# GPUReplay: A 50-KB GPU Stack for Client ML

Heejin Park
Purdue University
West Lafayette, Indiana, USA
bakhi@purdue.edu

Felix Xiaozhu Lin
University of Virginia
Charlottesville, Virginia, USA
felixlin@virginia.edu

## ABSTRACT

GPUReplay (GR) is a novel way for deploying GPU-accelerated computation on mobile and embedded devices. It addresses high complexity of a modern GPU stack for deployment ease and security. The idea is to record GPU executions on the full GPU stack ahead of time and replay the executions on new input at run time. We address key challenges towards making GR feasible, sound, and practical to use. The resultant replayer is a drop-in replacement of the original GPU stack. It is tiny (50 KB of executable), robust (replaying long executions without divergence), portable (running in a commodity OS, in TEE, and baremetal), and quick to launch (speeding up startup by up to two orders of magnitude). We show that GPUReplay works with a variety of integrated GPU hardware, GPU APIs, ML frameworks, and 33 neural network (NN) implementations for inference or training. The code is available at `https://github.com/bakhi/GPUReplay`.

## CCS CONCEPTS

• **Security and privacy → Systems security**; *Operating systems security*; *Mobile platform security*.

## KEYWORDS

GPU stack; secure GPU computation; record and replay; client ML

**ACM Reference Format:**
Heejin Park and Felix Xiaozhu Lin. 2022. 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 '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507754

## 1 INTRODUCTION

**GPU stacks** Smartphones or IoT devices commonly use GPUs to accelerate machine learning (ML). As shown in Figure 1(a), a modern GPU software stack spans ML frameworks (e.g. Tensorflow [11] and ncnn [97]), a GPU runtime (e.g. OpenCL or Vulkan runtimes) that translates APIs to GPU commands and code, and a GPU driver that tunnels the resultant code and data to GPU. A GPU stack[1] has a large codebase. Arm Mali, reported to be the most pervasive GPUs in the world [24], has a runtime of a 48-MB executable; the driver has 45K SLoC [25]. The stack often has substantial proprietary code and undocumented interfaces.

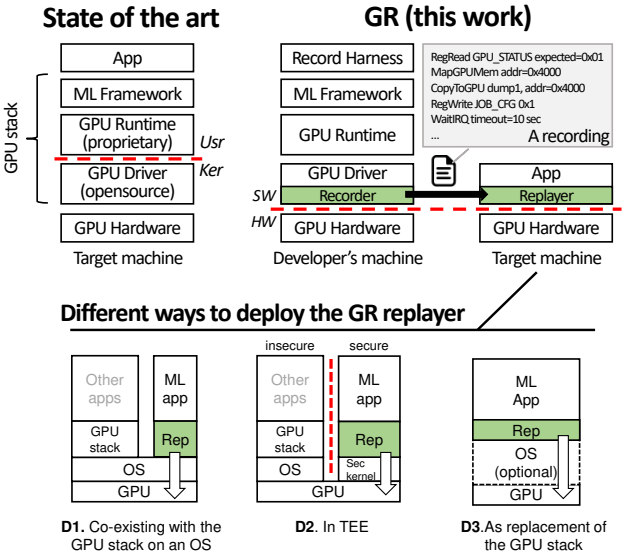


**Figure 1: The overview of GR**

Such a sophisticated GPU stack has created a number of difficulties. (1) Weak security [53, 90, 107]. In the year of 2020, 46 CVEs on GPU stacks were reported, most of which are attributed to the stack's complex internals and interfaces. (2) Deployment difficulty [101]. For instance, ncnn, a popular mobile ML framework, requires the Vulkan API. Yet the Vulkan runtime for Arm GPUs only exists on Android but not GNU/Linux or Windows [26]. Even on a supported OS, an ML app often only works with specific combinations of runtime/kernel versions [39, 49, 76]. (3) Slow startup. Even a simple GPU job may take several seconds to launch because of expensive stack initialization. This paper will show more details.

The complexity of a GPU stack was mostly for its original design goal: to support *interactive* apps with numerous dynamic GPU jobs. Such a goal is less important to ML apps, which often run a *prescribed* set of GPU jobs (albeit on new input data) [105]; many ML apps run GPU job batches without user interactions; they can multiplex on GPU at long intervals, e.g. seconds. The ML apps just need to quickly shove computation into GPU. They should not be burdened by a full-blown GPU stack.

**Our approach** GPUReplay (GR) is a new way to deploy and execute GPU compute with little changes to the existing GPU stack. We focus on integrated GPUs on system-on-chips (SoCs). Figure 1(b) overviews its workflow. At development time, developers run their ML app and record GPU executions. The recording is feasible: despite much of the GPU stack is a blackbox, it interacts with the GPU

[1]We stress that the GPU stack is software code running on *CPU*, not *GPU*


*ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland*

ACM ISBN 978-1-4503-9205-1/22/02.
https://doi.org/10.1145/3503222.3507754

at a narrow interface – registers and memory, which is managed by an open-source driver. Through lightweight instrumentation, an in-driver recorder can trace CPU/GPU interactions as a series of register accesses and memory dumps which enclose proprietary GPU commands and instructions. They are sufficient for reproducing the GPU computation.

To replay, an ML app invokes the recorded GPU executions on new input data. To the app, the GPU stack is substituted by a replayer, which is much simpler as it avoids GPU API translation, code generation, and resource management. It simply accesses GPU registers and loads memory dumps at specified time intervals. Throughout the process, the recorder/replayer remain oblivious to the semantics of most register accesses and memory dumps.

**Use cases** Figure 1 shows deployment scenarios of the replayer.

D1. *Co-existing with a GPU stack on the same OS.* This applies to smartphones. Common interactive apps without GR run on the GPU stack. When they are not using GPU, the OS runs GR-supported ML with replay. Once the interactive apps ask for GPU, the OS preempts GPU from the ongoing replay with short delays (Section 5).

D2. *In TEE.* This applies to Arm TrustZone [88]. On the same machine, apps not using GR run on the GPU stack in the normal world and GR-supported ML runs atop a replayer in the secure world. A secure monitor at EL3 switches GPU between the two worlds.

D3. *As a replacement for the system's GPU stack.* This applies to headless devices such as robots, where GR-supported ML apps share GPU cooperatively. Each ML app runs its own replayer instance.

**Benefits** GR offers the following benefits:

(1) *Security* First, GR better shields the GPU stack. The GPU stack serving the target ML app is detached from the app and instead resides on the developer's machine for recording only. Hence, the stack is no longer exposed to many threats in the wild but instead protected as part of software supplychain, for which attacks require high capabilities and long commitment [31]. Second, on target machines, the replayer replaces the GPU stack for the app (D1/D2) or for the whole system (D3). As a result, either the app or the whole system is free from vulnerabilities from the GPU stack, which originate in rich features such as buffer management [2, 8] and fine-grained sharing [4, 7, 10], as well as complex interfaces such as framework APIs [3], IOCTLs [6], and directly mapped memory [9]. By comparison, the replayer only has a few K SLoC and exposes several simple functions; replay actions have simple, well-defined semantics and are amenable to checks.

(2) *Ease of ML deployment* The replayer can run in various environments: at user or kernel level of a commodity OS, in a TEE, in a library OS, and even baremetal. Section 6 will present the details. GR brings mature GPU compute such as Tensorflow NNs to these environments without porting full GPU stacks. GR is compatible with today's GPU ecosystems. It requires no reverse engineering of proprietary GPU runtimes, commands, and shaders. Agnostic to GPU APIs, GR can record and replay diverse ML workloads.

(3) *Faster GPU invocation* GR reduces the GPU stack initialization to baremetal: register accesses and GPU memory copy. It removes expensive abstractions of multiple software layers, dynamic CPU/GPU memory management, and just-in-time generation of GPU commands and code.

**Challenges** First, we make reproduction of GPU workloads feasible despite the GPU's complex interfaces and proprietary internals. We identify and capture key CPU/GPU interactions and memory states; we selectively dump memory regions and discover the input/output addresses operated by GPU commands/shaders.

Second, we ensure GR's replay is correct in the face of nondeterministic CPU/GPU interactions. A key insight is that replay correctness is equivalent to the GPU finishing the same sequence of state transitions as recorded. To this end, we *prevent* many state divergences by eliminating their sources at the record time; we *tolerate* non-deterministic interactions that do not affect the GPU state at the replay time. GR's approach to nondeterminism sets it apart from prior record-and-replay systems [29, 48, 106]: targeting program debugging, they seek to reproduce the original executions with high fidelity and preserve all nondeterministic events in replay.

Third, we investigate a variety of practicality issues. We identify the minimum GPU hardware requirements. We show that GR requires low developer efforts, and such efforts are often amortized over a family of GPUs supported by one driver. We explore GR's deployment ranging from smartphones to headless IoT devices. We investigate how to map an ML workload to GR recordings and quantify the impact of recording granularities. We propose a scheduling mechanism for the replayer to share GPU with interactive apps.

**Results** GR works on a variety of GPUs (Arm Mali and Broadcom v3d), APIs (OpenCL, GLES compute, and Vulkan), ML frameworks (ACL [22], ncnn [97], Tensorflow [11], and DeepCL [89]), and 33 NN implementations. We build replayers for userspace, kernel, TrustZone, and a baremetal environment. We show that a recording with light patching can be replayed on different GPU hardware of the same family. Compared to the original GPU stack, the replayer's startup delays are lower by up to two orders of magnitude; its execution delays range from 68% lower to 15% higher.

This paper makes the following contributions:

(1) GPUReplay (GR), a new way to deploy GPU computation.

(2) A recorder that captures the essential GPU memory states and interactions for replay.

(3) A safe, robust replayer that verifies recordings for security, supports GPU handoff and preemption, and detects and recovers from replay failures.

(4) Realization of the design in diverse software/hardware environments.

## 2 MOTIVATIONS

### 2.1 The GPU Stack and Its Problems

**CPU/GPU interactions** As shown in Figure 2, CPUs request computation on GPUs by sending jobs to the latter. The GPU runtime directly emits GPU job binaries – GPU commands, metadata, and shaders – to GPU-visible memory[2]. The runtime communicates with the driver with ioctl syscalls, e.g. to allocate GPU memory or to start a job.

