to be used by the cloud VMs.

Why is GR-T practical? One may wonder if OS and device vendors are in a good position for crafting in-TEE GPU stacks. However, doing so would require deep customization across multiple parties (e.g. vendors of OS, TEEs, GPUs, and SoCs); in particular, the hardware vendors are secretive about their IPs. Even if they are willing to, they still face challenges in re-architecting the complex GPU stacks. By contrast, GR-T does not require such deep, cross-domain cooperation. Unlike alternative recording environments (§4.2), GR-T alleviates the burden of maintaining the current and future GPU SKUs from both the cloud provider and ML developers.

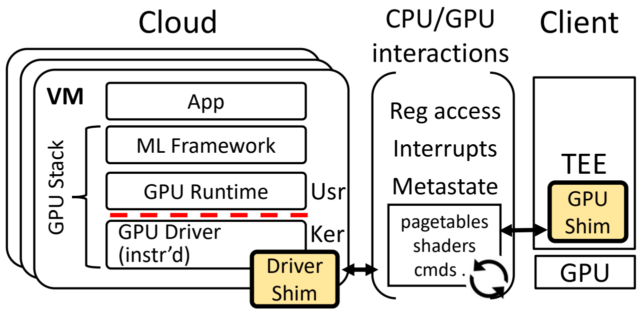


Figure 4. GR-T's online recording. The cloud collaborates with the client to dry run the GPU stack.

Limitations GR-T requires an Internet connection to function: it cannot create recordings when the client device is offline. The poor network condition can slow down the entire recording process. To record a workload, the TEE must faithfully allocate the same amount of memory as needed by the workload's actual run. As the secure memory available to TrustZone is typically pre-configured small [52, 53, 58], GR-T may need the SoC firmware to enlarge the secure memory for recording a high-memory ML workload.

Broader applicability While we show GR-T for recording, its optimizations can be used for remote debugging [67]. For instance, by comparing a client's GPU register logs and memory dumps with the ones from the cloud, the cloud may detect and report firmware malfunctioning and vendors may troubleshoot remotely. As replay has been used on IO devices other than GPU [30], our techniques can be used for generating recordings for these IO without possessing the actual IO hardware.

3.2 The GR-T architecture

Figure 4 shows the architecture. The cloud service manages multiple VM images corresponding to variants of GPU stack. The VM is lean, containing a kernel and the minimal software required by the GPU stack. Once launched, a VM is dedicated to serving only one client TEE. Neither a VM nor a recording is shared across clients. All the communication between the cloud VM and the TEE is authenticated and encrypted.

GR-T's recorder comprises two shims for the cloud (DriverShim) and the client TEE (GPUShim). DriverShim at the bottom of the GPU stack interposes access to the GPU hardware. It is implemented by automatic instrumenting of the GPU driver, injecting code to register accessors and interrupt handlers. GPUShim, instantiated as a TEE module, isolates the GPU during recording and prevents normal-world access.

After a record run, DriverShim processes logged interactions as a recording; it signs and sends the recording back to the client. To replay, the client TEE loads a recording, verifies its authenticity, and executes the enclosed interactions. During replay, the TEE isolates the GPU; before and after the replay, it resets the GPU and cleans up all the hardware state.

3.3 Challenge: long network delays

A GPU stack is designed with the premise that CPU and GPU co-locate on an on-chip interconnect with sub-microsecond delays. GR-T breaks the premise by distributing CPU/GPU over the Internet with tens of ms or even seconds of delays. The impacts are twofold. (1) GPU register accesses are translated into numerous network round trips. Taking MNIST inference as an example, the GPU driver issues 2800 register accesses, each requiring a round trip. (2) Long round trip time (RTT) makes memory synchronization slow. By design, CPU and GPU exchange extensive information via shared memory: commands, shader code, and input/output data. Since they are distributed with no shared physical memory, maintaining such a shared memory *view* can be prohibitively slow. As we will show in Section 5, classic distributed shared memory (DSM) misses a key opportunity in dry run.

The long recording delay, often hundreds of seconds (§7), renders GR-T unusable. (1) We observed that the GPU stack constantly throws exceptions and GPU resets or freezes from time to time. This is because the long delays violate many timing assumptions implicitly made by the stack code and the GPU firmware. (2) As the TEE has to exclusively lock the GPU for a record run, it blocks the normal-world apps from accessing the GPU for long and hurts the system interactivity. (3) As each record run (per client, per workload) requires a *dedicated* cloud VM for hundreds of seconds, GR-T is less cost-effective. (4) An ML workload has to wait long before its first execution.

4 Hiding Register Access Delays

To overcome long network delays, we retrofit known I/O optimizations to exploit new opportunities in mobile GPUs.

4.1 Register access deferral

Problem By design, a GPU driver weaves GPU register accesses into its instruction stream, executing register accesses and CPU instructions synchronously in program order. For example in Listing 1(a), the driver cannot issue the second register access (line 4) until the first access (line 3) and the preceding CPU instructions are completed. The synchronous register access leads to numerous network round trips. This is exacerbated by the fact that GPU register accesses are dominated by reads (more than 95% in our measurement), which cannot be simply buffered as writes. This is in Figure 5(a).

Basic idea We coalesce the round trips by making register accesses asynchronous: as shown in Figure 5(b), DriverShim defers register accesses as the driver executes, until the driver

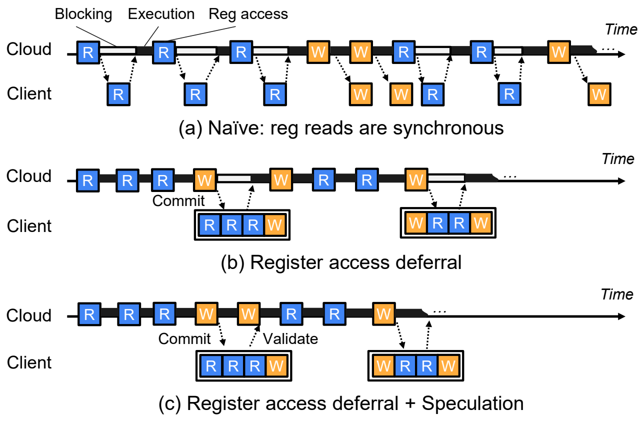

Blocking
Execution
Reg access
Time
Cloud
Client
R
W
(a) Naïve: reg reads are synchronous
Commit
(b) Register access deferral
Validate
(c) Register access deferral + Speculation

Figure 5. GR-T's strategies for hiding long RTTs.