**Why are GPU stacks complex?** Several key features of a GPU stack cater to graphics.

[2]GPU memory for short, with the understanding it is part of shared DRAM

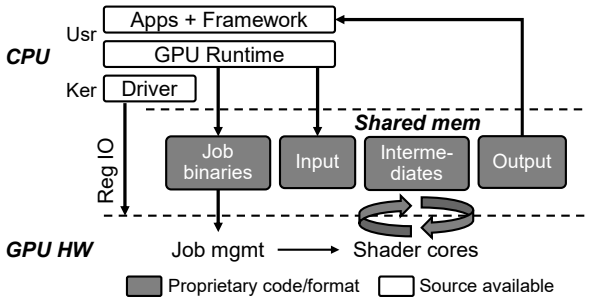


**Figure 2: The software/hardware for an integrated GPU**

(1) *Just-in-time (JIT) job generation.* Graphics apps emit numerous GPU jobs, from uploading textures to rendering fragments. For instance, during a game demo of 50 seconds [92], the v3d GPU executes 32K jobs. A game may rewrite shader sources for jobs [44]. Unable to foresee these jobs, the GPU stack generates their commands and shaders just in time.

(2) *Dynamic resource management.* Depending on user interactions, graphics apps generate GPU jobs with various input sizes, data formats, and buffer lengths. They require dynamic management of GPU time and memory, which may further entail sophisticated CPU/GPU coordination [13].

(3) *Fine-grained multiplexing.* Concurrent programs may draw on their screen regions. To support them, the GPU stack interleaves jobs at fine intervals and maintains separation.

**Compute for ML** shows disparate nature unlike graphics.

*Prescribed GPU jobs*: One app often runs pre-defined ML algorithms [105], requesting a smaller set of GPU jobs repeatedly executed on different inputs. Popular neural networks (NN) often have tens of GPU jobs each (§7). The needed GPU memory and time can be statically determined.

*Coarse-grained multiplexing*: On embedded devices, ML may run on GPU for long without sharing (e.g. object detection on a smart camera). On multiprogrammed smartphones, ML apps may run in background, e.g. model fine-tuning. Such an app tolerates delays of hundreds of milliseconds or seconds in waiting for a GPU; once on GPU, it can generate adequate workloads to utilize the GPU.

**Runtime blackboxes** Most GPUs have proprietary runtime, job binaries, and shaders. While GR can be more efficient had it known these internals or changed them, doing so requires deep reverse engineering and makes deployment harder. Hence, we avoid changing these blackboxes but only tap in the Linux GPU drivers which are required to be open-source.

**Design Implication** A GPU stack's dual modality for graphics and compute becomes a burden. While an ML app still needs the GPU stack for translating higher-level programming abstractions to GPU hardware operations, the translation can happen ahead of deployment. At run time, the ML app just needs a simple path to push the resultant operations to GPU.

## 2.2 GPU Trends We Exploit

**GPU virtual memory** Today, most integrated GPUs run on virtual address spaces. To configure a GPU's address space, the GPU stack populates the GPU's page tables and links GPU commands and shaders to the virtual addresses.

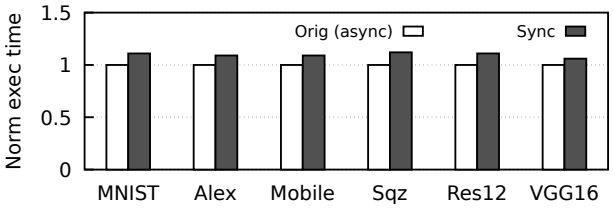


**Figure 3: Synchronous job submission increases NN inference delay modestly. ACL [22] + OpenCL on Mali G71**

**GPU autonomy** To reduce CPU overhead, a GPU job packs in much complexity – control flows, data dependency, and core schedule. The GPU parses a job's binary, resolves dependency, and dispatches compute to shader cores. A job may run as long as a few seconds without CPU intervention.

Take Mali G71 as an example: a job (called a "job chain") encloses multiple sub jobs and the dependencies of sub jobs as a chain. To run AlexNet for inference, the runtime (ACL v20.05) submits 45 GPU jobs, 5–6 GPU jobs per NN layer; the GPU hardware schedules a job over 8 shader cores.

**Synchronous job submission** Asynchronous GPU job submission is crucial to graphics, for which GPU executes smaller jobs. To hide job management delays, CPU streams jobs to GPU to keep the latter busy. Yet for compute, a job's management delay is amortized over the job's longer execution. For simplicity, shallow job queues in GPU drivers are common (max two outstanding jobs in Mali [16] and one in v3d/vc4 [56, 65]). Figure 3 shows that synchronous job submissions incur minor computation performance overhead: with six NN inferences on Mali G71 (see Table 6 for details), we find that enforcing synchronous jobs only adds 4% delays on average (max: 11%, min: 2%).

## 2.3 Design Choices

The trends above motivate the following choices.

GR focuses on synchronous GPU jobs, queuing them and executing one job at a time. It eschews recording or replaying concurrent GPU jobs. This deliberate decision ensures replay determinism: with concurrent GPU jobs, the number of possible CPU/GPU interactions would grow exponentially, making faithful replay difficult. The overhead of synchronous jobs is low as shown above.

For the same reason, GR eschews GPU sharing across apps during record and replay. Even without sharing, GR has important use cases. On smartphones, examples include background ML such as photo beautification and model fine-tuning; on headless smart devices without graphics, examples include ML pipelines for vision and prediction. Furthermore, the replayer can yield GPU to interactive apps with low delays (§5).

GR records at the lowest software level, i.e. the CPU/GPU boundary. This makes the replayer small and portable. By contrast, recording at higher levels, e.g. GPU APIs [43] or ML frameworks [52], would require the replayer to incorporate extensive runtime or driver functionalities.

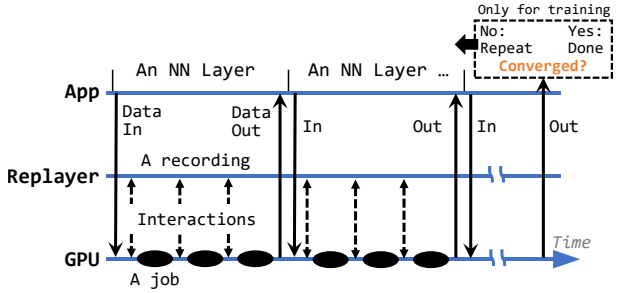


**Figure 4: Replaying NN execution with GR**

# 3 GR

## 3.1 Using GR

**A recording** encodes a fixed sequence of GPU jobs, including the CPU/GPU interactions and GPU memory dumps needed to execute these jobs. To capture a workload in one recording, the workload is required to execute all its jobs regardless of input, i.e. the workload's job graph contains no conditional branches that lead to different types of GPU jobs. The requirement does not preclude conditional branches *inside* a GPU job, i.e. among GPU instructions. This is because GR dumps a job's entire binary, which includes all the branches within, no matter whether they were exercised at the record time.

The above requirement is met by most, if not all, popular NNs, including all 44 NNs shipped with ACL, ncnn, and Tensorflow [22, 97, 98]. Note that some NNs (e.g. SqueezeNet and GoogLeNet) use "branches" to refer to routes in their job graphs, which are in fact executed unconditionally.

As examples, Figure 4 shows two common NN workloads.

- *NN inference* runs a sequence of NN layers $\{L_1 ... L_n\}$, each executing a sequence of GPU jobs unconditionally. To record, developers run the inference once and create recordings $\{R_1 ... R_n\}$, one recording per NN layer. An ML app supplies input and replays $\{R_1 ... R_n\}$ in sequence. After the replay, the replayer extracts output from GPU memory to the app.
- *NN training* runs a sequence of NN layers $\{L_1 ... L_n\}$ iteratively; after each iteration, it evaluates a predicate $\mathscr{P}$ and terminates if $\mathscr{P}$ shows the result has converged. To record, developers run one iteration and create a sequence of recordings $\{R_1 ... R_n\}$. They do not handle conditionals. An ML app runs a training iteration by replaying $\{R_1 ... R_n\}$. After the iteration, the app code on CPU evaluates $\mathscr{P}$. Unless $\mathscr{P}$ shows convergence, the app replays $\{R_1 ... R_n\}$ again on refined input.

The only exception to the above requirements, to our knowledge, is a conditional NN [45] using branches to choose among normal NNs. In this case, developers record branches as separate recordings; at run time, an ML app evaluates branch conditions on CPU and conditionally replays recordings. Conditional NNs are rare in practice to our knowledge.

**CPU/GPU coordination** Beyond the examples above, GR supports a workload consisting of interleaved CPU/GPU phases. For such a workload, the recorder generates multiple recordings, one recording per GPU phase. At run time, the app executes the CPU phases (not recorded) and replays for the GPU phases.

Such a hybrid execution is possible because GR stitches CPU and GPU phases by their input/output. To do so, the recorder automatically discovers input/output addresses for GPU recordings; before and after replaying each recording, the replayer deposits/extracts data to/from the GPU memory, respectively. In particular, CPU/GPU synchronizations (e.g. CPU waits for an OpenCL event) are recorded/replayed by GR as waits for GPU interrupts at the driver level. See Section 4 for details.

**Recording granularity** is a tradeoff between composability and efficiency; it does not affect correctness. In the examples above, developers record separate NN layers; alternatively, they may record a whole NN execution as one recording. While per-layer recordings allow apps to assemble new NNs programmatically, a monolithic recording improves replay efficiency due to reduction in data move and cross-job optimizations. Section 7 will evaluate these choices.

**Recording portability** By default, GR expects the GPU hardware (SKUs) and firmware versions used for record and replay to exactly match. As Section 6 will show, record/replay with different SKUs of the same family is possible, yet lightweight patching is needed.

**Developer efforts** are on three aspects. (1) Instrumenting a GPU driver to build a recorder. The effort is no more than 1K SLoC *per GPU family*, as the instrumentation applies to the family of GPU SKUs supported by the driver. See Section 4 for examples. (2) Recording their ML workloads. The effort is *per GPU SKU*. With minor patches, a recording can further be shared across GPU SKUs of the same family. (3) Building a replayer. The effort is a few K SLoC *per deployment environment*, e.g. for a TEE.

## 3.2 The GPU Model

GR builds on a small set of assumptions as summarized in Table 1. As the "least common denominator" of modern integrated GPUs, the assumptions constrain GPU behaviors to be a reproducible subset.

- *CPU/GPU interfaces* include memory-mapped registers, shared memory, and interrupts. Some GPUs, e.g. NVIDIA Tegra X1, may invoke DMA to access GPU registers [77]. All these interactions can be captured at the driver level.
- *Synchronous job submission.* Disabling asynchronous jobs avoids interrupt coalescing and the resultant replay divergence. The performance loss is modest as described in Section 2.2.
- *GPU virtual memory.* The replayer can manipulate the GPU page tables and load memory dumps to physical addresses of its choice. GR *can* work with legacy GPUs running on physical memory. Yet, the replayer must run on the same physical memory range as the record time.

**Replay correctness** The replayer offers the same level of correctness guarantee as the full GPU stack does: the replayer's assertion that a recorded workload (a series of GPU jobs) is completed is as sound as an assertion from the GPU stack. Our rationale is based on the GPU state.

A GPU state $< P, C, J >$ is all GPU-visible information affecting the GPU's execution outcome: P is the GPU's current protocol step, e.g. wait for commands; C is the GPU's hardware configuration; J is the job binary being executed. *We define a replay run as correct* if

**Table 1: Our GPU model fits popular integrated GPUs. *= To enforce sync job submission: Mali: reduce the job queue length; TegraX1: inject synchronization points to a command buffer; Adreno: check submitted job completion before a new command flush. NC: no changes**

| | Features | | | Interface Knowledge | | | |
|---|---|---|---|---|---|---|---|
| | MMIO | VirtMem | SyncJob* | JobStart | Pgtables | Reset | IRQ |
| Arm Mali [25] | Y | Y | [21] | [16] | [18] | [19] | [17] |
| Bcom v3d [56] | Y | Y | NC | [59] | [61] | [62] | [60] |
| Bcom vc4 [64] | Y | | NC | [65] | N/A | [67] | [66] |
| NV TegraX1 [78] | Y | Y | [85] | [84] | [83] | [79] | [86] |
| Qcom Adreno [68] | Y | Y | [69] | [72] | [73] | [71] | [70] |

the GPU at the replay time goes through the same state transitions as the record time.

The full GPU driver, as it runs, continuously assesses if the GPU state deviates from a correct transition path. The driver's only observations are *state-changing* events in CPU/GPU interactions: the events either changing the GPU state or indicating the GPU state has changed. State-changing events include: a register write; a register read returning a value different from the most recent read; a register read with side effect; interrupts.

Based on the rationale, the replayer asserts correctness based on matching state-changing events. If it observes the same sequence of state-changing events with all event parameters matched, then to the best knowledge of the GPU driver, the GPU makes the same state transitions and completes the recorded workload. The replay is correct per our definition.

Suppose a state divergence, such as silent data corruption, is missed by the replayer, it could have been missed by the full GPU driver as well. If we assume the driver is *gold*, i.e. it has made sufficient interactions to assess if GPU state has deviated from the correct transitions, then such silent divergences should neither occur to the driver nor the replayer.

**Nondeterministic CPU/GPU interaction** Even to repeat the same workload, the CPU/GPU interactions are likely to differ, e.g. CPU may observe diverging register values or receive extra/few interrupts. Hence, a raw trace cannot be replayed verbatim. The major nondeterminism sources are as follows. (1) Timing. For instance, a GPU job's delay may vary; the CPU may poll the same register for different times until its value changes. (2) GPU concurrency. The order of finishing concurrent jobs and the number of completion interrupts may vary. (3) Chip-level hardware resources, e.g. changes in a GPU's clockrate.

Because replay correctness only depends on GPU states, we treat nondeterminism as follows. (1) Nondeterminism not affecting GPU states. This includes most of the timing-related behaviors. The recorder discovers and summarizes them as replay actions, so that the replayer can tolerate (§4). (2) Affecting GPU states; preventable. This includes GPU concurrency and some configurable chip resources. We eliminate the nondeterminism sources, e.g. enforcing synchronous job submission as described in the GPU model above. (3) Affecting GPU states; non-preventable. This mainly includes strong contention and failures in chip resources, such as power failures. The replayer detects them and attempts re-execution.

# 4 RECORD

## 4.1 Interface Knowledge and Instrumentation

The knowledge needed by the recorder is in Table 1:

- The registers for starting a GPU job and for resetting GPU.
- The register pointing to the GPU page tables; the GPU page table's encoding for physical addresses. This allows to capture and restore the GPU virtual address space.
- The set of registers on which reads or writes do not change GPU state. This is to detect state-changing events.
- The events that a GPU interrupt handler starts and ends. Knowing them allows the replayer to enter and leave an interrupt context (via `eret`) just as the record time.
- (Optional) The events that the GPU hardware becomes busy or idle. The recorder uses them to remove unwanted delays.

We instrument the driver code: register accessors; register writes starting a GPU job; accessors of GPU page tables; interrupt handling. Many of these code locations are already abstracted as macros [57] or tracepoints [14]. We find manual instrumentation is more robust than tracing via page faults [1].

**Developer efforts** to extract interface knowledge and to instrument a driver are often amortized over a family of GPU SKUs supported by the driver. We confirm this is true for 6 GPU SKUs supported by the Arm Bifrost driver [25] and 17 GPU SKUs supported by the Adreno 6xx driver [68]. Although a driver may execute code conditionally depending on the GPU SKUs in use, the GPU interfaces in a GPU family, i.e. register names and semantics, are often identical.

## 4.2 Register Access

A recording consists of actions listed in Table 2. An action may summarize a sequence of register accesses showing nondeterminism without affecting GPU state. For instance, CPU may wait for GPU cache flush by polling a register [15, 58], where the number of register reads depends on the nondeterministic flush delay. Such polling is summarized by RegReadWait().

To do the above, the recorder recognizes nondeterministic register accesses that do not change GPU state. With the GPU interface knowledge described above, we inspect a driver's register accessors and instrument their callsites that match the patterns in Table 2. We tap in existing macros such as wait_for() [63, 81] and instrument tens of callsites per driver.

## 4.3 Dumping Proprietary Job Binaries

The recorder must record for a job's binary: (1) GPU commands for data copy or format conversion, often packed as nested arrays; (2) shaders, which include GPU code and metadata; (3) GPU page tables. A GPU binary is deeply linked against GPU virtual addresses: GPU commands contain pointers to each other, to the shader code, and to a job's input data; shaders also reference to code and data. Therefore, GR dumps all memory regions that may contain the job binary; to replay, GR restores the memory regions at their respective GPU virtual addresses.

**Time the dump** A GPU stack emits a job's binaries and updates GPU page tables lazily – often not until it is about to submit the

**Table 2: Replay actions in a recording**

| Replay Actions | Descriptions |
|---|---|
| **RegReadOnce**(r,val,ignore) | Read register @r once. A return value ≠ @val, then replay error. The read value may be ignored, in case of registers expected to return non-deterministic values. |
| **RegReadWait**(r,mask,val,timeout) | Poll register @r until its bits selected by @mask become @val. After the maximum wait time @timeout, report a replay error. |
| **RegWrite**(r,mask,val} | Write @val to register @r. @mask selects the written bits. Other bits are unchanged. |
| **SetGPUPgtable**(p) | Update the base address of GPU page table base to @p. To implement, the replayer updates a GPU register. |
| **MapGPUMem**(size,addr) | Allocate memory of @size and map to GPU virtual address @addr. The replayer loads a GPU page table dump and patch entries for relocation. |
| **UnMapGPUMem**(addr) | Unmap the GPU memory at @addr. Free physical memory. |
| **Upload**(d,addr) | Upload a memory dump @d to the GPU virtual address @addr, which must be mapped first. |
| **CopyTo/FromGPU**(gaddr,addr) | Move data between a GPU virtual address @gaddr and a CPU address @addr in the replayer's address space. For injecting input and extracting output. |
| **WaitIrq**(timeout) | Wait for a GPU interrupt before the next action. Interrupt handling is done by replaying the subsequent actions. Report a replay error if timeout. |

job. Accordingly, the recorder dumps GPU memory right before the driver kicks the GPU for a new job. At this moment, the runtime must have emitted the job's binary to the GPU memory; the memory dump must be consistent: synchronous job submission ensures no other GPU jobs are running at this time and mutating the memory.

**Locating job binaries in GPU memory** Memory dumps must include job binaries for correctness; they should exclude GPU buffers passed among jobs so that loading of memory dumps does not overwrite these buffers; they should leave out a job's scratch buffers as many as possible for space efficiency.

The challenge is that the recorder does not know exactly where GPU binaries are in memory: the GPU runtime directly emits the binaries to mmap'd GPU memory, bypassing the GPU driver and our recorder therein. A naive dump capturing all physical memory assigned to GPU can be as large as GBs. An optimization is to only dump memory mapped to GPU at the moment of job submission, which reduces a memory dump to MBs. Section 6 presents hardware-specific optimizations to further shrink memory dumps.

## 4.4 Locating Input and Output for a Recording

**Record by value vs. by address** A recording accepts one or more input buffers. By default, GR records an input buffer by address: the recorder captures the buffer's GPU address, allowing new data injected at the address at replay time. Use cases include an NN's input buffer. If developers intend to reuse an input buffer's values for replay, they may optionally annotate the input as "record by value" in the record harness. GR then captures the buffer values as part of memory dumps. Use cases include a buffer of NN parameters. An input recorded by value and by address simultaneously allows *optional* value overriding. Annotations only decide apps' responsibility for providing input data at the replay time; improper annotations do not break replay correctness.

**Discover input/output addresses** Recording *by value* is straightforward: just dump any memory region that *may* contain the input. Recording *by address* is more challenging: the recorder cannot track to which GPU address the runtime copies input, as the runtime is a kernel-bypassing blackbox; it does not know from which addresses the GPU code loads input, because the recorder cannot interpret the GPU code.

To reveal these memory locations, GR adopts a simple taint tracking. The record harness injects input magic values – synthetic, high-entropy data – and looks for them in GPU memory dumps. The rationale is that it is very unlikely that a high-entropy input (e.g. a 64x64 matrix with random elements) coincides another GPU memory region with identical values.