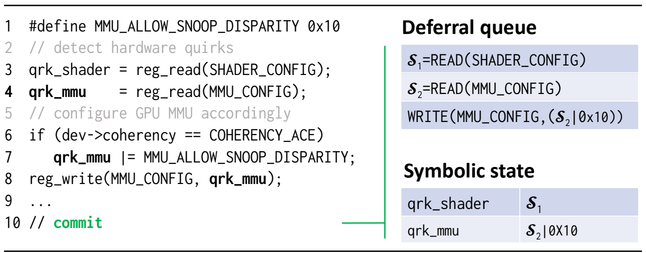
cannot continue execution without the value from a deferred register read. DriverShim then *synchronously* commits all deferred register accesses in a batch to the client GPU. After the commit, DriverShim stalls the driver execution until the client GPU returns the register access results.

To implement the mechanism, DriverShim injects the deferral hooks into the driver via automatic instrumentation. The driver *source code* remains unmodified.

Key mechanisms for correctness First, DriverShim keeps the deferral transparent to the client and its GPU. For correctness, the GPU must execute the same sequence of register accesses as if there was no deferral. The register accesses must be in their *exact* program order, because (1) GPU is stateful and (2) these accesses may have hidden dependencies. For instance, read from an interrupt register may clear the GPU's interrupt status, which is a prerequisite for a subsequent write to a job register. For this reason, DriverShim queues register accesses in their program order. It instantiates one queue *per kernel thread*, which is important to the memory model to be discussed later.

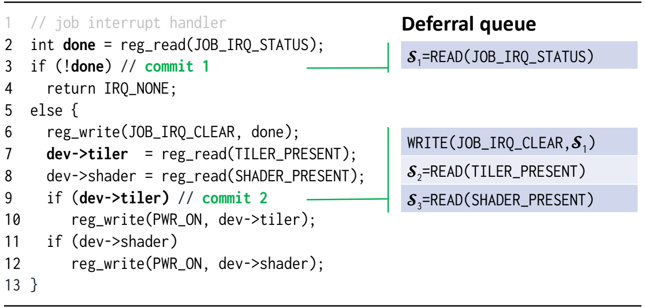
Second, DriverShim tracks data dependencies. This is because (1) the driver code may consume values from uncommitted register reads; (2) the value of a later register write may depend on the earlier register reads. Listing 1 (a) shows examples: variable `qrk_mmu` depends on the read from register `MMU_CONFIG`; the write to `MMU_CONFIG` on line 8 depends on the register read on line 4. To this end, for each queued register read, DriverShim creates a symbol for the read value and propagates the symbol in subsequent driver execution. Specifically, a symbol $\mathcal{S}$ can be encoded in a later register write to be queued, e.g. `reg_write(MMU_CONFIG,` $\mathcal{S}$`|0x10)`. After the next commit returns concrete register values, DriverShim *resolves* the symbols and replaces symbolic expressions in the driver state that encode these symbols.

Third, DriverShim respects control dependencies. The driver control flow may reach a predicate that depends on an



```
1  #define MMU_ALLOW_SNOOP_DISPARITY 0x10
2  // detect hardware quirks
3  qrk_shader = reg_read(SHADER_CONFIG);
4  qrk_mmu    = reg_read(MMU_CONFIG);
5  // configure GPU MMU accordingly
6  if (dev->coherency == COHERENCY_ACE)
7     qrk_mmu |= MMU_ALLOW_SNOOP_DISPARITY;
8  reg_write(MMU_CONFIG, qrk_mmu);
9  ...
10 // commit
```

(a) Data dependency



```
1  // job interrupt handler
2  int done = reg_read(JOB_IRQ_STATUS);
3  if (!done) // commit 1
4    return IRQ_NONE;
5  else {
6    reg_write(JOB_IRQ_CLEAR, done);
7    dev->tiler  = reg_read(TILER_PRESENT);
8    dev->shader = reg_read(SHADER_PRESENT);
9    if (dev->tiler) // commit 2
10      reg_write(PWR_ON, dev->tiler);
11   if (dev->shader)
12      reg_write(PWR_ON, dev->shader);
13 }
```

(b) Control dependency (symbolic expressions omitted)

Listing 1. Code examples of data and control dependencies. The register accesses are deferred in the queue; the driver keeps running with symbolic values until commit.

uncommitted register read, as shown in Listing 1 (b), line 3. DriverShim resolves such control dependency immediately: it commits all the queued register accesses including the one pertaining to the predicate.

When to commit? DriverShim commits register accesses when the driver triggers the following events.

• *Resolution of control dependency.* This happens when the driver execution is about to take a conditional branch that depends on an uncommitted register read.

• *Invocations of kernel APIs*, notably scheduling and locking. There are three rationales. (1) By doing so, DriverShim safely limits the scope of code instrumentation and dependency tracking to the GPU driver itself; it hence avoids doing so for the whole kernel. (2) DriverShim ensures all register reads are completed before kernel APIs that may externalize the register values, e.g. printk() of register values. (3) Committing register accesses prior to any lock operations (lock/unlock) ensures memory consistency, which will be discussed below.

• *Driver's explicit delay*, e.g. calling the kernel's delay family of functions [48]. The drivers often use delays as barriers, assuming register accesses preceding delay() in program order will take effect after delay(). For example, the driver writes a GPU register to initiate cache flush and then calls delay(), after which the driver expects that the cache flush is completed and coherent GPU data already resides in the shared memory. To respect such design assumptions, DriverShim commits register accesses before explicit delays.

Memory consistency for concurrent threads A GPU driver is multi-threaded by design. Since DriverShim defers register accesses with per-thread queues, if a driver thread assigns a symbolic value to a variable X, the actual update to X will not happen until the thread commits the corresponding register read. What if at this time another thread attempts to read X? Will it read the stale value of X?

DriverShim implements a known memory model of *release consistency* [27] to ensure no other concurrent threads can read X. The memory model is guaranteed by two designs. (1) Given that the Linux kernel and drivers have been thoroughly scrutinized for data race [49], a thread always updates shared variables (e.g. X) with necessary locks, which prevent concurrent accesses to the variables. (2) DriverShim always commits register accesses before the driver invokes unlock APIs, i.e. a thread commits register accesses before releasing any locks. As such, the thread must have updated the shared variables with concrete values *before* any other threads are allowed to access the variables.

Optimizations To further lower overhead, we narrow down the scope of register access deferral. We exploit an observation: GPU register accesses show high locality in the driver code: tens of "hot" driver functions issue more than 90% register accesses. These hot functions are analogous to compute kernels in HPC applications.

To do so, we obtain the list of hot functions via profiling offline. We run the GPU stack, trace register accesses, and bin them by driver functions. At record time, DriverShim only defers register accesses within these functions. When the driver's control flow leaves these hot functions, DriverShim commits queued register accesses. Note that (1) the choices of hot functions are for optimization and do not affect driver correctness, as register accesses outside of hot functions are executed synchronously; (2) profiling is done *once* per GPU driver, hence incurring low effort.