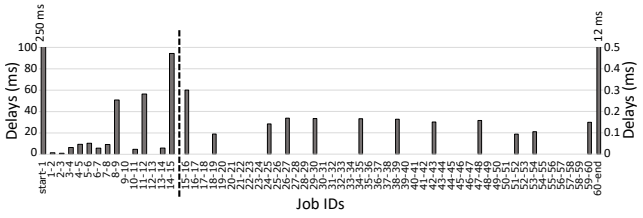


**Figure 5: Intervals between CPU/GPU interactions, accumulated by GPU job. Intervals among earlier jobs are longer than later ones. Workload: AlexNet inference. ACL [22] on Mali G71. Excluded: GPU busy time; parameters loading IO**

We took care of a few caveats. (1) The output often has lower entropy because it is smaller (e.g. a class label). In case of multiple matches of output magic in memory, GR repeats runs with different input magics to eliminate false matches. (2) The above technique cannot handle the case when the ML framework runs CPU code to reshape data before/after the data is moved to/from GPU. Fortunately, we did not see such a behavior in popular ML frameworks: Tensorflow, ncnn, and ACL. For efficiency, they always invoke GPU, if available, for data reshaping. While we are aware of rigorous, fine-grained taint tracking [30], our simpler technique is sufficient for locating GPU input/output. This saves us from configuring symbolic execution on a closed-source GPU runtime of tens of MBs, which requires expertise and non-trivial effort.

## 4.5 Pace Replay Actions

CPU cannot replay as fast as possible, otherwise GPU may fail to catch up. For example, CPU needs to delay after resetting the GPU clock/power for them to stabilize [20, 82] and delay after requesting GPU to flush cache [80].

The recorder sets a minimum interval $T$ for each action: if the replayer takes $t$ to execute the current action, it pauses for at least $T - t$ to before the next action. Setting proper intervals is non-trivial. When running the GPU stack, CPU paces its interactions with GPU intentionally (e.g. calling delay()) or unintentionally (e.g. running unrelated apps). The recorder should not preserve the observed intervals, as doing so will unnecessarily slow down the replay.

Figure 5 shows an example, where most long intervals are unintended delays from CPU: (1) Resource management, such as initialization of GPU memory management; (2) JIT generation of GPU commands and shaders; (3) OS asynchrony, such as scheduling delays; (4) Recording overhead, e.g. dumping GPU memory; (5) Abstraction tax, e.g. frequent IOCTLs. Doing none of these, the

replayer should simply skip the resultant intervals and fast-forward to the next action.

The challenge is to differentiate unintended delays from intended delays. It is unrealistic for the recorder to profile the complex, multi-threaded GPU stack. Instead, it follows a simple heuristics: *if the GPU hardware has been idle through one interval, the interval is safely skippable.* The rationale is that an idle GPU can always keep up with CPU's next action without pause. With this heuristics, we add tens of lines of code per driver, which can prove GPU idle for more than half of the observed intervals. Skipping them speeds up the replay significantly, as we will show in Section 7. The recorder simply preserves the remaining intervals for replay.

# 5 REPLAY

The replayer provides the following APIs. (1) *Init/Cleanup*: acquire or release the GPU with reset. (2) *Load*: load a recording file, verify its security properties, and allocate the required GPU memory. (3) *Replay*: replay the recording with input/output buffers supplied by the app. The replayer consists of a static verifier; an interpreter that parses/executes a recording in sequence; a nano GPU driver to be invoked by the interpreter.

## 5.1 Verification of Security Properties

The replayer statically verifies the following security properties. While a full GPU driver may implement similar checks, the replayer provides stronger guarantees due to its simplicity and independence of an OS kernel.

- *No illegal GPU register access by CPU.* A recording contains GPU register names, which are resolved by the replayer as addresses based on the CPU memory mapping.
- *No illegal memory access by GPU.* A recording only specifies sizes and GPU addresses of memory regions. It is up to the replayer to allocate the underlying physical pages and set up GPU page tables. The replayer ensures the allocated physical pages contain no sensitive data. The GPU MMU prevents GPU code from accessing any CPU memory.
- *Maximum GPU physical memory usage.* The replayer scans a recording for `MapGpuMem` entries (Table 2) to determine the GPU memory usage at any given moment. Based on the result, apps or the replayer can reject memory-hungry recordings.

The replayer cannot decide semantic correctness which is orthogonal to security. Section 7.1 will present discussions.

## 5.2 The Nano GPU Driver

The nano driver abstracts GPU hardware; it only has of 600 SLoC. Most driver functions directly map to replay actions: mapping GPU registers to CPU addresses, copying data in and out of GPU memory, rewriting the GPU page table entries for loading memory dumps, etc. The driver includes a bare minimum interrupt handler, which simply switches the CPU to the interrupt context and continues to replay the subsequent actions. The interrupt management, such as waiting for an interrupt, acknowledging an interrupt, and checking interrupt sources, is done implicitly by replaying the corresponding actions.

**Table 3: GR implementations. * = used in evaluation. See Table 6 for evaluated recordings**

| GPU HW (Boards) | Compatible GPU stacks | Recordings | Replayers |
|---|---|---|---|
| Mali-G71 * (Hikey960) | 1. ACL + Open CL * 2. DeepCL + OpenCL * 3. ACL + GLES compute 4. Tensorflow + ACL + OpenCL Driver: Arm Mali r23p0-01re10 | Inference: 18 Training: 1 | 1.User* 2.TEE |
| Mali G52 (Odroid N2) | | | |
| Mali G31 (Odroid C4) | | | |
| Brcm v3d * (Raspberry Pi 4) | 1. ncnn + Vulkan * 2. Py-videocore6 Driver: drm/v3d in Linux 5.11 | Inference: 15 Math: 2 | 1.Kernel* 2.Baremetal |

## 5.3 GPU Handoff and Preemption

During replay, the replayer fully owns the GPU and does not share with other apps. Before and after a replay, it soft-resets the GPU, ensuring the GPU starts from a clean state without data leaking, e.g. no subsequent apps will see unflushed GPU cache. The replayer allows the OS to reset and preempt the GPU at any time (e.g. yielding to an interactive app) without waiting for ongoing GPU jobs to complete. Hence, preemption incurs short delays. A preemption disrupts the current replay. To mitigate it, we implement optional checkpointing: periodically making copies of GPU memory and registers. A disrupted replay resume from the most recent checkpoint. Section 7 evaluates preemption and checkpointing experimentally.

## 5.4 Handling Replay Failures

Replay failures are GPU state divergences due to non-preventable nondeterminism at run time. Based on our GPU model (§3), the replayer will not miss detecting any state divergences the full GPU stack can detect. When the replayer faces failures, it attempts to recover through re-execution: resetting the GPU and starting over the whole recording; if the divergence persists, the replayer injects additional delay to the action intervals that precede the divergence occurrence.

Re-execution with delays can overcome transient failures and many timing-related failures, which are the most common failures based on the driver code comments, documentations, and our own experience. Examples include an underclocked GPU for replay fails to keep up with the replay actions; high contention on shared memory cause GPU jobs to timeout.

Re-execution cannot overcome persistent failures, e.g. reoccurring hardware errors. A full driver is unlikely to overcome such errors either. In this case, the replayer seeks to emit meaningful errors as the full driver does: it reports the failed action and the associated source locations in the full driver.

# 6 IMPLEMENTATIONS AND EXPERIENCES

As summarized in Table 3, we implement GR for Arm Mali (reported to ship billions of devices [24]) and Broadcom v3d (the GPU for RaspberryPi 4). The current implementations work for a variety of ML workloads (inference, training, and math kernels), programming abstractions (OpenCL, Vulkan, and GLES compute), and GPU runtimes (the official ones as well an experimental runtime fully written in Python).

## 6.1 The Recorder for Arm Mali

We implement a recorder for Mali Bifrost family; it records complex and diverse GPU workloads, including 18 inferences and 1 training, some of which will be evaluated in Section 7. Leveraging ArmNN [23], our prototype for Mali is compatible with TensorFlow NN models. We add around 700 SLoC to Mali's stock driver, which is 1% of the driver's 45K SLoC.

Our recorder exploits Mali's page permission to shrink memory dumps. If a GPU-visible page is mapped as *executable* to GPU, the recorder treats the page as part of job chains and dumps it. If a GPU-visible page is *non-executable* to GPU and is *unmapped* from CPU, the recorder treats the page as part of GPU internal buffers and excludes it from dumping. This is because GPU-visible pages are mapped to CPU on demand; an unmapped page must never have been accessed by CPU.

## 6.2 The Recorder for Broadcom V3D

Our recorder for v3d adds around 1K SLoC to v3d's stock driver. To dump GPU memory, the recorder follows v3d's registers pointing to shaders and control lists. It handles the cases where lists/shaders may contain pointers to other lists/shaders of the same or different memory regions. Unlike Mali, the v3d page tables lack executable bits. Being conservative, the recorder has to dump more pages than Mali in general. To further exclude unwanted GPU memory regions from dumping, the recorder exploits as hints the flags of syscalls that allocate the GPU memory. To reduce the storage overhead, the recorder compresses the memory dumps with zlib [47].

## 6.3 Replayers in Various Environments

**A baremetal implementation** As a proof of concept, we built a standalone replayer for v3d without any OS.

To avoid filesystems, we statically incorporate compressed recordings in the replayer binary. The whole executable binary (excluding recordings) is around 50 KB. In the executable, the replayer itself is about 8 KB. We link zlib [47] for recording decompression (about 9 KB) and a baremetal library [94] for Rpi4. The library functions include CPU booting, interrupts, exception, and firmware interfaces (about 15 KB executable); CPU cache, MMU, and page allocation (4 KB); timers and delays (4 KB); string manipulation and linked lists (9 KB).

A major challenge is to bring up the GPU power and clocks. Modern GPUs depend on power/clock domains at the SoC level [104]. Linux configures power and clocks by accessing various registers, sometimes communicating with the SoC firmware [93]. The process is complex, SoC-specific, and often poorly documented. While replayers at the user or the kernel level reuse the configuration done by the kernel transparently, the baremetal replayer must configure GPU power and clocks itself. To do so, we instrument the Linux kernel, extract the register/firmware access, and port it to the replayer.

**A user-level implementation** We built a replayer for Mali as a daemon with kernel bypassing [36, 37]. To support the daemon, the kernel parses the device tree and exposes to the userspace the GPU registers, memory regions, and interrupts. The replayer maps GPU registers and memory via mmap(); it directly manipulates GPU page tables via mapped memory; it receives GPU interrupts by select() on the GPU device file.