4.2 Speculation

Basic idea Even with deferred register accesses, each commit is still synchronous taking one RTT (Figure 5(b)). DriverShim further makes *some* commits asynchronous to hide their RTTs. The idea is shown in Figure 5(c): rather than waiting for a commit C to complete, DriverShim predicts the values of all register reads enclosed in C and continues driver execution with the predicated values; later, when C completes with the actual read values, DriverShim validates the predicated values: it continues the driver execution if the all predictions were correct; otherwise, it initiates a recovery process. Misprediction incurs performance penalty but does not violate correctness.

Why are register values predictable? Our observation is that the driver issues *recurring segments* of register accesses, to which the GPU responds with identical values most of the time. Such segments recur within a workload (e.g. MNIST inference) and across workloads (e.g. MNIST and AlexNet inferences).

What causes recurring segments? (1) Routine GPU maintenance. For instance, before and after each GPU job, the driver flushes GPU's TLB/cache. The sequences of register accesses and register values (e.g. the final status of flush operations) repeat themselves. (2) Repeated GPU state transitions. For instance, each time an idle GPU wakes up, the driver exercises the GPU's power state machine, for which the driver issues a fixed sequence of register writes (to initiate state changes) and reads (to confirm state changes). (3) Repeated hardware discovery. For instance, during its initialization, the driver probes GPU hardware capabilities by reading tens of registers. The register values remain the same as the hardware does not change.

When to speculate? Not all register accesses belong to recurring segments. To minimize misprediction, DriverShim acts *conservatively*, only making predictions when the history of commits shows high confidence.

When DriverShim is about to make a commit C, it looks up the commit history at the same driver source location. It considers the most recent k historical commits that enclose the same register access sequence as C: if all the k historical commits have read identical sequences of register values, DriverShim uses the values for prediction; otherwise, DriverShim avoids speculation for C, executing it synchronously instead. k is a configurable parameter controlling confidence that permits prediction. We set $k = 3$ in our experiment.

How does driver execute with predicted values? Based on predicted register values, the GPU driver may mutate its state and take code branches; DriverShim may make a new commit without waiting for outstanding commits to complete. To ensure correctness, DriverShim stalls the driver execution until all outstanding commits are completed and the predictions are validated, when the driver is about to externalize *any* kernel state, e.g. calling printk() on a variable. This condition is simple, as it does not differentiate if the externalized state depends on predicted register values. As a result, checking the condition is trivial: DriverShim just intercepts a dozen of kernel APIs that may externalize kernel state. DriverShim eschews fine-grained tracking of data and control dependencies throughout the whole kernel.

Optimization: Only checking the above condition has a drawback: in the event of misprediction, both the driver and the GPU have to roll back to valid states, because both may have executed based on mispredicted register values. Listing 1 (b) shows an example: if the read of `JOB_IRQ_STATUS` (line 2) is found to be mispredicted after the second commit (line 10), the driver already contains an incorrect state (in `dev`) and the GPU has executed incorrect register accesses (e.g. write to `JOB_IRQ_CLEAR`).

To this end, DriverShim can relieve the client GPU from rollback in case of misprediction. It does so by preventing spilling speculative state to the client. Specifically, DriverShim *additionally* stalls the driver before committing register accesses that themselves are speculative, i.e. having dependencies on predicted values. For example, in Listing 1 (b), the second commit must be stalled if the first is yet to complete, because the second commit consists of register accesses (`JOB_IRQ_CLEAR` and `TILER/SHADER_PRESENT`) that casually depend on the outcome of the first commit. To track speculative register accesses, DriverShim *taints* the predicted register values and follows their data/control dependencies in the driver execution. In the above example, when the driver takes a conditional branch based on a speculative value (line 3), DriverShim taints all updated variables and statements on that branch to be speculative, e.g. `dev->tiler`. For completeness, the taint tracking applies to any kernel code invoked by the driver.

```
u32 cmd = PGT_UPDATE;
int max = MAX_LOOP;
u32 val = reg_read(MMU_STATUS);

while (--max && (val & STATUS_ACTIVE))   } Offload in a shot
   val =  reg_read(MMU_STATUS);

if (max == 0)                            ── A predicate to
   return -1                                be predicted
else reg_write(MMU_CMD, cmd);
```

Listing 2. Code example of a polling loop.

How to recover from misprediction? When DriverShim finds an actual register value different from what was predicated, the GPU stack and/or the GPU should restore to valid states. We exploit the GPU replay technique [57] for both parties to reset and fast-forward *independently*. To initiate recovery, DriverShim sends the client the location of the mispredicted register access in the interaction log. Then both parties restart and replay the log up to the location. In this process, GPUShim feeds the recorded stimuli (e.g. register writes) to the physical GPU; DriverShim feeds the recorded GPU response (e.g. register reads and interrupts) to the GPU stack. Because both parties need no network communication, the recovery takes only a few seconds, as will be evaluated in Section 7.3.

4.3 Offloading polling loops

A GPU driver often invokes polling loops, e.g. to busy wait for register value changes as shown in Listing 2. Polling loops contribute a large fraction of register accesses; they are a major source of control dependencies.

Problem Naive execution of a polling loop incurs multiple round trips, rendering the aforementioned techniques ineffective. (1) Deferring register access does not benefit much, because each loop iteration generates control dependency and requests a synchronous commit. (2) Speculation on a polling loop is difficult: by design above, DriverShim must predict the iteration count upon which the terminating condition is met, which often depends on GPU timing (e.g. a GPU job's delay) and is nondeterministic in general.

Observations Fortunately, most polling loops are simple, meeting the following conditions.
• Register accesses in the loop are *idempotent*: the GPU state is not be affected by re-execution of the loop body.
• The iteration count has only a local impact: the count is a local variable and does not escape the function enclosing the loop. The count is evaluated with some simple predicates, e.g. (`count<MAX`).
• The addresses of kernel variables referenced in a loop are determined prior to the loop, i.e. the loop itself does not compute these addresses dynamically.
• The loop body does not invoke kernel APIs that have an external impact, e.g. locking and printk().

DriverShim uses static analysis to find all of the simple polling loops in the GPU driver. Complex polling loops that misfit the definition above are rare; DriverShim just executes them without optimizations.

Solution DriverShim executes simple polling loops as follows. (1) *Offloading*. DriverShim commits a loop in a shot to the client GPU, incurring only one RTT. To do so, DriverShim offloads a copy of the loop code as well as all variables to be referenced in the loop. GPUShim runs the loop and returns updated variables. Offloading respects release memory consistency as described in Section 4.1, because accesses to shared variables inside the loop must be protected with locks and the loop itself does not unlock. (2) *Speculation*. DriverShim further masks the RTT in offloading a loop. Rather than predicting the exact iteration count (e.g. the final value of `max` in Listing 2), DriverShim extracts and predicts the *predicate* on the iteration count, e.g. (`max?=0`), which is more predictable. When the client returns the actual iteration count, DriverShim evaluates the predicate in order to validate the prediction.

5 Memory Synchronization

Problem Mobile CPU and GPU were intended to share physical memory. As the driver (cloud) and the GPU (client) run on their own local memories, we need to synchronize a shared memory *view* between them as in Figure 6. Memory synchronization has been a central issue in distributed execution [8, 18, 27, 67]. A proven approach is relaxed memory consistency: one node pushes its local memory updates to other nodes only when the latter nodes are about to see the updates. Accordingly, prior systems choose synchronization points based on program behaviors, e.g. synchronizing thread-local memory at the function call boundary [18] or synchronizing shared memory of a data-race free program as part of lock/unlock operations [27].

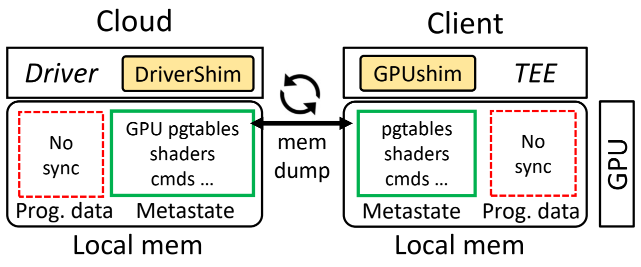


Figure 6. Selective memory synchronization of GPU metastate only but not program data.

Unlike these prior systems, the memory sharing protocol between CPU and GPU is never explicitly defined. For example, they never use locks. From our observations, we conjecture that CPU and GPU write to disjoint memory regions and order their memory accesses by *some* register accesses and *some* driver-injected delays. However, it would be brittle to build GR-T based on such vague assumptions.

Approach Our idea is to constrain the GPU driver behaviors so that we can make *conservative* assumptions for memory synchronization. To do so, we configure the driver's job queue length to be 1, which effectively serializes the driver's job preparation and the GPU's job execution. Such a constraint has been applied in prior work and shows minor overhead [57]. With the constraint, the driver emits GPU jobs to the shared memory only when the GPU is idle; the GPU is executing jobs from the memory only when the driver is idle. Therefore, *the driver and the client GPU will never access the shared memory simultaneously.*

When to synchronize? The cloud and client synchronize when GPU is about to become busy or idle:
• *Cloud ⇒ client.* Right before the register write that starts a new GPU job, DriverShim dumps its local memory allocated to GPU and sends it to the client. The memory dump is consistent: at this moment, the GPU driver has emitted and flushed all the memory states needed for the new job, and has updated the GPU page tables for mapping the memory.
• *Client ⇒ cloud.* Right after the client GPU raises an interrupt signaling job completion, GPUShim forwards the interrupt and uploads its memory dump to the cloud. The memory dump is also consistent: at this moment the GPU must have written back the job status and flushed job data from cache to local memory.

We further implement continuous validation as a safety net. After DriverShim sends its memory dump to the client, it unmaps the dumped memory regions from CPU and disables DMA to/from the memory. As such, any spurious access to the memory region will be trapped to DriverShim as a page fault and reported as an error. In the same fashion, GPUShim unmaps the shared memory from the *GPU*'s page table when the GPU becomes idle; any spurious access from GPU will be trapped.

What to synchronize? As shown in Figure 6, we minimize the amount of memory transfer with the following insight: *for recording, it is sufficient to synchronize only the GPU metastate in memory*, including GPU commands, shader code, and page tables. Synchronizing program data, such as input/output and intermediate GPU buffers, is unnecessary. Fortunately, program data constitutes most of GPU memory.

How to locate metastate in the shared memory, given that GPU memory layout is proprietary? We implement a combination of techniques. (1) Some GPU page tables have permission bits which suggest the usage of memory pages. For instance, the Mali GPUs map metastate as *executable* because the state contains GPU shader code [9]. (2) For GPU hardware lacking permission bits, GR-T infers the usage of memory regions from IOCTL() flags used by ML workloads to map these regions. For instance, a region mapped as read-only cannot hold GPU commands, because the GPU runtime needs the write permission to emit GPU commands. (3) If the above knowledge is unavailable, DriverShim simply fills an ML workload's inputs and parameters as zeros. Doing so will sparsify the GPU's program data, making memory dumps highly compressible.

We further apply standard compression. Both shims use range encoding to compress memory dumps; each shim calculates and transfers the deltas of memory dumps between consecutive synchronization points.

6 Implementations

Platforms We implement the GR-T prototype on the following platforms. The cloud service runs on a single board computer (SBC) with quad Arm Cortex-A55 cores. The client runs on Hikey960, a mobile development board with a Mali G71 MP8 GPU. Our choice of Arm processors for the cloud is for prototyping ease rather than a hard requirement; the cloud service can run on x86 machines with binary translation [67].

The cloud service runs Debian 9.4 (Linux v4.14) with a GPU stack composed of an ML framework (ACL v20.05 [11]), a runtime (`libmali.so`), and a driver (Mali Bifrost r24 [12]). Below the service, KVM-QEMU (v4.2.1) runs as the VM hypervisor. The client runs Debian 9.13 (Linux v4.19) and OPTEE (v3.12) as its TEE.

DriverShim We build our code instrumentation tool as a Clang plugin. For static analysis and code manipulation, the plugin traverses the driver's abstract syntax tree (AST). With the Clang/LLVM toolchain [19], our tool compiles the GPU driver and links it against DriverShim. By limiting the scope to the hot driver functions in the Mali GPU driver (§4.1), our instrumentation tool processes 19 functions in total. The instrumentation itself incurs negligible overhead. We implement DriverShim as a kernel module (~1K SLoC) to be invoked by the instrumented driver code; the module performs dependency tracking, commit management, and speculation, as described in Section 4 and 5.

DriverShim communicates with the client via TCP-based messages. We install GPU devicetrees in the cloud VM, so the GPU stack can run transparently even a physical GPU is not present [67]. To support multiple GPU types, we implement a mechanism for the cloud service to load per-GPU devicetree when a VM boots. As a result, a single VM image can incorporate multiple GPU drivers, which are dynamically loaded depending on the specific client GPU model.