**Table 4: Codebase comparisons. Binaries are stripped.**

| | The original stack | | | Ours | |
|---|---|---|---|---|---|
| GPU | ML Framework | Runtime | Driver | Rec | Replayer |
| Mali Bifrost | • ACL: 500 KSLoC, 30MB<br>• DeepCL: 18 KSLoC, 2.1MB | libmali.so: 48 MB | 45K SLoC | 0.7K SLoC | • Usr+kernl: 2.2+0.6 KSLoC 25KB+20KB<br>• In-TEE: 1K SLoC, 10 KB |
| Bcm v3d | • ncnn: 223 KSLoC, 11 MB | libvulkan_broadcom.so: 7 MB | 3K SLoC | 1K SLoC | • Kernel only: 1K SLoC; 107 KB (whole driver)<br>• Baremetal: 4K SLoC, 50 KB |

**A kernel-level implementation** We built a replayer for v3d as a kernel module. The replayer directly invokes many functions of the stock GPU driver, e.g. for handling GPU interrupts and memory exceptions; it exposes several IOCTL commands for an app to load a recording and inject/extract input/output. Once turned on, the replayer disables the execution of the stock driver until replay completion or GPU preemption.

**A TrustZone implementation** We built a replayer for Mali in the secure world on the Hikey960 board. We added a small driver (in 100 SLoC) to the TrustZone kernel (OPTEE) for switching the mappings of GPU register and memory between the normal/secure worlds. The replayer is a straightforward porting of the user-level replayer. The replayer is in around 1K SLoC, only 0.3% of the whole OPTEE (300K SLoC).

## 6.4 Reusing Recordings Across GPU SKUs

It is possible to share recordings across GPUs of the same family: these GPUs are likely to share job formats, shader instruction sets, and most register/page table semantics. We analyze three Mali GPUs: G31 (low end), G52 (mainstream), and G71 (high end). We manage to patch a recording from G31/G52 and replay it on G71. Our patch adjusts: (1) Page table format: re-arranging the permission bits in the G31 page table entries, which are in a different order than G71 due to G31's LPAE support. (2) MMU configuration: flipping a bit in the translation configuration register to enable read-allocation caching expected by G71. (3) Core scheduling hints: changing the value of core affinity register (JS_AFFINITY) so a job is mapped to G71's all 8 shader cores. Overall, the patch includes fixes for two registers per recording and one register per job. Section 7.5 reports replay performance of a patched recording.

Despite our limited success above, we note that it would be difficult to replay with fewer GPU resources (e.g. record on G71 and replay on G31). This is because doing so would require (1) proprietary GPU knowledge, e.g. to relocate GPU shaders and compact memory and (2) a more sophisticated replayer, e.g. to swap GPU memory.

# 7 EVALUATION

We evaluate GR with the following questions.

- Does GR make GPU computations more secure?
- Overhead: Do recordings increase app sizes? How does the replay speed compared to that of the original GPU stack?
- Do our key design choices matter?

**Table 5: GR eliminates common vulnerabilities and exposures (CVEs) in the GPU stack**

| GR's design (D1–3: scenarios) | Example CVEs | Description | Effect | Vulnerability |
|---|---|---|---|---|
| **Remove GPU runtime from app (D1,D2,D3)** | CVE-2014-1376, High | Improper restriction of OpenCL calls [3] | Arbitrary code execution | `App.I` |
| | CVE-2019-5068, Med | Exploitable shared memory permissions [9] | Unauthorized mem access | `App.C` |
| | CVE-2018-6253, Med | Malformed shaders cause infinite recursion [5] | App hang | `App.A/GPU.A` |
| **Remove GPU driver (D2, D3)** | CVE-2017-18643, High | Leak of GPU context address of GPU mem region [4] | Sensitive info disclosure | `Kernel.C` |
| | CVE-2019-20577, High | Invalid address mapping of GPU buffer [8] | Kernel crash | `Kernel.I` |
| | CVE-2020-11179, High | Race condition by overwriting ring buffer [10] | Arbitrary kernel mem r/w | `Kernel.I` |
| | CVE-2019-10520, Med | Continuous GPU mem allocating via IOCTL [6] | GPU mem exhausted | `Kernel.A` |
| | CVE-2014-0972, N/A | Lack of write protection for IOMMU page table [2] | Kernel mem corruption | `Kernel.I` |
| **Disable fine-grained GPU sharing (D1,D2)** | CVE-2019-14615, Med | Learning app's secret from GPU register file [7] | App data leak | `App.C` |

`I`: Integrity; `C`: Confidentiality; `A`: Availability

## 7.1 Analysis

**Semantic bugs**, e.g. emission of wrong GPU commands, may pre-exist in the GPU stack for recording. Such bugs may propagate to the target machines, resulting in wrong replay results. GR neither mitigates nor exacerbates these bugs. Fortunately, semantic bugs are rare in production GPU stacks to our knowledge. GR's recorder and replayer may introduce semantic bugs. The chance, however, is slim: as shown in Table 4, they are small as a few K SLoC with simple logic. Our validation experiments in Section 7.2 strengthen our confidence. We next focus on security, a major objective of GR.

**Threat models** Corresponding to three deployment scenarios (D1-3) in Section 1: (D1) a user/kernel-level replayer on a commodity OS trusts the OS while facing local unprivileged and remote adversaries; (D2) a replayer in TEE trusts the TEE kernel while facing the local OS adversaries and remote ones; (D3) a baremetal replayer only faces remote adversaries.

We assume it is difficult to compromise the recording environment, including OS, GPU stack, and code signing: doing so often requires long campaigns to infiltrate the developers' network where risk management is likely rigorous [31]. We will nevertheless discuss the consequences of such attacks.

**Thwarted attacks** corresponding to three deployment scenarios are as follows. (D1) When a replayer coexists with the GPU stack on the same OS, the app using the replayer is free of GPU runtime vulnerabilities which cause unauthorized access to app memory [9], arbitrary code execution in the app [3], and app hang [5]. (D2) When a replayer runs in TEE and coexists with the GPU stack outside the TEE, the app is free from attacks against the GPU stack by the local OS. (D3) When a replayer completely replaces the GPU stack in a system, the system is free from GPU stack vulnerabilities that cause kernel information disclosure [4], kernel crash [8], and kernel memory corruption [2]. Table 5 summarizes the eliminated vulnerabilities.

**Attacks against GR** (1) *Attacks against developers' machines or recording distribution.* This is difficult as described above. Nevertheless, successful adversaries may fabricate recordings containing arbitrary actions and memory dumps. A fabricated recording may hang GPU but cannot break security guarantees enforced by the replayer, e.g. no illegal register access (§5.1). (2) *Attacks against the replayer or its TCB.* The chance of replayer vulnerabilities is slim due to simplicity. Nevertheless, successful adversaries may subvert recording verification. By compromising a user-level replayer or kernel-level/baremetal replayers, adversaries may gain unrestricted access to the GPU or the whole machine, respectively.

**Table 6: NN inference for evaluation. Choices of NNs for Mali vs. v3d are slightly different because their ML frameworks do not implement exactly the same set of NNs**

| Model (#layers) | GPU Mem (MB) | # Jobs | # RegIO | RecSize (MB) Unzip | RecSize (MB) Zipped |
|---|---|---|---|---|---|
| MNIST (4) | 4.7 | 18 | 2977 | 2.2 | 0.1 |
| AlexNet (8) | 683.2 | 45 | 8542 | 3.8 | 0.2 |
| MobileNet (28) | 44.9 | 54 | 12663 | 2.7 | 0.1 |
| SqueezeNet (26) | 36.9 | 71 | 12129 | 2.8 | 0.1 |
| ResNet12 (12) | 261.3 | 78 | 15934 | 3.4 | 0.1 |
| VGG16 (16) | 1738.3 | 71 | 23056 | 6.4 | 0.4 |

**(a)** Mali Bifrost

| Model (#layers) | GPU Mem (MB) | # Jobs | # RegIO | RecSize (MB) Unzip | RecSize (MB) Zipped |
|---|---|---|---|---|---|
| YOLOv4-tiny (38) | 75.7 | 92 | 4708 | 2.0 | 0.3 |
| AlexNet (8) | 139.2 | 40 | 2024 | 9.5 | 0.3 |
| MobileNet (28) | 42.3 | 66 | 3057 | 4.7 | 0.2 |
| SqueezeNet (26) | 26.8 | 85 | 4323 | 18.0 | 0.5 |
| ResNet18 (18) | 87.0 | 119 | 5253 | 66.0 | 1.7 |
| VGG16 (16) | 423.5 | 71 | 3742 | 4.4 | 0.3 |

**(b)** v3d

## 7.2 Validation of Replay Correctness

We add extensive logging to both the original driver code and the replayer: they log *all* the GPU registers on each CPU/GPU interaction; they take snapshots of GPU memory before each job submission and after each interrupt. We then compare these logs across runs and look for any discrepancies.

We run two inference workloads, MNIST and AlexNet, each for 1,000 times. In each replay run, we create strong interferences with GPU by co-executing CPU programs that: (1) generate high memory traffic which contends with GPU register and memory access; (2) burn CPU cycles to trigger SoC thermal throttling. We also repeat the tests with GPU running at different clockrates. Each MNIST (AlexNet) run generates a log of 3K (8K) registers accesses and 46 (120) memory snapshots, respectively. The only detected discrepancies are the numbers of register polling and GPU job delays, which do not affect GPU states; all other logs match.