GPUShim We build GPUShim as a TEE module. Following the TrustZone convention, GPUShim communicates with the cloud using the GlobalPlatform APIs implemented by OPTEE [26]. The communication is authenticated and encrypted by SSL 3.0 with the TEE, before it is forwarded through the normal-world OS.

By design, the trusted firmware on the client dynamically switches the GPU between the normal world and the TEE with a configurable TrustZone address space controller (TZASC) [44]. Yet, our client platform (Hikey960) has a proprietary TZASC which lacks public documentation [33]. We workaround this issue by statically reserving memory regions for GPU and mapping the memory regions and GPU registers to the TEE.

We modify the secure monitor to route the GPU's interrupts to the TEE. GPUShim forwards the interrupts to DriverShim for handling. We avoid interrupt injection to the VM hypervisor and keep it unmodified.

To bootstrap the GPU, the client TEE needs to access SoC resources not managed by the GPU driver, e.g. power/clock for GPU. For strong security, we protect these resources inside the TEE as did in prior work [44] instead of invoking the normal-world OS via RPC [67].

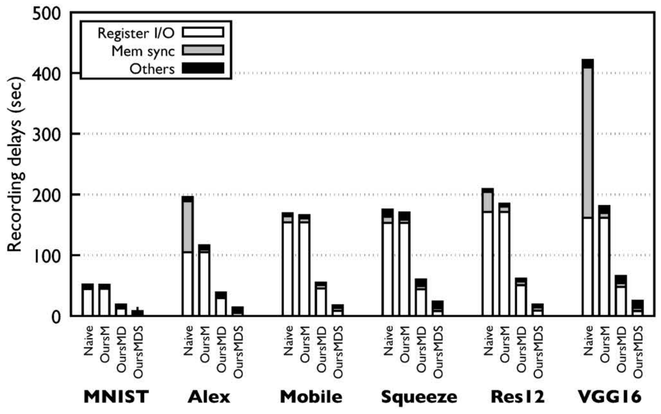
7 Evaluation

The evaluation answers the following questions.

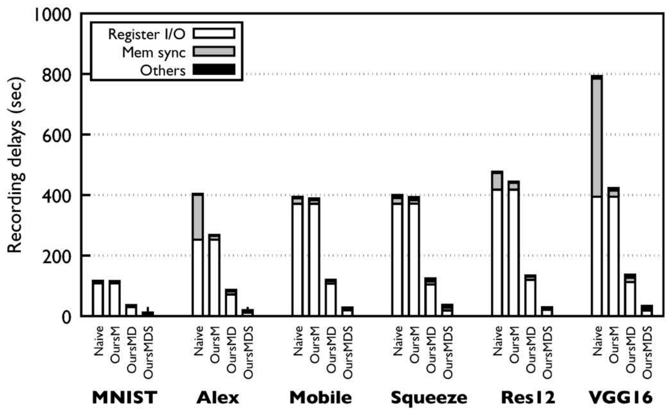
- Is GR-T secure against attacks? (§ 7.1)
- What are the delays of GR-T? (§ 7.2)
- Are GR-T's optimizations significant? (§ 7.3)
- What is the energy implication of GR-T? (§7.4)

7.1 Security analysis

Threat model We trust the cloud service and its GPU stack, assuming that the VMs are attested [65, 66] when the client TEE requests a connection to them. We also assume that each client TEE can communicate with the cloud VMs over a secure channel where all the data is encrypted (e.g. using attested TLS [39]). We trust the client's TrustZone and hardware but not its OS. We consider two types of adversaries: (1) a local, privileged adversary who controls the client OS; (2) a network-level adversary who can eavesdrop the cloud/client communications during recording.



(a) Recording with WiFi conditions (RTT: 20ms, BW: 80Mbps)



(b) Recording with cellular conditions (RTT: 50ms, BW: 40Mbps)

Figure 7. Recording delays of our design (`OursMDS`) are significantly lower than other versions.

NNs (# GPU jobs)	# Blocking RTTs			MemSync (MB)	
	OursM	OursMD	OursMDS	Naive	OursM
MNIST (23)	2837	585	65	3.07	0.8
AlexNet (60)	5008	1392	196	454.9	4.2
MobileNet (104)	7307	2097	320	37.4	11.8
SqueezeNet (98)	7373	2049	303	41.3	11.3
ResNet12 (111)	8326	2352	345	151.2	13.0
VGG16 (96)	7662	2184	309	1215.2	10.2

Table 1. Statistics of record runs, showing GR-T significantly reduces network round trips that block the recording and the memory synchronization traffic.

Security overhead and TCB Compared to the total recording delay, the secure communication in GR-T incurs negligible overhead; the cloud VM attestation is cheap as done locally on the cloud. Establishing a secure channel leads to a couple of additional RTTs; the entailed data encryption overhead is low as the payload for each commit is small (200 – 400 Bytes in our measurement).

GR-T's TCB includes the cloud VM (including the GPU stack) and the client's TEE. The TCB is better shielded and exposes a smaller attack surface, as compared to the TCB for client GPU computation outside the TEE, which includes the client's entire OS.

Integrity GR-T's *recording integrity* is jointly ensured by (1) the trusted cloud service, (2) the client's TrustZone hardware, and (3) the encrypted cloud/client communication. In particular, GPUShim locks the GPU MMIO region during recording, preventing any local adversary from tampering with GPU registers or shared memory. GR-T's *replay integrity* is ensured by the TrustZone hardware. Since the replayer only accepts recordings signed by the cloud, it exposes no additional attack surface to adversaries.

Confidentiality For *recording*, TEE does not leak ML data, e.g. model parameters or inputs, as such data never leaves the TEE. This is due to input independence (§2.3): to record

	Delay (ms)					
	MNIST	Alex	Mobile	Squeeze	Res12	VGG16
Native	15.2	63	60.9	64.3	362.1	372.2
OursMDS	4.8	54.8	45.2	54.3	373.9	364.8

Table 2. Replay delays of GR-T (`OursMDS`) are similar to Native, which executes benchmarks on the GPU stack in the normal world of the same device.

computation, the cloud does not need the actual input or parameters. The TEE, however, must reveal ML model structures and GPU commands/shaders to the cloud. Although the network traffic is encrypted, sophisticated eavesdroppers may learn certain model information via network side channels. Such side channels can be mitigated by orthogonal solutions [34, 77].

Since *replay* is within the client TEE and requires no client/cloud communication, its data confidentiality is given by TrustZone. We notice TrustZone may leak data to local adversaries via hardware side channels, which can be mitigated by existing solutions [50, 76].

Availability Like any cloud-based service, *recording* availability of GR-T depends on network conditions and the cloud availability, which are vulnerable to DDoS attacks. Its *replay* availability is at the same level of the TrustZone TEE, when the GPU power is managed by the TEE not the OS [44].