We further verify that the replayer produces correct compute results. We replay all the workloads in Table 6 (a) 2,000 times each. We create random input, inject interference, and compare the GPU's

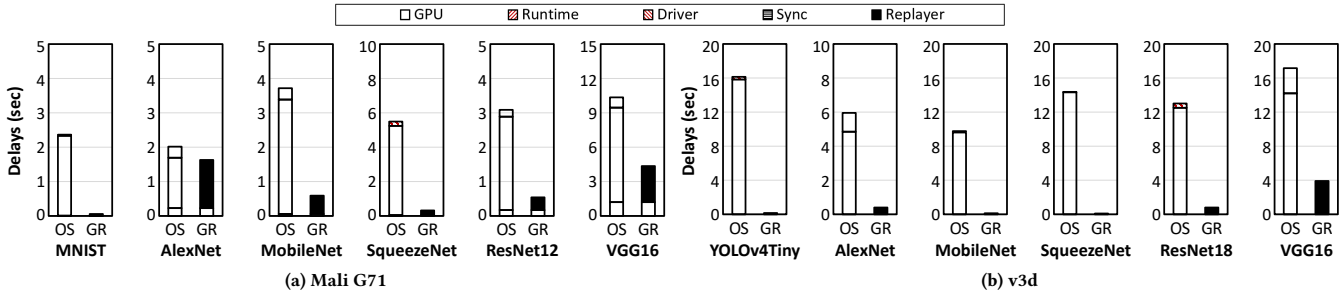


**Figure 6: Startup delays prior to NN inference. The replayer (GR) takes much less time than the original GPU stack (OS).**

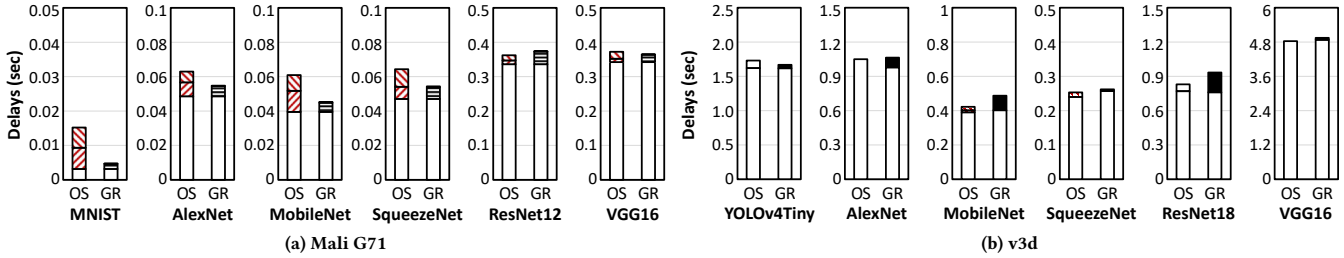


**Figure 7: NN inference delays. The replayer (GR) incurs similar delays as compared to the original GPU stack (OS).**

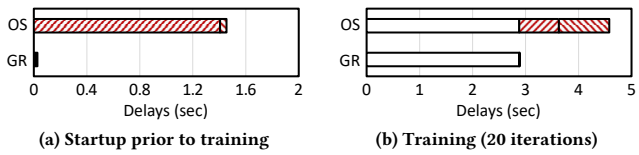


**Figure 8: NN training delays. Benchmark: MNIST training atop DeepCL + OpenCL on Mali G71 (OS: orig stack; GR: GR).**

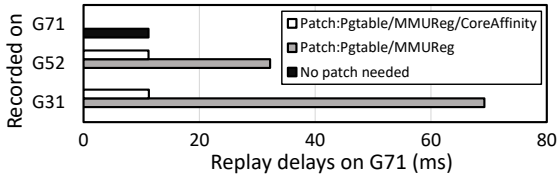


**Figure 9: Mali G71 can replay recordings from other GPUs at full hardware speed. Benchmark: 16M elements vecadd**

outcome with the reference answers computed by CPU. The replayer always gives the correct results. The reasons are (1) our design enforces determinism, e.g. by disallowing concurrent kernels and (2) no hardware errors during our benchmarks.

**Failure detection & recovery** We run a CPU program to artificially inject transient, non-preventable failures during the replay of AlexNet: (1) offlining GPU cores forcibly and (2) corrupting GPU page table entries. The replayer successfully detects the failures as diverging reads of a status register and GPU memory exceptions, because the original driver checks the register and enables the interrupt. Re-execution resets GPU cores and re-populates the page table, finishing the execution.

## 7.3 Memory Overheads

**Recording sizes** A GPU recording is as small as a few hundred KBs when compressed as shown in Table 6. The size is a small fraction of a smartphone app, which is often tens of MBs [28]. Of a recording, memory dumps are dominant, e.g. on average 72% for Mali. Some v3d recordings are as large as tens of MBs uncompressed because they contain memory regions that the recorder cannot safely rule out from dumping. Yet, these memory regions are likely GPU's internal buffers; they contain numerous zeros and are highly compressible.

**CPU/GPU memory** The replayer's GPU memory consumptions show a negligible difference compared to that of the original GPU stack, because the replayer maps all the GPU memory as the latter does. The replayer's CPU memory consumption ranges from 2 – 10 MB (average 5 MB) when executing NN inference, much lower than the original stack (220 – 310 MB, average 270 MB). This is because the replayer runs a much smaller codebase; by directly loading GPU memory dumps, it avoids the major memory consumers such as GPU contexts, NN optimizations, and JIT commands/shader generation.

## 7.4 Replay Speed

We study the inference delays on a variety of NNs as listed in Table 6. Compared to the original GPU stacks (native execution), the replayer's startup delays are significantly lower: by 26% – 98% (Mali) and lower by 77% – 99% (v3d); Our replay is even 20% faster (Mali) and only 5% slower (v3d) on average. Our overhead is much lower than prior TEE systems for secure GPU computation [46, 99, 100].

**Startup delays** We measure the startup delay from the time the testing app initializing a GPU context until the first GPU job is ready for submission. Figure 6 shows the results. Both the stacks for Mali and v3d take seconds to start up, yet showing different bottlenecks: Mali is bottlenecked at the runtime (libMali.so) compiling shaders

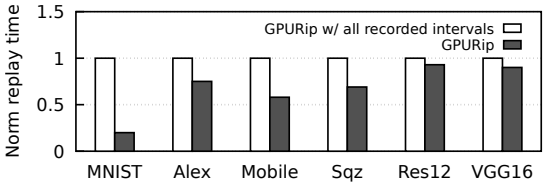


**Figure 10: GR removes unnecessary intervals between replay actions. Benchmark: ACL NN inference atop Mali G71**

and allocating memory; v3d is at the framework (ncnn) loading NNs and optimizing pipelines. By contrast, the replayer spends most time on GPU reset, loading of memory dumps, and reconstructing page tables.

Our startup comparison should *not* be interpreted as a quantitative conclusion, though. We are aware of optimizations to mitigate bottlenecks in GPU startup, e.g. caching compiled shaders [32] or built NN pipelines [96]. Compared to these point solutions, GR is systematic and pushes the caching idea to its extreme – caching the whole initialization outcome at the lowest software layer.

**NN inference delays** We measured the delay from the moment an app starting an inference with its ML framework to the moment app getting the outcome. The results are shown in Figure 7. In general, on benchmarks where the CPU overhead is significant, the replayer sees lower delay than the full stack, e.g. by 70% on MNIST (Mali). This is because the replayer minimizes user-level executions, kernel-level memory management, and user/kernel crossings such as IOCTLs. On larger NNs with long GPU computation, GR sees diminishing advantages and sometimes disadvantages. GR's major overheads are (1) loading of memory dumps containing unneeded data that GR cannot exclude, e.g. 66 MBs for ResNet18 (v3d); (2) short GPU idles from synchronous jobs (0.5% – 3% on Mali); (3) pause between replay actions.

**NN training delays** GR shows similar advantages. Our benchmark is MNIST with DeepCL [89] atop OpenCL. Each training iteration runs 72 GPU jobs and 5.7K register accesses. DeepCL already submits jobs synchronously with CLFlush(). As shown in Figure 8, the replayer incurs 99% less startup delay due to the removal of parameter parsing and shader compilation. Over 20 iterations, the replayer incurs 40% less delays because it avoids DeepCL and the OpenCL runtime.

## 7.5 Validation of Key Designs

**Cross-GPU record/replay** (§6.4) Figure 9 demonstrates it on different GPUs of the same family. We have recorded the same workload on Arm Mali G31 (low-end, 1 shader core) and G52 (mainstream, 2 cores). We attempt to replay the two recordings on Mali G71 (high-end, 8 cores). With patched GPU page tables and MMU register values, the replay completes with correct results, albeit with 4x – 8x lower performance. Further patching the core affinity register makes the replay utilize G71's all 8 shader cores, resulting in full performance.

**Skip intervals in replay** (§4.5) Without the technique, the replayer's NN inference will be 1.1x – 4.9x longer, as shown in Figure 10; startup delays will be up to two orders of magnitude longer, closer to that of a full stack (not shown in the figure).

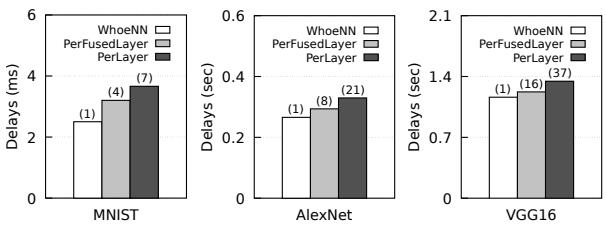


**Figure 11: NN inference delays (including startup) with various granularities. The count of recordings is annotated.**

**Impact of recording granularity** We tested three granularities: one monolithic recording per NN (high efficiency); one recording per NN layer (high composability); per fused layer with layer fusion done by ACL [22] (a middle ground). Figure 11 shows that recordings of fused layers incur only 15% longer delays on average than a monolithic recording. The additional delays come from replayer startup (see Figure 6). We conclude that for NN inference, recording every fused layer is a useful tradeoff between composability and efficiency.

**Preemption delay for interactiveness** (§5.3) We measure the delay perceived by an interactive app when it requests to preempt GPU from the replayer. On both tested GPUs, the delay is below 1 ms, which translates to minor performance degradation, e.g. loss of 1 FPS for a 60 FPS app. The reason is preemption simplicity: a preemption primarily flushes GPU cache and GPU TLB followed by a GPU soft reset.

**Checkpoint & restore** (§5.3) Our results show that GPU state checkpointing is generally inferior to re-executing the whole replay. For instance, MobileNet making one checkpoint every 16 GPU jobs (50–60 jobs in total) slows down the whole NN execution by 8x. The primary cause is memory dump. MobileNet takes 140 ms to dump all GPU memory (51 MBs) while re-executing the NN takes only 45 ms.

## 8 RELATED WORK

**Record and replay** was primarily used for diagnosis and debugging [29, 48, 106]. It has been applied to mobile UI apps [38, 91], web apps [75], virtual machines [35], networks [102], and whole systems [42]. None of prior work has applied the idea to the CPU/GPU interactions. Related to GR, Replaying syscalls and framework calls have been popular in reverse engineering GPU runtimes [12, 27, 40, 55] and reducing GPU scheduling overhead [52], respectively. Unlike them, GR records at the CPU/GPU boundary and therefore achieves the goal of a lean, trustworthy replayer.

**Refactoring GPU stacks** To leverage TEE, recent works isolate part of or the whole GPU stack for security. Sugar [107] subsumes a full GPU stack to an app's address space. Graviton [100] pushes the function of isolation and resource management from OS to a GPU's command processor. Telekine [43] spans a GPU stack between local and cloud machines at the API boundary. HIX [46] ports the entire GPU stack to a secure enclave and restricts the IO interconnect. HETEE [109] instantiates dedicated hardware controller and fabric to isolate the use of GPU. While efficacy has been shown, a key drawback is the high engineering effort (e.g. deep modifications of

GPU software/hardware), limited to a special hardware component (e.g. software-defined PCIe fabric) and/or likely loss of compatibility with stock GPU stacks. Contrasting to all the above approaches of *spatial* refactoring, GR can be viewed as *temporal* refactoring of a GPU stack – between the development time and the run time.

**GPU virtualization** often interposes between GPU stack layers in order to intercept and forward interactions, e.g. to a hypervisor [95] or to a remote server [34]. The interposed interfaces include GPU APIs [34, 108] and GPU MMIO [33, 95]. Notably, AvA [108] records and replays API calls during GPU VM migration. GR shares the principle of interposition and gives it a new use – for recording computations ahead of time and later replaying it on a different machine.

**Optimizing ML on GPU** Much work has optimized mobile ML, e.g. by exploiting CPU/GPU heterogeneity [51]. Notably, recent studies found CPU's software inefficiency leaving GPU underutilized, e.g. suboptimal CLFlush [50] or expensive data transformation [103]. While prior solutions fix the causes of inefficiency in the GPU stack [50], GR offers *blind* fixes without knowing the causes: replaying the CPU outcome (e.g. shader code) and removing GPU idle intervals.

**Secure ML** Much work has transformed ML workloads rather than the GPU stack; outsourcing security-sensitive compute to TEE, they preserve data/model privacy or ensure compute integrity [41, 54, 74]. They often support CPU-only compute and their workload transformation is orthogonal to GR. While Slalom [99] proposed secure GPU offloading, it requires GPU stack in TEE and is limited to linear operations.

## 9 CONCLUDING REMARKS

**Broader applicability** (1) The idea of GR applies to discrete GPUs. Our GPU hardware assumptions (§3.2) see counterparts on discrete GPUs albeit in different forms, e.g. registers and memory mapped via PCIe. In particular, GR can leverage NVIDIA MIG [87] to enable app multiplexing: the replayer can own an MIG instance while other apps use other instances; they are multiplexed on a physical GPU transparently by MIG. However, discrete GPUs raise new challenges including more complex CPU/GPU interactions, higher GPU dynamism, and recording cost due to larger memory dumps. (2) While this paper focuses on ML workloads, GR can extend to more GPU computation including numeric analysis and physics simulation. (3) GR's principle is applicable to other TEEs. A replayer in an SGX enclave is possible, but would need additional support such as MMIO remoting or SGX's extension for MMIO [46] because by default enclaves cannot directly access GPU registers.

**Recommendation to GPU vendors** We build GR without vendor support, respecting the GPU runtime blackbox (§2) and only reasoning/modifying at the driver level. It would be more attractive if vendors can implement GR and maintain as part of their GPU stacks. On one hand, the vendors can make GR more robust with first-party knowledge (e.g. GPU state machines for detecting state divergence) and lightweight interface augmentation (e.g. the runtime directly discloses a job's input/output addresses). On the other hand, the modifications to GPU stacks are very minor and the GPU runtime internals still remain proprietary.

## ACKNOWLEDGMENTS

The authors were supported in part by NSF awards #1846102, #1919197, and #2106893. We thank our shepherd, Dr. Mark Silberstein, and the anonymous reviewers for their insightful suggestions.

## REFERENCES

[1] [n.d.]. In-kernel memory mapped I/O tracing. https://www.kernel.org/doc/html/latest/trace/mmiotrace.html.
[2] 2014. CVE-2014-0972: Unprivileged GPU command streams can change the IOMMU page table. https://nvd.nist.gov/vuln/detail/CVE-2014-0972.
[3] 2014. CVE-2014-1376: Improper Restriction to Unspecified OpenCL API calls. https://nvd.nist.gov/vuln/detail/CVE-2014-1376.
[4] 2017. CVE-2017-18643: Information disclosure on Samsung gpu. https://nvd.nist.gov/vuln/detail/CVE-2017-18643.
[5] 2018. CVE-2018-6253: Vulnerability in the OpenGl Usermode Drivers. https://nvd.nist.gov/vuln/detail/CVE-2018-6253.
[6] 2019. CVE-2019-10520: out of memory in snapdragon. https://nvd.nist.gov/vuln/detail/cve-2019-10520.
[7] 2019. CVE-2019-14615: Information Leakage Vulnerability on the Intel Integrated GPU Architecture. https://nvd.nist.gov/vuln/detail/CVE-2019-14615.
[8] 2019. CVE-2019-20577: SMMU page fault in MALI GPU Driver. https://nvd.nist.gov/vuln/detail/CVE-2019-20577.
[9] 2019. CVE-2019-5068: Exploitable Shared Memory Permission Vulnerability in Mesa 3D Graphics Library. https://nvd.nist.gov/vuln/detail/CVE-2019-5068.
[10] 2020. CVE-2020-11179: Qualcomm Adreno GPU Ringbuffer Corruption / Protected Mode Bypass. https://nvd.nist.gov/vuln/detail/CVE-2020-11179.
[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi
[12] alyssa rosenzweig. [n.d.]. Dissecting the apple m1 gpu. https://rosenzweig.io/blog/asahi-gpu-part-1.html.
[13] Alyssa Rosenzweig. [n.d.]. Dissecting the Apple M1 GPU, part II. https://rosenzweig.io/blog/asahi-gpu-part-2.html.
[14] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: Config_Mali_System_Trace. https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/mali_kbase.h#364.
[15] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: kbase_cache_clean_worker(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/backend/gpu/mali_kbase_instr_backend.c#349.
[16] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: kbase_job_hw_submit(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/backend/gpu/mali_kbase_jm_hw.c#56.
[17] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: kbase_job_irq_handler(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/backend/gpu/mali_kbase_irq_linux.c#45.
[18] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: kbase_mmu_insert_pages(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/mali_kbase_mmu.c#606.
[19] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: kbase_pm_init_hw(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/backend/gpu/mali_kbase_pm_driver.c#1162.
[20] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: kbase_pm_set_policy(). https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/backend/gpu/mali_kbase_pm_policy.c#557.
[21] Android kernel. [n.d.]. Arm Bifrost Graphics Driver: SLOT_RB_SIZE. https://android.googlesource.com/kernel/arm64/+/refs/tags/android-11.0.0_r0.67/drivers/gpu/arm/midgard/backend/gpu/mali_kbase_jm_defs.h#27.
[22] Arm. [n.d.]. Arm Compute Library. https://github.com/ARM-software/ComputeLibrary.
[23] Arm. [n.d.]. ArmNN. https://github.com/ARM-software/armnn.
[24] Arm. [n.d.]. Mali for all occasions: New GPUs for all graphics workloads, use cases and consumer devices. https://community.arm.com/developer/tools-software/graphics/b/blog/posts/new-suite-of-arm-mali-gpus/.
[25] Arm. [n.d.]. Open Source Mali Bifrost GPU Kernel Drivers. https://developer.arm.com/tools-and-software/graphics-and-gaming/mali-drivers/bifrost-kernel.
[26] Arm. [n.d.]. Vulkan SDK for Android. https://github.com/ARM-software/vulkan-sdk.

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

[28] Brendon Boshell. [n.d.]. Average App File Size: Data for Android and iOS Mobile Apps. https://sweetpricing.com/blog/2017/02/average-app-file-size/.

[29] Ang Chen, W. Brad Moore, Hanjun Xiao, Andreas Haeberlen, Linh Thi Xuan Phan, Micah Sherr, and Wenchao Zhou. 2014. Detecting Covert Timing Channels with Time-Deterministic Replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 541–554. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chen_ang

[30] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. Association for Computing Machinery, New York, NY, USA, 391–402. https://doi.org/10.1145/1455770.1455820

[31] Cybersecurity and Infrastructure Security Agency, NIST. [n.d.]. Defending Against Software Supply Chain Atatcks. https://www.cisa.gov/publication/software-supply-chain-attacks/.

[32] Arm Developer. [n.d.]. Mali Offline Compiler. https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/mali-offline-compiler.

[33] Micah Dowty and Jeremy Sugerman. 2009. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 73–82. https://doi.org/10.1145/1618525.1618534

[34] Jose Duato, Antonio J Pena, Federico Silla, Juan C Fernandez, Rafael Mayo, and Enrique S Quintana-Orti. 2011. Enabling CUDA acceleration within virtual machines using rCUDA. In *2011 18th International Conference on High Performance Computing*. IEEE, 1–10.

[35] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2003. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2003), 211–224. https://doi.org/10.1145/844128.844148

[36] Linux Foundation. [n.d.]. Data Plane Development Kit (DPDK). http://www.dpdk.org.

[37] Linux Foundation. [n.d.]. The User Space IO (UIO). https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html.

[38] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *2013 35th International Conference on Software Engineering (ICSE)*. 72–81.

[39] Google. [n.d.]. Tensorflow GPU support. https://www.tensorflow.org/install/gpu.

[40] Grate. [n.d.]. Open Source reverse-engineering tools aiming at NVIDIA Tegra2+3D engine. https://github.com/grate-driver/grate.

[41] Zhongshu Gu, Heqing Huang, Jialong Zhang, Dong Su, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. 2018. Securing Input Data of Deep Learning Inference Systems via Partitioned Enclave Execution. *CoRR* abs/1807.00969 (2018). arXiv:1807.00969 http://arxiv.org/abs/1807.00969

[42] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 193–208.