7.2 Performance

Methodology As shown in Table 1, we test GR-T on inference with 6 popular NNs running atop ARM Compute Library [11]. We measure GR-T's recording delay under two network conditions as controlled by NetEm [31]: i) WiFi-like (20 ms RTT, 80 Mbps) and ii) cellular-like (50 ms RTT, 40 Mbps) [59]. The hardware platform is described in Section 6.

We compare the following recorder implementations:

- `Naive` incurs a round trip per register access and synchronizes entire GPU memory before/after a GPU job.

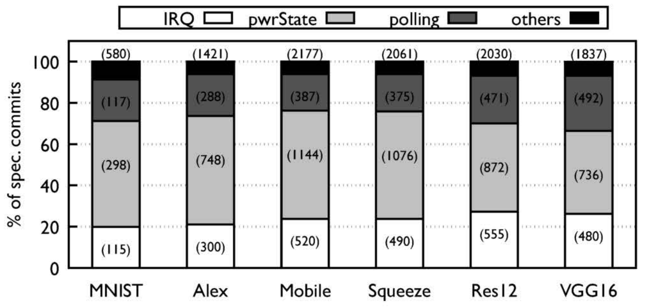


Figure 8. Breakdown of speculative commits, normalized to 100%. The actual numbers of commits are shown in parentheses.

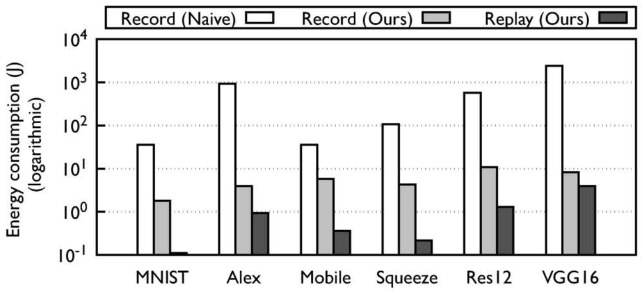


Figure 9. System energy for record and replay.

- OursM includes metaonly memory synchronization (§5).
- OursMD, in addition to OursM, includes register access deferral (§4.1); it generates an RTT per commit.
- OursMDS additionally includes speculation (§4.2). It represents GR-T with all our techniques.

Recording delays Figure 7 shows the end-to-end recording delays. Naive incurs long recording delays: on WiFi, the delays range from 52 seconds (MNIST, a small NN) to 423 seconds (VGG16, a larger NN); on cellular network, the delays range from 116 seconds to 795 seconds. As discussed in Section 3.3, such high delays render the system unusable. Compared to Naive, OursMDS reduces the delays by up to 95%, to 18 seconds (WiFi) and 30 seconds (cellular) on average. With the optimized delays, the GPU software/hardware never throw exceptions; the delays are acceptable to users because they are comparable to mobile app installation delays reported to be 10 – 50 seconds [37].

Replay delays GR-T's replay is faster in most of the benchmarks as shown in Table 2. Compares native executions, GR-T's replay delays range from 68% lower to 3% higher (25% lower on average). GR-T performance advantage comes from its removal of the complex GPU stack. We notice that these results are consistent with the prior work [57].

7.3 Validation of key designs

Efficacy of deferral Figure 7 (OursM vs. OursMD) shows the impact of register access deferral: it reduces the overall delays by 65% (WiFi) and 69% (cellular). Table 1 further shows that the deferral reduces the number of round trips by 73% on average. With deferral, each commit encloses 3.8 register accesses on average.

Efficacy of speculation We run all six benchmarks with retaining register access history in between, allowing GR-T to reuse history across benchmarks. Figure 7 (OursMDS vs. OursMD) shows that speculation reduces the recording delays by 60% to 74%. Table 1 further shows OursMDS achieves 86% reduced number of round trips on average. Such benefits mainly come from coalescing round trips via asynchronous commits.

We further investigate the speculation success rates and find 95% of commits (99% register accesses) satisfy the speculation criteria (§4.2). These commits are generated by GPU driver routines that fall into four categories. (1) *Init*: probe hardware configuration when loading the driver. (2) *Interrupt*: read and clear interrupt status. (3) *Power state*: periodic manipulation of GPU power states. (4) *Polling*: busy wait for GPU to finish TLB or cache operations. Figure 8 shows a breakdown of commits by category. All register values in these commits are highly predictable.

The commits that fail the criteria are due to reads of nondeterministic register values. For example, on each job submission, the Mali GPU driver reads and writes a register LATEST_FLUSH_ID which reflects the GPU cache state and can be nondeterministic.

Misprediction cost We have not observed misprediction in our 1,000 runs of each workload. To validate that GR-T can handle misprediction, we artificially inject into record runs wrong register values. In all the cases of injection, GR-T always detects mismatches between the predicted and the injected register value, initiating rollback of the software and the hardware states properly. In the worst case (misprediction at the end of a record run), we measure the delays of rollback as 1 and 3 seconds for MNIST and VGG16, respectively. The delays are primarily dominated by driver reload and GPU job recompilation on the cloud side, which exceed the replay delays on the client GPU.

Selective memory synchronization Figure 7 (OursM vs. Naive) shows that the technique reduces the recording delays by 1% – 57% on average. The reduction is more pronounced on large NNs such as AlexNet and VGG16 (34% – 57%). Table 1 shows the network traffic for memory synchronization is reduced by 72% – 99%.

Polling offloading (§4.3) The numbers of polling loop instances range from 117 (MNIST) to 492 (VGG16), which generate from 130 to 550 round trips. Offloading each polling instance reduces round trips by 13 – 58 per benchmark. This is because without offloading, a polling loop often takes a few RTTs (the RTT is long as compared to GPU operations being polled such as cache flush); with offloading and speculation, the polling loop takes one RTT.

7.4 Energy consumption

We measure the whole client energy using a digital multimeter which instruments the power barrel of the client device (Hikey960). The client device has no display. It uses the on-board WL1835 WiFi module for communication; it runs no other foreground applications. Each workload runs 500 times and we report the average per run. Figure 9 shows the results.

Record. The energy consumed by recording is moderate, ranging from 1.8J – 8.2J, which is comparable to energy for installing a mobile app, e.g. 16J for Snapchat (80MB) on the same device. Note that it is one-time consumption per ML workload. Compared to `Naive`, GR-T reduces the system energy by 84% – 99%.

Replay. As a reference, we measure replay energy per benchmark. It ranges from 0.01 – 1.3 J, consistent with the replay performance in Table 2. The replaying energy is comparable with the native GPU execution on the client device (not shown in the figure).

8 Related Work

Remote I/O is adopted for cross-device I/O sharing [8, 55] and task offloading [21, 34]. Unlike GR-T, their remoting boundary is at higher levels, e.g. file [8], Android binder IPC [55], and runtime API [34]. Doing so would bloat the TEE with implementation of these high-level APIs.

Similar to GR-T, prior works interpose low-level primitives, e.g. forwarding I/O from VM to mobile system [67] or low-level memory access from emulator to real device [40, 74]. However, their cross-device interfaces are wired as opposed to wireless Internet addressed by GR-T. Contrasting to their concrete executions, GR-T targets dry run and therefore contributes unique optimizations that were absent.

Device isolation with TEE Recent works isolates GPUs by enclosing the GPU stack in the TEE [36, 51] or even part of the GPU stack in hardware [71]. They, however, require deep modification of the GPU software/hardware and/or bloat the TEE. TrustZone is also leveraged by prior works to secure devices, e.g. peripheral IO [44] and displays [7, 45, 56]; none of them use replay as we did for a complex GPU stack. Although recent works [30, 57] secure devices based on record and replay the device interactions, they cannot solve the problem of recording (§2.4), which is limited to developers machines unlike GR-T's online recording.

Speculative execution is widely explored by prior works; based on caching and prefetching, they facilitate asynchronous file I/O [16, 54, 62] or speed up VM replication [22] and distributed systems [75]. GR-T gives this conventional wisdom a fresh context: mobile GPUs. Catering to GPU computation, GR-T avoids register prefetch; it predicts register values with heuristics specific to mobile GPU drivers, which were unexploited by prior works.

Mobile cloud offloading Cloud offloading [18, 20, 27] partitions mobile application between a device and the cloud; both partitions collaborate to execute. GR-T can be viewed as an extreme case of offloading: the whole GPU stack is offloaded while only GPU hardware remains on device. As opposed to prior offloading for concrete execution, GR-T's offloading is for dry run.

GPU record and replay at a variety of API levels is used to reverse-engineer GPU commands [6, 28, 46], enhance performance [41], profiling [14], and lean software deployment [30, 57]. While prior works focus on *what* to record, GR-T focuses on *how* to record for TEE, for which GR-T contributes remote GPU recording.

Secure client ML Many works protect the confidentiality of ML input and model parameters [29, 43, 52, 53]. Their ML computation runs on CPU instead of GPU. While recent work [69] proposes verifiable GPU compute in TEE, the entailed expensive homomorphic encryption unlikely fits client devices.

9 Conclusions

GR-T is a novel system architecture to run GPU computation inside the TrustZone TEE. It provides a safe, practical recording process. The key idea for recording is to leverage a cloud service which dry runs the GPU stack interacting with the client GPUs over wireless communication. With a series of I/O optimization techniques specific to mobile GPUs, GR-T significantly reduces the time and energy consumed by clients.

Acknowledgments

The authors were supported in part by NSF awards #2128725, #1919197, #2106893, and Virginia's Commonwealth Cyber Initiative. The authors thank the shepherd Prof. Mark Silberstein and the anonymous reviewers for their insightful feedback.

References

[1] CVE-2014-1376: Improper restriction to unspecified opencl api calls. https://nvd.nist.gov/vuln/detail/CVE-2014-1376, 2014.

[2] CVE-2019-14615: Information leakage vulnerability on the intel integrated gpu architecture. https://nvd.nist.gov/vuln/detail/CVE-2019-14615, 2019.

[3] CVE-2019-20577: Smmu page fault in mali gpu driver. https://nvd.nist.gov/vuln/detail/CVE-2019-20577, 2019.

[4] CVE-2019-5068: Exploitable shared memory permission vulnerability in mesa 3d graphics library. https://nvd.nist.gov/vuln/detail/CVE-2019-5068, 2019.

[5] CVE-2020-11179: Qualcomm adreno gpu ringbuffer corruption / protected mode bypass. https://nvd.nist.gov/vuln/detail/CVE-2020-11179, 2020.

[6] alyssa rosenzweig. Dissecting the apple m1 gpu. https://rosenzweig.io/blog/asahi-gpu-part-1.html.

[7] A. Amiri Sani. Schrodintext: Strong protection of sensitive textual content of mobile applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, page 197–210, New York, NY, USA, 2017. Association for Computing Machinery.

[8] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A system solution for sharing i/o between mobile systems. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 259–272, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Android kernel. Arm Bifrost Graphics Driver: KBASE_REG_GPU_NX. https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/mali_kbase_mem.h#208.

[10] Android kernel. Arm Bifrost Graphics Driver: kbase_show_gpuinfo(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/mali_kbase_core_linux.c#2698.

[11] Arm. Arm Compute Library. https://github.com/ARM-software/ComputeLibrary.

[12] Arm. Open Source Mali Bifrost GPU Kernel Drivers. https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/bifrost-kernel.

[13] Arm. Open Source Mali Midgard GPU Kernel Drivers . https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/midgard-kernel.

[14] ARM-software. Software for capturing gles calls of an application and replaying them on a different device. https://github.com/ARM-software/patrace.

[15] F. Brasser, D. Gens, P. Jauernig, A. Sadeghi, and E. Stapf. SANCTUARY: arming trustzone with user-space enclaves. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[16] F. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. In *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, Feb. 1999. USENIX Association.

[17] H. Chen, C. Fu, B. D. Rouhani, J. Zhao, and F. Koushanfar. Deepattest: An end-to-end attestation framework for deep neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 487–498, New York, NY, USA, 2019. Association for Computing Machinery.

[18] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[19] Clang. a C language family frontend for LLVM. https://clang.llvm.org/.

[20] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proc. USENIX/ACM MobiSys*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[21] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 121–135, New York, NY, USA, 2015. Association for Computing Machinery.

[22] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, Apr. 2008. USENIX Association.

[23] R. de Jong and A. Sandberg. Nomali: Simulating a realistic graphics driver stack using a stub gpu. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 255–262, 2016.

[24] GadgetVersus. Various lists of graphics cards. https://gadgetversus.com/graphics-card/.

[25] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 201–210, New York, New York, USA, 20–22 Jun 2016. PMLR.

[26] Global Platform. Tee internal core api specification. https://globalplatform.org/specs-library/tee-internal-core-api-specification, 2021.

[27] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. OSDI'12, page 93–106, USA, 2012. USENIX Association.

[28] Grate. Open source reverse-engineering tools aiming at nvidia tegra2+3d engine. https://github.com/grate-driver/grate.

[29] Z. Gu, H. Huang, J. Zhang, D. Su, A. Lamba, D. Pendarakis, and I. Molloy. Securing input data of deep learning inference systems via partitioned enclave execution. *CoRR*, abs/1807.00969, 2018.