[43] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. 2020. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 817–833. https://www.usenix.org/conference/nsdi20/presentation/hunt

[44] id-Software. [n.d.]. DOOM-3-BFG: LoadGLSLShader(). https://github.com/id-Software/DOOM-3-BFG/blob/master/neo/renderer/RenderProgs_GLSL.cpp#L960.

[45] Yani Ioannou, Duncan P. Robertson, Darko Zikic, Peter Kontschieder, Jamie Shotton, Matthew Brown, and Antonio Criminisi. 2016. Decision Forests, Convolutional Networks and the Models in-Between. *CoRR* abs/1603.01250 (2016). arXiv:1603.01250 http://arxiv.org/abs/1603.01250

[46] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. 2019. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 455–468. https://doi.org/10.1145/3297858.3304021

[47] Jean-loup Gailly and Mark Adler. [n.d.]. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://zlib.net/.

[48] Yang Ji, Sangho Lee, Mattia Fazzini, Joey Allen, Evan Downing, Taesoo Kim, Alessandro Orso, and Wenke Lee. 2018. Enabling Refinable Cross-Host Attack Investigation with Efficient Data Flow Tagging and Tracking. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1705–1722. https://www.usenix.org/conference/usenixsecurity18/presentation/jia-yang

[49] Khronos Group. [n.d.]. Khronos Conformant Products. https://www.khronos.org/conformance/adopters/conformant-products.

[50] Sumin Kim, Seunghwan Oh, and Youngmin Yi. 2021. Minimizing GPU Kernel Launch Overhead in Deep Learning Inference on Mobile GPUs. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications* (Virtual, United Kingdom) *(HotMobile '21)*. Association for Computing Machinery, New York, NY, USA, 57–63. https://doi.org/10.1145/3446382.3448606

[51] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. uLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages. https://doi.org/10.1145/3302424.3303950

[52] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 8343–8354. https://proceedings.neurips.cc/paper/2020/file/5f0ad4db43d8723d18169b2e4817a160-Paper.pdf

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

[54] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. 2019. Occlumency: Privacy-Preserving Remote Deep-Learning Inference Using SGX. In *The 25th Annual International Conference on Mobile Computing and Networking* (Los Cabos, Mexico) *(MobiCom '19)*. Association for Computing Machinery, New York, NY, USA, Article 46, 17 pages. https://doi.org/10.1145/3300061.3345447

[55] The Mesa 3D Graphics Library. [n.d.]. Mesa for Panfrost. https://docs.mesa3d.org/drivers/panfrost.html.

[56] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver . https://www.kernel.org/doc/html/latest/gpu/v3d.html.

[57] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: register accessors. https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_drv.h#L170.

[58] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: v3d_clean_caches(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_gem.c#L189.

[59] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: v3d_csd_job_run(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_sched.c#L245.

[60] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: v3d_irq(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_irq.c#L85.

[61] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: v3d_mmu_insert_ptes(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_mmu.c#L87.

[62] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: v3d_reset(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_gem.c#L110.

[63] Linux. [n.d.]. drm/v3d Broadcom V3D Graphics Driver: wait_for(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/v3d/v3d_drv.h#L290.

[64] Linux. [n.d.]. drm/vc4 Broadcom VC4 Graphics Driver . https://www.kernel.org/doc/html/latest/gpu/vc4.html.

[65] Linux. [n.d.]. drm/vc4 Broadcom VC4 Graphics Driver: submit_cl(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_gem.c#L369.

[66] Linux. [n.d.]. drm/vc4 Broadcom VC4 Graphics Driver: vc4_irq(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_irq.c#L196.

[67] Linux. [n.d.]. drm/vc4 Broadcom VC4 Graphics Driver: vc4_reset(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/vc4/vc4_gem.c#L286.

[68] Linux. [n.d.]. Qualcomm adreno graphics driver. https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno.

[69] Linux. [n.d.]. Qualcomm adreno graphics driver: a6xx_flush(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L54.

[70] Linux. [n.d.]. Qualcomm Adreno Graphics Driver: a6xx_irq(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L1046.

[71] Linux. [n.d.]. Qualcomm Adreno Graphics Driver: a6xx_recover(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L934.

[72] Linux. [n.d.]. Qualcomm Adreno Graphics Driver: a6xx_submit(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/adreno/a6xx_gpu.c#L140.

[73] Linux. [n.d.]. Qualcomm Adreno Graphics Driver: msm_gpummu_map(). https://elixir.bootlin.com/linux/latest/source/drivers/gpu/drm/msm/msm_gpummu.c#L28.

[74] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. 2020. DarkneTZ: Towards Model Privacy at the Edge Using Trusted Execution Environments. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. Association for Computing Machinery, New York, NY, USA, 161–174. https://doi.org/10.1145/3386901.3388946

[75] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 417–429. https://www.usenix.org/conference/atc15/technical-session/presentation/netravali

[76] NVIDIA. [n.d.]. CUDA Compatibility. https://docs.nvidia.com/deploy/cuda-compatibility/index.html.

[77] NVIDIA. [n.d.]. Host1x Hardware Description. http://http.download.nvidia.com/tegra-public-appnotes/host1x.html.

[78] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver. https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=summary.

[79] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: __gk20a_do_unidle(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/os/linux/module.c;h=807df2cadfbc6d1d76008021e5dfabdea232c72b;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l630.

[80] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: gk20a_mm_l2_flush(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/gk20a/mm_gk20a.c;h=10ca84d9dfe23b4adfb607dc50041abbc2216217;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l541.

[81] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: gk20a_wait_for_idle(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/gk20a/gk20a.c;h=c3068b76ccb08695621292b5d7f354d9c4785732;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l441.

[82] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: gm20b_tegra_unrailgate(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/os/linux/platform_gk20a_tegra.c;h=c39e4f0e6cdbfe37f880a05cbe17af0fa9af604c;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l368.

[83] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: __nvgpu_gmmu_update_page_table(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/common/mm/gmmu.c;h=748e9f455ca33d2c9388dc789d9696616f4dfbe5;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l591.

[84] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: nvgpu_submit_channel_gpfifo(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/common/fifo/submit.c;h=b0f38ff1cfa48ba36111e6734aa2017415aba575;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l318.

[85] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: nvgpu_submit_prepare_syncs(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvgpu.git;a=blob;f=drivers/gpu/nvgpu/common/fifo/submit.c;h=b0f38ff1cfa48ba36111e6734aa2017415aba575;hb=7bf2833f340f87ea643d3ef66b0e4c22ffb1e891#l40.

[86] NVIDIA. [n.d.]. NVIDIA Tegra Linux Driver: syncpt_thresh_cascade_isr(). https://nv-tegra.nvidia.com/gitweb/?p=linux-nvidia.git;a=blob;f=drivers/video/tegra/host/host1x/host1x_intr.c;h=c0fd8611273c65619116c704dcd462903b80036c;hb=6dc57fec39c444e4c4448be61ddd19c55693daf1#l39.

[87] NVIDIA. 2021. NVIDIA MULTI-INSTANCE GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[88] Heejin Park, Shuang Zhai, Long Lu, and Felix Xiaozhu Lin. 2019. StreamBox-TZ: Secure Stream Analytics at the Edge with TrustZone. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 537–554. https://www.usenix.org/conference/atc19/presentation/park-heejin

[89] Hugh Perkins. [n.d.]. OpenCL library to train deep convolutional networks. https://github.com/hughperkins/DeepCL.

[90] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. 2016. CUDA Leaks: A Detailed Hack for CUDA and a (Partial) Fix. 15, 1, Article 15 (Jan. 2016), 25 pages. https://doi.org/10.1145/2801153

[91] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: A Remote Execution Based Record-and-Replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 571–582. https://doi.org/10.1145/2884781.2884854

[92] Quake3 World. [n.d.]. Quak3: Demo System. https://www.quake3world.com/q3guide/demos.html.

[93] Raspberry Pi Foundation. [n.d.]. RaspberryPi Firmware Mailbox property interface. https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface.

[94] rsta2. [n.d.]. A C++ bare metal environment for Raspberry Pi with USB (32 and 64 bit). https://github.com/rsta2/circle/.

[95] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *Proc. USENIX ATC*. USENIX Association, Philadelphia, PA, 109–120. https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki

[96] Tencent. [n.d.]. ncnn Pipeline Cache. https://github.com/Tencent/ncnn/blob/master/src/pipelinecache.cpp/.

[97] Tencent. [n.d.]. Tencent ncnn framework. https://github.com/Tencent/ncnn.

[98] TensorFlow. [n.d.]. Example NNs by TensorFlow. https://www.tensorflow.org/lite/guide/hosted_models.

[99] Florian Tramer and Dan Boneh. 2019. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJVorjCcKQ

[100] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 681–696. https://www.usenix.org/conference/osdi18/presentation/volos

[101] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 331–344. https://doi.org/10.1109/HPCA.2019.00048

[102] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. 2011. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) *(USENIXATC'11)*. USENIX Association, USA, 29.

[103] Sam (Likun) Xi, Yuan Yao, Kshitij Bhardwaj, Paul Whatmough, Gu-Yeon Wei, and David Brooks. 2020. SMAUG: End-to-End Full-Stack Simulation Infrastructure for Deep Learning Workloads. *ACM Trans. Archit. Code Optim.* 17, 4, Article 39 (Nov. 2020), 26 pages. https://doi.org/10.1145/3424669

[104] Chao Xu, Felix Xiaozhu Lin, Yuyang Wang, and Lin Zhong. 2015. Automated OS-level Device Runtime Power Management. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/2694344.2694360

[105] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 2125–2136. https://doi.org/10.1145/3308558.3313591

[106] M. Yan, Y. Shalabi, and J. Torrellas. 2016. ReplayConfusion: Detecting cache-based covert channel attacks using record and replay. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14.

[107] Zhihao Yao, Zongheng Ma, Yingtong Liu, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. 2018. Sugar: Secure GPU Acceleration in Web Browsers *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 519–534. https://doi.org/10.1145/3173162.3173186

[108] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 807–825. https://doi.org/10.1145/3373376.3378466

[109] J. Zhu, R. Hou, X. Wang, W. Wang, J. Cao, B. Zhao, Z. Wang, Y. Zhang, J. Ying, L. Zhang, and D. Meng. 2020. Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1450–1465. https://doi.org/10.1109/SP40000.2020.00054