[30] L. Guo and F. X. Lin. Minimum viable device drivers for arm trustzone. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 300–316, New York, NY, USA, 2022. Association for Computing Machinery.

[31] S. Hemminger. Network emulation with netem. Linux conf au, 2005.

[32] M. Hogan, F. Liu, A. Sokol, and J. Tong. Nist cloud computing standards roadmap. *NIST Special Publication*, 35:6–11, 2011.

[33] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan. vtz: Virtualizing ARM trustzone. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 541–556, Vancouver, BC, 2017. USENIX Association.

[34] T. Hunt, Z. Jia, V. Miller, A. Szekely, Y. Hu, C. J. Rossbach, and E. Witchel. Telekine: Secure computing with cloud gpus. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 817–833, Santa Clara, CA, Feb. 2020. USENIX Association.

[35] James Reed and Michael Suo. Introduction to Torchscript. https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html, 2021.

[36] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–468. ACM, 2019.

[37] C. J. Jiang, S. Li, G. Huo, and L. Luo. Research on the relationship between app size and installation time in intelligent mobile devices. In *2019 IEEE Fourth International Conference on Data Science in Cyberspace (DSC)*, pages 270–277, 2019.

[38] Kashish Kumawat, Tech Centurion. Mobile GPU Rankings 2021 (Adreno/Mali/PowerVR). https://www.techcenturion.com/mobile-gpu-rankings.

[39] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. Integrating remote attestation with transport layer security. *CoRR*, abs/1801.05863, 2018.

[40] K. Koscher, T. Kohno, and D. Molnar. SURROGATES: Enabling near-real-time dynamic analyses of embedded systems. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., Aug. 2015. USENIX Association.

[41] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 8343–8354. Curran Associates, Inc., 2020.

[42] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann. On-device neural net inference with mobile gpus. *CoRR*, abs/1907.01989, 2019.

[43] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, F. Xu, C. Xu, L. Zhang, and J. Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[44] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '18, page 1–13, New York, NY, USA, 2018. Association for Computing Machinery.

[45] W. Li, M. Ma, J. Han, Y. Xia, B. Zang, C.-K. Chu, and T. Li. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[46] T. M. D. G. Library. Panfrost. https://docs.mesa3d.org/drivers/panfrost.html.

[47] Linux. Qualcomm adreno graphics driver: gpu_list. https://elixir.bootlin.com/linux/v5.15-rc5/source/drivers/gpu/drm/msm/adreno/adreno_device.c#L23.

[48] Linux. delays - Information on the various kernel delay / sleep mechanisms. https://www.kernel.org/doc/Documentation/timers/timers-howto.txt/, 2021.

[49] Linux. The Kernel Concurrency Sanitizer (KCSAN). https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html, 2021.

[50] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 549–564, Austin, TX, Aug. 2016. USENIX Association.

[51] R. Liu, L. Garcia, Z. Liu, B. Ou, and M. Srivastava. Secdeep: Secure and performant on-device deep learning inference framework for mobile and iot devices. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, IoTDI '21, page 67–79, New York, NY, USA, 2021. Association for Computing Machinery.

[52] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis. Ppfl: Privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 94–108, New York, NY, USA, 2021. Association for Computing Machinery.

[53] F. Mo, A. S. Shamsabadi, K. Katevas, S. Demetriou, I. Leontiadis, A. Cavallaro, and H. Haddadi. Darknetz: Towards model privacy at the edge using trusted execution environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, MobiSys '20, page 161–174, New York, NY, USA, 2020. Association for Computing Machinery.

[54] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 191–205, New York, NY, USA, 2005. Association for Computing Machinery.

[55] S. Oh, H. Yoo, D. R. Jeong, D. H. Bui, and I. Shin. Mobile plus: Multi-device mobile platform for cross-device functionality sharing. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, page 332–344, New York, NY, USA, 2017. Association for Computing Machinery.

[56] C. M. Park, D. Kim, D. V. Sidhwani, A. Fuchs, A. Paul, S.-J. Lee, K. Dantu, and S. Y. Ko. Rushmore: Securely displaying static and animated images using trustzone. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 122–135, New York, NY, USA, 2021. Association for Computing Machinery.

[57] H. Park and F. X. Lin. Gpureplay: A 50-kb gpu stack for client ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2022, page 157–170, New York, NY, USA, 2022. Association for Computing Machinery.

[58] H. Park, S. Zhai, L. Lu, and F. X. Lin. Streambox-tz: Secure stream analytics at the edge with trustzone. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 537–554, Renton, WA, July 2019. USENIX Association.

[59] S. Park, J. Lee, J. Kim, J. Lee, S. Ha, and K. Lee. Exll: An extremely low-latency congestion control for mobile cellular networks. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 307–319, New York, NY, USA, 2018. Association for Computing Machinery.

[60] R. D. Pietro, F. Lombardi, and A. Villani. Cuda leaks: A detailed hack for cuda and a (partial) fix. 15(1), Jan. 2016.

[61] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1039–1056. USENIX Association, Aug. 2020.

[62] A. Raman, G. Yorsh, M. Vechev, and E. Yahav. Sprint: Speculative prefetching of remote data. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 259–274, New York, NY, USA, 2011. Association for Computing Machinery.

[63] J. W. M. W. Y. W. J. Z. Rendong Liang, Ting Cao and Y. Liu. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *The 28th Annual International Conference on Mobile Computing and Networking*, MobiCom '22, 2022.

[64] A. A. Sani and T. Anderson. The case for i/o-device-as-a-service. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 66–72, New York, NY, USA, 2019. Association for Computing Machinery.

[65] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, USA, 2009. USENIX Association.

[66] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, Bellevue, WA, Aug. 2012. USENIX Association.

[67] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 291–307, Baltimore, MD, Aug. 2018. USENIX Association.

[68] Tencent. Tencent ncnn framework. https://github.com/Tencent/ncnn.

[69] F. Tramer and D. Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *International Conference on Learning Representations*, 2019.

[70] S. B. E. V. S. M. L. N. G. A. D. E. Y. Vinh Nguyen, Michael Carilli. Accelerating PyTorch with CUDA Graphs. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/.

[71] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, Carlsbad, CA, Oct. 2018. USENIX Association.

[72] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu. Asymo: scalable and efficient deep-learning inference on asymmetric mobile cpus. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 215–228, 2021.

[73] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, and P. Zhang. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019.

[74] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.

[75] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 29–42, USA, 2008. USENIX Association.

[76] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. Truspy: Cache side-channel information leakage from the secure world on arm devices. *IACR Cryptol. ePrint Arch.*, 2016:980, 2016.

[77] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association.